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

# Exercise sheet 5

*To be corrected in tutorials in the week from 18.11 to 22.11.2019*

## Exercise 1 [*C standards and the* `bool` *type*]

**Take-home message:** In `C` you can use the `bool` data type using a modern compiler and including the `stdbool.h` library. If you compile your code with `g++`, no library is necessary since in `C++` the `bool` data type has been existing from the beginning. Your code will be compiled as `C++` code, though.

**The long story:** In computer science, the Boolean data type is a data type that has one of two possible values (usually denoted true and false) which is intended to represent the two truth values of logic and Boolean algebra. Although you might expect a `bool` data type to have always existed in `C`, this was only introduced in 1999 and it does not belong to the original standard of the language[1]. The `C` standards are `C89` (sometimes also called `C90`), `C99`, `C11` and `C18`. It is not important to know the details behind every standard, but a couple of remarks should be kept in mind.

- The `C` programming language is *alive* and it evolves getting new features, which often make it more powerful.
- It should be possible to tell the compiler to stick to the rules of a given standard. A default behaviour is fixed if nothing is requested, but this default behaviour depends on the version of the compiler!
- The new Boolean data type introduced in the `C99` standard is called `_Bool`. Only including the `stdbool.h` library it is possible to use it simply as `bool` and the new keywords `true` and `false` become accessible.

*Why should all this be relevant?* – you might be wondering. In short, because it is extremely handy to use a `bool` variable whenever some logic is needed and to write `true` instead of 1 and `false` instead of 0 makes any code a much clearer and readable.

(i) Which `C` standard uses your compiler by default?

(ii) What does the `-pedantic` option of your compiler do?

(iii) Play around with the following code.

```c
#include <stdio.h>

int main(void){
  _Bool a=1;
  _Bool b=0;
  /* bool a=true;  */
  /* bool b=false; */
  printf("a=%s\n", a ? "true" : "false");
  printf("b=%s\n", b ? "true" : "false");
}
```

Does it compile? Try to compile it via `gcc -std=c90 -pedantic`. Do you understand what the compiler tells you? Replace the variable definitions with the out-commented ones. What do you need to compile the modified code?

[1] `C` has been standardised by the ANSI since 1989. This means that there are a set of rules and features which belong to the '89 `C` standard. You can roughly thing to a standard as a *version* of the language.

## Exercise 2 [*Sum and Products*]

Write a `C` program which evaluates the following two expressions. Tackle one task with a `for` loop and the other with a `while` loop.

$$\text{(i)} \quad \sum_{k=1}^{N} k^2 \qquad \text{(ii)} \quad N!! \equiv \prod_{k=0}^{\lceil \frac{N}{2} \rceil - 1} (N-2k) = N\,(N-2)\,(N-4)\cdots$$

In task (ii), $\lceil x \rceil$ denotes the least integer greater than or equal to $x$. Fix $N$ in an appropriate way and test your code. Which data type are you going to use for $k$ and which for storing the result?

How would you tackle the following, more challenging calculation? Which can be a good stopping condition for your loop? Can you guess the result of the sum? Use a `do-while` loop.

$$\text{(iii)} \quad \sum_{k=1}^{\infty} \frac{(2k)!!}{(2k+1)!!}\, 2^{1-k}$$

**Hint:** Don't evaluate the two double factorials alone before taking their ratio, but think of a better way.

## Exercise 3 [*Prime numbers*]

By definition, a prime number is a positive natural number which has *exactly* two divisors, 1 and itself. Write a first program which

(i) asks a number $N$ to the user;

(ii) checks if it is a prime number;

(iii) prints its prime decomposition if it is not a prime number;

(iv) counts the number of prime numbers up to the given number.

**Optional:** In the prime decomposition, group the same factors together. The program should print something like `1000 = 2^3 * 5^3`.

**Structure your code:** Although you did not discuss in detail the functions syntax, it should not be difficult to structure your code writing simple functions. Refer to the second lecture notes in case of doubts. Implement the check of task (ii) in a `bool IsPrime(unsigned int inputNumber)` function and write a function which, given a number, prints its prime decomposition to the standard output. A possible signature for the latter could be `void PrintPrimeDecomposition(unsigned int inputNumber)`.

**Time to Test!** What happens if you give 1 or a negative number to your code? Run your code with several integers, like `17`, `40309`, `65536` and `100003`. For $N = 1000$, your code should tell that there are 168 prime numbers up to $N$.

(v) Remove any `scanf` function and fix the number to $N = 10^7$.

(vi) Run your code and measure the execution time. Do it from your shell using the `time` command, which has to be placed before your executable (e.g. `time ./myEx`). The `time` command gives you more time measurements, consider only the *real* one.

(vii) Try to recompile your code with different optimization levels[2] (e.g. `-O1`, `-O2`, `-O3`) and measure again the execution time. This is the first time you can experience what the compiler can do for you! **Attention:** Never use optimization flags like `-Ofast` which imply `-ffast-math`, because they will probably make your *floating point* result inaccurate and then wrong!

[2] Here we refer to the names of the `gcc` compiler options. If you use a different compiler you have to check out which are the equivalent names.