Einführung in die Programmierung für Physiker
WS 2019/2020 – Marc Wagner

Alessandro Sciarra: sciarra@itp.uni-frankfurt.de

Exercise sheet 9

To be corrected in tutorials in the week from 16.12 to 20.12.2019

Exercise 1 [*Dynamic memory allocation*]

In the lecture you discovered how to reserve memory for your program at run time and you also discussed in detail one possible way to allocate memory to store a matrix later on. The presented code looked similar to the following.

```
void allocateMatrix(int*** matrix, int rows, int columns){
    *matrix = (int**)malloc(rows * sizeof(int*));
    if( *matrix == NULL){
        printf("There has been an error allocating memory!\n");
        exit(1);
    }
    for(int i = 0; i < rows; ++i){
        (*matrix)[i] = (int*)malloc(columns * sizeof(int));
        if((*matrix)[i] == NULL){
            printf("There has been an error allocating memory!\n");
            exit(1);
        }
    }
}
```

Starting from the above code, consider the following tasks.

(i) Define a matrix of type `int**` in the `main` function, allocate memory using the above function and initialize it to some value.

(ii) How is the function `allocateMatrix` reserving memory? Is the allocated chunk of memory contiguous?

(iii) You should know that it is possible to take differences of memory addresses. Devise a way to understand how far away in memory are two variables. The `sizeof` operator might be useful in this respect. Use your method to check your answer to task (ii).

(iv) Implement a `allocMatrix_contiguous` function which allocates memory in a contiguous way, i.e. that reserves a chunk of `rows*columns*sizeof(int)` bytes of memory. What should you do to still be able to access the matrix elements using the access operator, i.e. via the `matrix[][]` notation?

(v) Check that your allocation is indeed contiguous and that `size(int)` is indeed the distance between contiguous elements in the matrix. In this regard, e.g., `matrix[0][columns-1]` and `matrix[1][0]` should be contiguous. Why might this approach be advantageous?

Exercise 2 [*Const-correctness*]

The use of the `const` keyword in `C` is, strictly speaking, not mandatory, but it helps us to defend ourselves... from ourselves! Actually, there are typical use cases, one of which we will analyse in a moment. Before, let us clarify the meaning of the keyword and how to use it, especially together with pointers. What `const` means should be clear from the keyword itself. *Something* is marked as constant and it cannot be changed (and this allows the compiler to give you an error if you try to violate this rule you set yourself). However, it is very important to understand *what* exactly is marked as constant.

Consider now the following signatures of a function which takes a number and returns its square root.

```
double sqrt_I(double  x);          double sqrt_IV(const double* x);
double sqrt_II(const double x);    double sqrt_V(double* const x);
double sqrt_III(double* x);        double sqrt_VI(const double* const x);
```

(i) Be sure to understand the signatures. What are you allowed to change about `x` inside each function? Define a `double` variable in your `main` and call each function. Do they all work?

(ii) Are there differences between the six versions? What happens exactly underneath when each of them is called? What is copied from the `main` to the scope of the function?

(iii) A `double` variable is usually 8 bytes large. Imagine for a moment of having it 8 *Mega*bytes large. Which version of the square root function would you choose? Why?

From this point on, every parameter passed to a function which is not supposed to be changed inside the function should be marked as `const`. You might wonder if also parameters that are intended to be copied to the function should be marked as constant, although any modification in the function would not be seen from the caller. An always valid and general suggestion is hard to make and arguments might be found both in favour and against it. Just find your way and *be consistent in your code*. However, what is crucial (and everybody agrees about it) is to mark as constant everything that is not supposed to change but that might by accident modified in the function and propagate to the caller. This is basically const-correctness in `C`.