Week 5

Exercise 40

../40/40.txt

```
1. Pointer variables and arrays
They are very similar, in that declaring a size_t array of size 10, i.e. size_t array
    [10] is actually just a pointer to the first element of that array (i.e. array[0]
     = *array). The difference lies in the fact that the location that an array
    points to is immutable, whereas a pointer variable can be changed.

2. Pointer variables and reference variables
See the drawing below (Figure 1).

3. Pointer arithmetic
An example of this can be found in Figure 1, part b. It refers to the fact that
    pointers of a certain type can be incremented or decremented to reach the next
    element from its starting position. For example, given an integer array named '
    intArray', defining an integer pointer *intArray will point it to the start of
    said array. Thus, *intArray + 1 will point towards the second element in that
    array, as the pointer now points one integer-sized storage block further than the
     start of said array. Or rather, it points towards the addresses associated
    therewith.

4. Accessing an element in an array using only a pointer vs. index expression
Using a pointer will skip a step when accessing an element, which is to determine the
     size of the array element and add that to the address of the first element.
    Instead, it can simply move over the size of a single element over and over again
    . In the exercise, size_t is mentioned, of which the size is known and constant
    and so elements can be accessed directly. This can be especially true when
    elements are repeatedly accessed, such as in loops, resulting in cumulative
    benefits. However, personally, I feel that the index expressions establish a
    closer link to mathematical equivalents, such as matrices, which makes their use
    more accessible, and I wonder how much of the advantages are still present given
    the state of compiler optimisation.
```

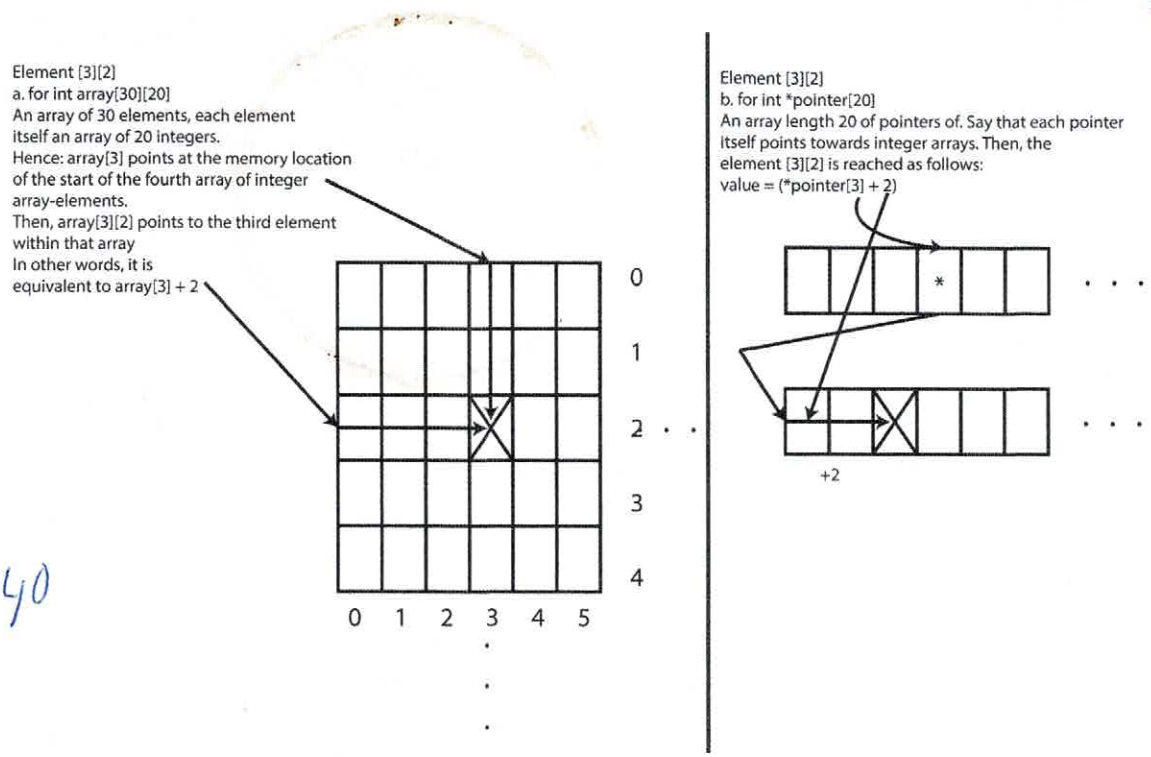

Figure 1: Illustration

Exercise 41

../41/main.cc

```
1  // Iterating over environ and argv: main file
2
3  #include "main.ih"
4
5  int main(int argc, char const *argv[])
6  {
7    extern char **environ;
8
9    for (size_t index = 0; index != argc; ++index)  // For elements of argv (0-argc)
10     cout << environ[index] << '\n';                // print associated env var
11
12   for (size_t index = 0; environ[index] != nullptr; ++index)  // For all elements
13     cout << argv[index] << '\n';                               // of environ[] print
14 }                                                              // associated argv
```

declarations in the .ih file, please

0?

first here environ's contents are printed, not argv's

and here argv's contents. Also: most likely a segfault. At least an array bound error

(041)

Exercise 43

../43/43.txt

definition:	rewrite:
`int x[8];`	`x[3] = x[2];`
pointer notation:	`*(x + 3) = *(x + 2)`
semantics:	x + 3 points to the location of the 3th int beyond x. Which is set to be equal to x + 2 which is the location of the 2nd int beyond x.
`char *argv[8];`	`cout << argv[2];`
pointer notation:	`cout << *(argv + 2);`
semantics:	argv + 2 points to the location of the 2nd argument beyond the first argument which is the programs name. Which is then passed to cout.
`int x[8];`	`&x[10] - &x[3];`
pointer notation:	`&*(x+10)-&*(x+3)`
semantics:	*(x+10) points to the the 10th int beyond x. Then the reference & makes it instead return its location. The same happens for &*(x+3). Since one points to the 3rd int beyond x and the other points to the 10th int beyond x the result is the difference 10 - 3 = 7.
`char *argv[8];`	`argv[0]++;`
pointer notation:	`*argv = *argv + 1;`
semantics:	*argv then points to the start of the programs name. Then by adding 1 to it, it still points to the programs name. But now it points to 1 byte beyond the start of the programs name. Such that the programs name would be /45 instead of the initial ./45 if it were passed to cout.
`char *argv[8];`	`argv++[0];`
pointer notation:	`*(argv + 1)`
semantics:	*(argv + 1) points to the first argument beyond the programs name.
`char *argv[8];`	`++argv[0];`
pointer notation:	`1 + *(argv)`
semantics:	1 + *(argv) then points to the start of the programs name. Then by adding 1 to it, it still points to the programs name. But now it points to 1 byte beyond the start of the programs name. Such that the programs name would be /45 instead of the initial ./45 if it were passed to cout.
`char **argv;`	`++argv[0][2];`
pointer notation:	`+++(*(argv) + 2)`
semantics:	First the outer pointer (the 2) points to the 2nd column. Then the inner pointer points the the 0th element in the 2nd row. Then 1 is added to its contents. Which is of type char.