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.
```

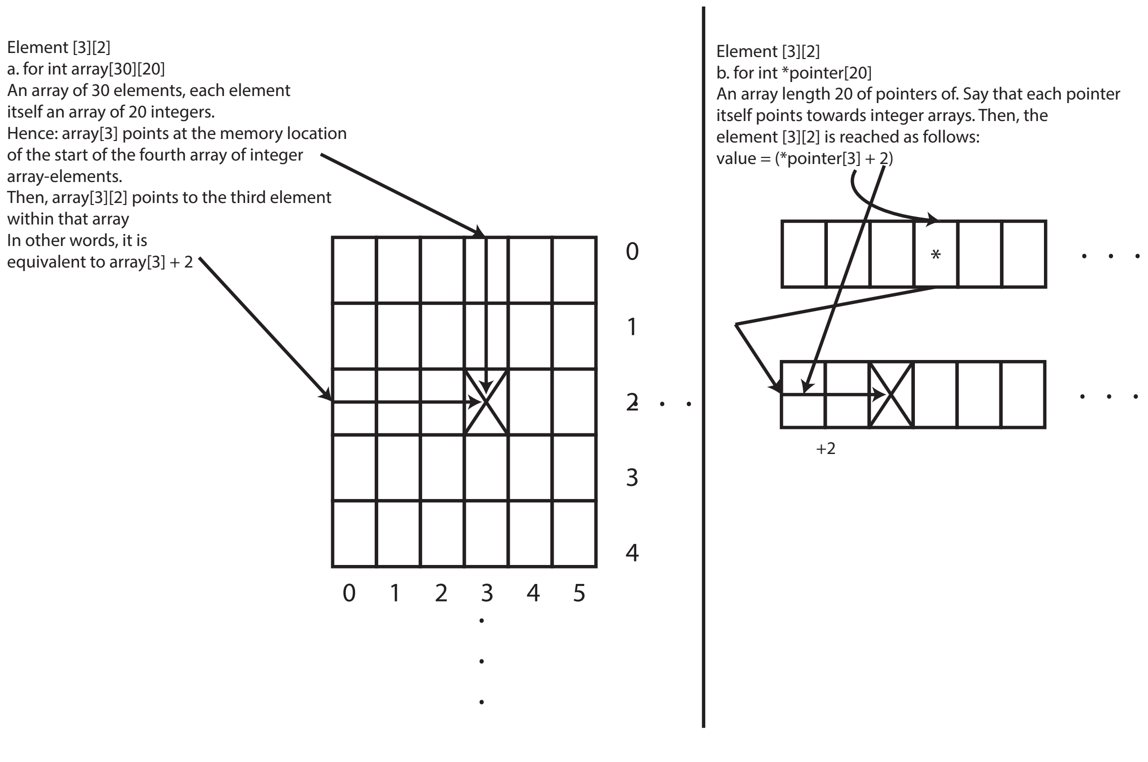
Element [3][2]
a. for int array[30][20]
An array of 30 elements, each element
itself an array of 20 integers.
Hence: array[3] points at the memory location
of the start of the fourth array of integer
array-elements.
Then, array[3][2] points to the third element
within that array
In other words, it is
equivalent to array[3] + 2
0
1
2
3
4
0 1 2 3 4 5
Element [3][2]
b. for int *pointer[20]
An array length 20 of pointers of. Say that each pointer
itself points towards integer arrays. Then, the
element [3][2] is reached as follows:
value = (*pointer[3] + 2)
*
+2

Figure 1: Illustration

Exercise 41

../41/main.cc

```
// Main file

#include "main.ih"

int main(int argc, char *argv[])
{
  size_t distanceEnvArgv = *environ - *argv;  // Determine distance env and environ

  *argv += distanceEnvArgv;      // Move argv to position environ
  *environ -= distanceEnvArgv;   // Move environ to position argv

  for (size_t index = 0; argv[index] != 0; ++index) // Loop through "argv"
    cout << argv[index] << '\n';                     // And print elements

  for (size_t index = 0; environ[index] != 0; ++index)  // Loop through "environ"
    cout << environ[index] << '\n';                     // And print elements
}
```

Exercise 42

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.
----------------------------------------------------------------------------
```

Exercise 44

../44/main.ih

```cpp
// Main: internal header file

using namespace std;

#include <cstddef>

enum FIXEDVARS
{
  DIM = 10
};

void inv_identity(int (*entryRow)[DIM]);
void allOnes(int (*entryRow)[DIM]);
void diagZeroes(int (*entryRow)[DIM]);

void printArray(int const (*square)[DIM]);  // Testing purposes

// It does seem nicer to write a separate header file for these functions.
// Would it suffice to move them to a subdirectory and include a /matrixF/matrixF.h
// or /matrixF/matrixF.ih in main.ih?
```

../44/main.cc

```cpp
// Main file

#include "main.ih"

int main()
{
  int square[DIM][DIM]; // Declare square 2D array

  int (*row)[DIM] = square;  // Define row as pointing to rows of 2D array

  inv_identity(row);  // Pass row to function
  // printArray(square); // Only for testing purposes
}
```

../44/inv_identity.cc

```cpp
// Matrix function: make array into inverted identity matrix

#include "main.ih"

void inv_identity(int (*entryRow)[DIM])
{
  allOnes(entryRow);    // Make all entries ones
  diagZeroes(entryRow); // Make diagonal zeroes
};
```

../44/allOnes.cc

```cpp
// Matrix function: make 2D array into matrix of ones

#include "main.ih"

void allOnes(int (*entryRow)[DIM])
{
  for (int (*row)[DIM] = entryRow; row != entryRow + DIM; ++row)
  {
    for (int *column = *row, *end = column + DIM; column != end; ++column)
```

```
      (*column) = 1;
  }
};
// Loop through all the rows, then within those rows the columns (now individual
// elements) and set them all to one.
```

../44/diagZeroes.cc

```
// Matrix function: make diagnonal zeroes

#include "main.ih"

void diagZeroes(int (*entryRow)[DIM])
{
  for (int *entry = *entryRow, index = 0; index != DIM; ++index, entry += DIM + 1)
    (*entry) = 0;
};
// Sets every eleventh (dimension + 1) element to zero
```

Week 5

Exercises 45 & 47

../45–47/main.ih

```
// Main file: internal header

#include "strings/strings.h"

#include <string>
#include <iostream>

extern const char **environ;

using namespace std;
```

../45–47/main.cc

```
// Main file
// This is just an example main file to demonstrate the workings of the Strings class
// The constructors will also work for other NTBSs, but environ and argc/argv
// are convenient examples to use.

#include "main.ih"

int main(int argc, char const **argv)
{
  Strings objectA = Strings(cin);     // Create Strings using cin
  Strings objectB = Strings(environ); // Create Strings using environ
  Strings objectC = Strings(argc, argv);  // Create Strings based on argc, argv

  Strings::stringsSwap(objectA, objectB); // Swap environ and istream Strings

  // objectA.printStrings(); // Print what is now environ Strings
  // objectB.printStrings(); // Print what is now istream Strings
  // objectC.printStrings(); // Print the unchanged objectC
  // These are for testing purposes
}
```

../45–47/strings/strings.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <cstddef>
#include <string>
// #include <ioforward>

class Strings
{
  size_t d_size = 0;       // Number of elements in d_str
  std::string *d_str = 0; // Stored strings

  public:
    Strings(size_t numStrings, char const **strings); // argc, argv constructor
    Strings(char const **strings);                    // environ constructor
    Strings(std::istream &input);                     // istream constructor
    Strings();                                        // default constructor

    void printStrings() const;                        // Just for testing

    // 46
    size_t size() const;
    // std::string* data(); // Not implemented
    // std::string* at(size_t index, bool) const; // Not implemented
    // std::string* at(size_t index); // Not implemented

    // 47
    static void stringsSwap(Strings &objectA, Strings &objectB);

  private:
    void add(char const *novelString);          // Add char array to d_str
};

#endif
```

../45–47/strings/strings.ih

```
#include "strings.h"
#include <iostream>
//#define CERR std::cerr << __FILE__": "

using namespace std;
```

../45–47/strings/addChar.cc

```
#include "strings.ih"

void Strings::add(char const *novelString)
{
  std::string *temporary = new string[d_size + 1];
  // Create a pointer temporary that points towards a newly allocated
  // piece of memory in which an array of
  // d_size + 1 initialised strings are held

  for (size_t index = 0; index != d_size; ++index)
    temporary[index] = d_str[index];
  // Transfer over the current array of strings to temporary

  temporary[d_size] = novelString;
  // Add the new element to the end of temporary

  delete[] d_str;
  // Delete/deallocate the memory currently pointed at by d_str

  d_str = temporary;
  // Point d_str to the memory pointed at by temporary

  ++d_size;
  // Increment d_size
}
```

../45–47/strings/c_argcargv.cc

```
#include "strings.ih"

Strings::Strings(size_t numStrings, char const **strings)
{
  std::cout << "Argc / argv constructor called. \n";

  for (size_t index = 0; index != numStrings; ++index)
    add(strings[index]);
  // For NTBSs 0 to numStrings within strings, pass them to the add function
}
```

../45–47/strings/c_default.cc

```
#include "strings.ih"
```

```

Strings::Strings()
{
  std::cout << "Default constructor called. \n";
};
```

../45–47/strings/c_environ.cc

```
#include "strings.ih"

Strings::Strings(char const **strings)
{
  std::cout << "environ constructor called. \n";

  for (size_t index = 0; strings[index] != 0; ++index)
    add(strings[index]);
  // For NTBSs 0 to when a null char is encountered, pass them to the add function
};
```

../45–47/strings/c_istream.cc

```
#include "strings.ih"

Strings::Strings(std::istream &input)
{
  std::cout << "istream constructor called. \n"
            << "Enter an empty line (enter/return) to hault input. \n";

  std::string newEntry; // Define string newEntry
  while (getline(input, newEntry))  // Loop while getline works, setting
  {                                 // newEntry to the new line
    if (newEntry.empty()) // If getline creates an empty string
      break;  // Break out of the whie loop (happens when enter/return is pressed)

    add(newEntry.c_str());  // Call the add using the newly entered string.
    // Note that the string is converted to a NTBS to work with the add
    // function. Alternatively another add function could be written.
  }
}
```

../45–47/strings/stringsSwap.cc

```
#include "strings.ih"

void Strings::stringsSwap(Strings &objectA, Strings &objectB)
{
  Strings temporary = objectA;
  // First, a Strings object temporary is created using an implicit (i.e. non-user
  // defined) / trivial copy constructor. In other words, temporary is constructed
  // based on a constant reference to objectA, thus temporary is now a copy of
      objectA
  // (in a new location in memory). Strings temporary(objectA); would do the same.
  objectA = objectB;
  // This a default class assignment. Now, both objectA and objectB point to the same
  // memory, which must be remedied.
  objectB = temporary;
  // This assigns objectB to the same memory as temporary.
  // Since temporary is not destroyed, this solution works fine, but really an
  // overloaded assignment operator and copy constructor should be written.
};
```