Exercises **Session 3**

November 30

Problem 1: Matrix and vector classes

In this exercise we will create two classes, `Vector` and `Matrix`, that can carry out basic array functionality as well as the mathematical operations we expect from such objects.

- Create the two classes in question and set up their data stucture (privately)
- Write suitable constructors and destructors
- Create some sort of access operators, how would you solve that for the matrix?

  Advanced: Create a `VectorView` class to read individual rows of the matrix, use this class to recreate the `[][]` syntax of normal multidimensional arrays
- Create `.size()` and `.resize(...)`
- Make some convinient filling functions, such as zero to zero, or the identity
- Add other functions you feel your objects should have, such as `.norm()` and `.transpose()`
- Now it is time to overload the various arithmetic operators, `+`, `-`, `*` (`/`).
  How does this work with combined expressions? `(a+b)*c`

Problem 2: Address book

In this exercise we will implement an address book where all the entries are sorted based on the contact's name. To store the addresses we will create a linked list storage class.

- Create the `AddressBookEntry` class / struct. Every address should at least have a name, and address and a phone number. Decide how you want to store the various pieces of information and which ones you feel are important
- Implement the necessary boolean operators so that you can later compare entries

Next we will make the linked list to store the entries. A linked list consists of two classes. The overlying control structure that will manage the list interface (`LinkedList`), and nodes that contain the linked list data (`LinkedListNode`). The liked list doesn't keep an array of the nodes, only the first node of the list, then every node contains its data along with some sort of way to pointing to the next node in the chain.

- Create the `LinkedListNode` class and give it data, a member function to access the underlying data as well as `.next()` and `.last()` functions
- Create the `LinkedList` class that work as an outer interface and enables functionality such as `.insert()`, `.remove()`, `.search()`

- Adapt this to work for you `AddressBook` class so that you have a working address book that is sorted and can be manipulated

Problem 3: Minesweeper

Finally we will implement a terminal based minesweeper game. There are many ways of structuring your code to implement this, but here are a couple of classes you can start out with:

- A `Minesweeper` class that encapsulates the game. It should be able to parse input and make those into commands on the game board. It should also manage looping until a win or loose condition is met along with tying the other parts of the game together
- A `MinesweeperBoard` class that keeps track of the mines on the board as well as the mine adjacency number (the one that says how many mines surround an empty tile)
- A `MinesweeperMask` class that works as a mask in top of the board. It is the layer that is between the board and the player and it contains information such as whether a tile is open or closed as well as whether the player has made a guess on the tile

Sit down and think about how these objects would interact with each other before you start programming, and as you write code you might realise that more helper classes could be useful (e.g. a separate IO layer).

Advanced: If you have played minesweeper before, you know that if you open a tile that has no adjacent mines, the game will automatically also open all adjacent tiles. How can you implement a similar feature in your program using recursion?

Advanced: Another feature you might want is the option to "control click" an opened tile. In minesweeper this works so that if a tile has the number one on it, and you as a player has made a sufficient amount of guesses around it, the game will open all the other nodes for you. This is also a cascading effect.