Solutions to this sheet are due on 15.05.2020 til 16:00. Please hand in a digital version of your answers via e-mail. The e-mail’s subject has to contain ccpp20. Do zip-compress your solutions. **Note:** If you copy text elements/code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited. All of your files that belong to your solution have to be contained in a single .zip file that is named according to the following naming scheme: `<name>_<surname>_solution_<X>.zip`. Replace `<name>` and `<surname>` with your actual name and replace `<X>` with the number of the exercise sheet. You can look up your results using this link [here](https://docs.google.com/spreadsheets/d/1LtRFGuJ2kKxpo1V34UJfKgX74ILbO5oLzhp1Dhx926FZ1cS3/edit?usp=sharing).

This exercise sheet is about functions. Furthermore, you will make yourself familiar with some other useful container type. And at last, you will have a quick look at pointers. You are free to use the code snippets provided at [here](https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/SS2020/code_02.zip).

You can achieve 16 points in total.

**Exercise 1.**

In this exercise, you will implement some basic linear algebra using C++. In particular, you will implement a few functions that perform some useful operations on mathematical vectors. We will use `std::vector` to represent a mathematical vector $v \in \mathbb{R}^n$. Write a program that implements a function for each of the following tasks. Check your function implementations by calling them on small test data as shown in the following code snippet:

```cpp
#include <cmath>
#include <iostream>
#include <vector>

using namespace std;

void print_dvector(const vector<double> &v) {
    for (const double &d : v) {
        cout << d << " \n";
    }
    cout << "\n";
}

double euclidean_length(const vector<double> &v);
```
```cpp
double scalar_product(const vector<double> &v, const vector<double> &w);
vector<double> normalize(vector<double> v);
double euclidean_distance(const vector<double> &v, const vector<double> &w);

int main()
{
    vector<double> a = {1, 2, 3};
    vector<double> b = {4, 5, 6};
    // You have to provide the implementations for the four function declarations
    // in above to make this code work.
    cout << "length of 'a': " << euclidean_length(a) << '\n';
    cout << "scalar product of 'a' and 'b': " << scalar_product(a, b) << '\n';
    print_dvector(normalize(a));
    cout << "distance between 'a' and 'b': " << euclidean_distance(a, b) << '\n';
    return 0;
}
```

Implement a function called ...

a) `euclidean_length` that computes the euclidean length of a vector. (1 P.)

The euclidean length of a vector \( v \in \mathbb{R}^n \) is defined as 
\[ ||v|| = \sqrt{\sum_{i=1}^{n} v_i^2}. \]

b) `scalar_product` that computes the scalar product of two vectors. (1 P.)

The scalar product \( \langle \cdot, \cdot \rangle \) of two vectors \( x, y \in \mathbb{R}^n \) is defined as 
\[ \langle x, y \rangle = \sum_{i=1}^{n} x_i \cdot y_i. \]

c) `normalize` that computes a normalized version of a vector. (1 P.)

A normalized vector can be obtained by dividing each of its entries by its (euclidean) length.

d) `euclidean_distance` that computes the euclidean distance of two vectors. (1 P.)

The euclidean distance of two vectors \( x, y \in \mathbb{R}^n \) is defined as 
\[ ||x - y||_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \]

**Exercise 2.**

Fibonacci numbers are numbers from an integer sequence, called Fibonacci sequence. Every number in this sequence is the sum of the two preceding ones: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... The Fibonacci sequence \( F_n \) can be defined by the following recurrence relation (recursion):

\[ F_1 = 1, \ F_2 = 1, \ F_n = F_{n-1} + F_{n-2} \]

a) Implement the function `unsigned fibonacci_rec(unsigned n)` such that it calculates the \( n \)-th Fibonacci number by using the recursive definition from above. (2 P.)

b) Implement the function `unsigned fibonacci_nonrec(unsigned n)` such that it computes the \( n \)-th Fibonacci number but uses sequential code rather than recursion. (Hint: use three variables and a loop.) (2 P.)

c) Compute the 50-th Fibonacci number using both of your Fibonacci implementations. Is there a noticeable difference in the runtime? Why does the recursive version takes so much longer to compute? (Note: it is not the negligible overhead caused by a function call.) (2 P.)
Exercise 3.
Declare a variable `mymap` of type `std::map`<string, int> which is declared in the standard template library. (Use `#include <map>`.)
Please refer to [http://en.cppreference.com/w/cpp/container/map](http://en.cppreference.com/w/cpp/container/map) on how to use `std::map`. You can find detailed descriptions as well as example code. Have a look at the member functions (constructor), `operator[]` and the corresponding examples.

a) Add the following tuples to `mymap` that map a person’s name to their age: ("Peter", 40), ("Brian", 4), ("Stewie", 1), ("Chris", 15), ("Meg", 14). (1 P.)

b) Write a function that prints the contents of `mymap` to the command line. (2 P.)

c) Add the tuple ("Lois", 41) to `mymap` and print the contents of the map again. (1 P.)

Exercise 4.
We have already learned that pointer and reference types can be quite useful. We also discussed that one can represent points-to relationships as a graph. Consider the following (not very useful) code snippet:

```c++
int i, j, k;
int *a = &i;
int *b = &k;
int **p = &a;
int **q = &b;
int *c = *q;
```

a) Watch the following video that provides an excellent introduction to pointers: [https://youtu.be/Rxv59krECNw?t=4m18s](https://youtu.be/Rxv59krECNw?t=4m18s) (0 P.)

b) Draw the corresponding directed graph that captures the points-to relations of the above code snippet. Use nodes to represent variables and directed edges to represent points-to information. Annotate each node with its respective variable’s name and type. (2 P.)