# Einführung in die Programmierung für Physiker

WS 2019/2020 – Marc Wagner

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

## Exercise sheet 8

*To be corrected in tutorials in the week from 09.12 to 13.12.2019*

### Exercise 1 [*Arrays decaying to pointers*]

In this exercise we will explore in detail what exactly happens when an array is passed to a function and how this changes for multi-dimensional arrays. You might think that this is a repetition of something you already learnt: If you already correctly understood everything you have then a good opportunity to test your knowledge.

Let us start from the one-dimensional case and let us focus, without loss of generality, on type `int`. It is also possible to fix the size, let us say 5.

(i) Start defining an array `int v[5]` in the `main` function of your program and initialize it to some values. Define then a pointer `v_ptr` to it. How would you make the pointer point to the array? Find two equivalent but different syntax to achieve that. Print the array to the screen both using the array itself and using the pointer. You might at this point think that the name of an array is implicitly like a pointer to the first element of the array. This statement is slightly inaccurate and there are subtle differences which we will now explore. A more precise statement would be that *array names can decay to pointers*.

(ii) Check that the size of the array is five times the size of an `int`. Use the `sizeof` operator for this purpose.

(iii) Consider the following function.

```
void changeArray(int* array, int size){
    printf("In changeArray, sizeof(array) = %lu\n", sizeof array);
    if(size>2)
        array[2]=111;
    array=NULL;
}
```

Call it from the `main`, using `v_ptr` and printing `v` to the output before and after the function call. Can you explain the discrepancy between the call of `sizeof` in the `main` and in the function? Is `v_ptr` still a valid pointer after the function call? Did `v[2]` change in the `main`? Are you somehow surprised?

(iv) Can you call `changeArray` passing `v` as first argument?

(v) Change the signature of the function to `void changeArray(int array[], int size)` and repeat task (iii) passing `v` to it. Did anything change? Can you call this function passing `v_ptr` as first argument?

(vi) Change the signature of the function to `void changeArray(int array[1], int size)` and repeat task (iii). Does this function work? Can you call it either with `v` or with `v_ptr` as first argument? Why?

Now that you should have understood that arrays cannot naively be passed by value to a function and that only a copy of the pointer to its address is passed to the function, you should be ready to generalize this idea to multi-dimensional arrays. We will just focus on two-dimensional arrays, again with fixed sizes and type `int`.

(vii) Start defining in the `main` function of your program a two-dimensional array `int M[2][3]` and initialize it with integers between 1 and 6. Such an array can be seen as a matrix with 2 rows and 3 columns, but you could think of `M` as an array with two components, which in turn are 3-component one-dimensional arrays. Define then a pointer `M_ptr` to it. This has to be a pointer to a one dimensional array with three integer components. How would you make the pointer point to the matrix? Find two equivalent but different syntaxes to do so. Print the matrix to the screen both using the matrix itself and using the pointer.

(viii) Call the `sizeof` operator on `M`, `M[0]`, `M[0][0]`, `M_ptr` and `*M_ptr`. Can you explain the output?

(ix) Consider the following function.

```
void changeMatrix(int matrix[][3], int rows, int columns){
    printf("In changeMatrix:\n");
    printf("         sizeof(matrix) = %lu\n", sizeof matrix);
    printf("      sizeof(matrix[0]) = %lu\n", sizeof matrix[0]);
    printf("   sizeof(matrix[0][0]) = %lu\n", sizeof matrix[0][0]);
    if(rows>1)
        matrix[1][0]=111;
    matrix=NULL;
}
```

Call it from the `main`, using `M` and printing `M` to the output before and after the function call. Can you explain the discrepancy between the various `sizeof` results in the `main` and in the function? What did the last line in the function body do? Did `M` change in the `main`?

(x) Can you call `changeMatrix` passing `M_ptr` as first argument?

(xi) Consider changing the signature of the function to any of the following.

(a) `void changeMatrix(int (*matrix)[3], int rows, int columns)`

(b) `void changeMatrix(int  *matrix[3], int rows, int columns)`

(c) `void changeMatrix(int  matrix[][], int rows, int columns)`

(d) `void changeMatrix(int matrix[1][3], int rows, int columns)`

(e) `void changeMatrix(int  matrix[][2], int rows, int columns)`

Which is a valid alternative for the function signature in task (ix)? Those that do not work, why are they not working?

(xii) Change now the signature to `void changeMatrix(int** matrix, int rows, int columns)`. Why is it wrong to call this function either with `M` or `&M_ptr` as first argument? Your code might compile and you might want to run it ignoring the warning(s) the compiler gave you... this is never wise!

## Exercise 2 [*The Sieve of Sundaram*]

The sieve of Sundaram – discovered by the Indian mathematician S. P. Sundaram in 1934 – is an algorithm to find all the prime numbers up to a specified integer. Fixed $N \in \mathbb{N}$ and given the list of the integers between 1 and $N$, remove all the numbers of the form $i+j+2\,i\,j$, where $i \in \mathbb{N}$, $j \in \mathbb{N}$, $1 \leq i \leq j$ and $i+j+2\,i\,j \leq N$. All the remaining numbers, doubled and incremented by one, give all the prime numbers smaller than $M \equiv 2N+2$ except 2. Implement the Sieve of Sundaram in a code which gets the value $N$ as first command line option `argv[1]` and either prints all the prime numbers smaller than $M$ or prints how many prime numbers exist up to $M$.

**Time to Test!** For $N = 500$, your code should tell that there are 168 prime numbers up to $M = 1002$.

**Hint:** Even if it could sound natural, do *not* use an array of `int` variables. Instead, think of the array index as your starting list for the sieve and use a `bool` data type to keep track of *deleted* numbers, which are indeed never deleted.