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

Exercise sheet 7

To be corrected in tutorials in the week from 02.12 to 06.12.2019

Exercise 1 [*Pointers and variables*]

Pointers and variables are logically different, but sometimes some confusion arises when dealing with the operators `&` and `*`. Let us try to play a bit with them in order to explore some typical use cases.

(i) Start writing a very basic program. Declare an integer variable `N`, set it to 17 and declare a pointer `Np` to an integer variable making it pointing to the previously defined integer variable.

(ii) Print to the screen the content of `N`, its address, the content of `Np`, its address, and the integer it points to. Which conversion specifier of `printf` do you have to use? Is it clear to you where you have to use `N`, `&N`, `Np`, `&Np`, `*Np` and what they mean?

(iii) Set the content of `N` to 101 and print both `N` and `*Np`.

(iv) Set again the content of `N` to 17, this time using `Np`, and print both `N` and `*Np`.

Now it should be clear to you what a pointer is and how to use it. However, you could still wonder why pointers should be used and how to decide in favour of a pointer instead of a simple variable. There are many reasons, some more involved than others, but probably the simplest one is to allow functions to change variables from the calling scope. Before looking more in detail what this means, let us play another bit with our previous code.

(v) Add the following couple of functions to your code.

```
void SetToTen(int n){ n = 10; }
void SetToTenPtr(int *n){ *n = 10; }
```

(vi) In your main, call each function once with `N` and once with `Np`, printing `N` and `*Np` both before and after each call. Restore `N = 17` before calling the second function. Are you surprised of the result? What is happening?

As you saw yourself, whenever we want a function to change the value of a variable belonging to another scope, we need to use pointers. Functions and pointers can bring some danger with them, though. Consider the following function.

```
#include <stdio.h>
int* GetTen(void){ int n = 10; return &n; }

int main(void){
    int* Np = GetTen();
    printf("Ten from function is %d\n", *Np);
}
```

There is a subtle bug in it. Can you find it? Try to compile and run your code (your compiler could warn you about something going on in the code).

To conclude, write another short program which uses *only* one function to calculate the perimeter and the area of a square, given its side, and somehow *returns* both values. What do you learn from this small task? Can you `return` several values of possibly different type in `C`?

Exercise 2 [*Dealing with arrays*]

It is often needed to collect a set of variables of the same type and treat them as a single object (e.g. passing them all over to a function). This is a rather basic use case of *arrays* and in this exercise you will have a good opportunity to make some practice with them.

(i) Fix a constant size N for your arrays in one of the ways you learnt in the lecture.

(ii) In the `main` of your program declare and initialize two arrays containing `double` numbers.

(iii) Create a generic function which prints a given `double` array. Do not rely on the fixed size of the array, but pass another variable representing it. Use it in the `main` to print your arrays.

(iv) Declare a two-dimensional $N \times N$ array which is meant to contain the Kronecker product of the two existing arrays.

$$\left(\vec{x} \otimes \vec{y}\right)_{ij} = x_i \, y_j$$

How would you delegate this initialization of the two-dimensional array to a function?

(v) Write two more functions, which find the maximum of a given array and the index thereof.

Exercise 3 [*Command line arguments*]

The use of pointers and arrays is required to provide any program a set of so-called *command line arguments*. These are strings which are given on the command line after the name of the executable file when the program is run. For example,

```
./myCprogram --sum -N 100
```

If nothing special is done in the code, command line arguments are simply ignored. However, it is often handy to use these to either pass values to the program or to select a mode in which the code should be run. If you think about the commands you use in the terminal, this is always the case. Since a `C` program, once compiled, is analogous to any terminal external command, it makes sense to think of command line options in the same way.

Write a program which accept the following command line options and acts accordingly. If any different option is given, the program should print an error and abort.

`--help` The program should print a list of avaliable options with a short description and exit.

`-N` The program saves the following command line option in an integer variable. The value 0 is not accepted. How can you store a `char*` variable into an `int` one?

`--sum` The *sum* mode is selected. The program calculates the sum of all integer numbers between sign(N) and N, where N is the value specified by the `-N` option.

`--product` The *product* mode is selected. The program calculates the product of all integer numbers between sign(N) and N, where N is the value specified by the `-N` option.

If you want to make your program even handier to be used, think about implementing the following features.

(i) It should be possible to give the `--help` option at any place and even together with wrong options. In this case the program should simply print the help message and exit.

(ii) How would you add short version for every options? It would be nice to be able to specify `-h` alternatively to `--help`, `-s` and `-p` instead of `--sum` and `--product`, respectively.

(iii) Either `--sum` or `--product` should be specified. If both are specified, you can decide to give an error or, since in this case there is no conflict, to let the program perform both actions.