Exercise Sheet #3

How to use the OLAT system
Solutions should be handed in through the OLAT webpage:

(a) Follow the course link: `https://olat-ce.server.uni-frankfurt.de/olat/auth/RepositoryEntry/12621676550/CourseNode/1616555215421696005`.

(b) On the side toolbar to the left click on 'Enrolment' and sign up to your respective tutorial group.

(c) Under 'Homework sheets', select the current sheet and upload your solution. You can upload a .zip file if you have multiple files.

(d) Remember: Name your file by your name and problem number, e. g. `SmithJoe_problem2a_additionalInfo.xyz`. **Never** use a space-character in a file name!

Problem 1 (*Battleship game*) 10 Pts

The well-known battleship game is played by two players, each of them having two 10×10 square grids, which play the role of a battlefield and a tracking map. On one grid the player arranges ships and records the shots by the opponent. On the other, tracking grid the player records their own shots, trying to find all the battleships of the opponent. The rows and columns of the grids are labelled by capital letters and numbers, respectively. As a starting setup, both players place the ships of different sizes at random positions on their primary grid. The set of given ships is:

#	Class of ship	Size
1×	Carrier	5
1×	Battleship	4
1×	Cruiser	3
2×	Destroyer	2
2×	Submarine	1

Write a `C++` code to implement the one-player (asymmetric) version of the game, i. e. a player against the computer, but the computer is not shooting back. The player shall see a tracking grid for the game, while the battlefield of the computer is not shown. The game proceeds in series of rounds. In each round the player launches a rocket targeting a certain position (a grid cell) of the battlefield, the tracking grid is then updated and refreshed, indicating

whether the the shot was successful or not.

As a starting point you can use the code presented in the lecture following the instructions below:

- write a method to randomly place the ships on the battlefield at different positions and orientations (ships shall not overlap, and must be fully on the field)
- implement a function for checking whether the player's shot hit a ship of the computer
- mark the position of the hit on the tracking grid, placing on `o/x` corresponding to successful/unsuccessful trials
- update the screen after every round

Problem 2 (*Pointers*) 10 Pts

(a) Define three pointers of the type `int`, `double`, `long double` that each point to an array of the corresponding type. Print the addresses of the arrays (i. e. the values of the pointers) and the value that the pointer represents. Then increment the pointers (e. g. by `++p` or `p=p+1`) and print both the value of the pointer and the element it points. What do you find for the different data types?

(b) With `reinterpret_cast<intptr_t>(p)`, you can convert the hexadecimal representation of a pointer `p` into decimal (for more information, see `https://en.cppreference.com/w/cpp/language/reinterpret_cast` and `https://en.cppreference.com/w/cpp/types/integer`). For the aforementioned data types, reprint the decimal addresses of two consecutive elements in each array. The jumps in the address numbers should correspond to the memory needed to store these data types.

(c) Write a function `int myfunc(int &x, int &y)` that takes two integers by reference and returns their product. Rewrite the function such that it takes two pointers as arguments instead.

(d) Write a function `double scalarProduct(double *arr1, double *arr2)` that returns the scalar product of two vectors stored in `arr1`, `arr2`.

(e) Now implement the vector product of two vectors. Therefore you should create the function `void outerProduct(double *arr1, double *arr2, double *result)`, where the two vectors are stored in `arr1`, `arr2` and

the result is to be stored in `result`. This structure is the typical way to return arrays from functions.

Problem 3 (*Sorting Algorithm, advanced!*) 10 Pts

A sorting algorithm is an algorithm that puts elements of a list in a certain order, we will consider numerical order of lists of integers.

Implement the following sorting algorithms:

- Radix Sort `https://en.wikipedia.org/wiki/Radix_sort`
- Bubble Sort `https://en.wikipedia.org/wiki/Bubble_sort`
- Another sorting algorithm of your choice `https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms`

These algorithms can be classified by their average performance on randomly generated lists. This is important because many algorithms need a different number of steps depending on the particular set of numbers in the list.

Analyse the algorithms' average performance:

- Sort $N_l = 1000$ different lists, containing n random integer numbers having up to $n_d = 4$ digits, recording the average run-time.
- Analyse the scaling of the average run-time as a function of $n \in [100, ..., 10000]$.
- Compare the scaling that you find for the different algorithms with each other and with the theoretical scaling.
- Plot and discuss these results.

Hint: you can measure the run-time within C++ with the `chrono` library. To compile the example on the next page, you need to set the `-std=c++11` option in the g++ compiler.

```cpp
#include <iostream>
// chrono library, a collection of types that track the system time.
#include <chrono>

using namespace std;
// include the namespace of the chrono library
using namespace std::chrono;

/* dummy function whose execution takes a measurable amount
of time by adding a lot of numbers in a for loop.*/
void function(){
    for( int k = 0; k < 1000; ++k ){
        long long number = 0;
        for( long long i = 0; i < 2000000; ++i ){
            number += 5;
        }
    }
}

// main function
int main(){
    // store the current time in t1 at the highest resolution available
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    // run the dummy function
    function();
    // store the current time after the function execution
    high_resolution_clock::time_point t2 = high_resolution_clock::now();

    /* duration_cast is a function template that converts a
    std::chrono::duration to a duration of different type,
    in this case microsecs.
    count() returns the count of ticks in the interval.*/
    int dur_count = duration_cast<microseconds>( t2 - t1 ).count();

    // convert into units of seconds
    double dur_sec = ((double) dur_count)/1e6;

    // print the duration in seconds
    cout << "Execution time in seconds: " << dur_sec << endl;
    return 0;
}
```