Tutors: C. Czaban, A. Dromard, J. Glesaaen, M. Neuman, D. Palao, C. Schäfer, A. Sciarra

Tutorial III

November 7

Excercise 1 [*Babylonian roots*] The purpose of this exercise is to write a program that computes the square root of a positive number using the so-called *Babylonian method* (also known as *Heron's method*). The algorithm for computing $\sqrt{S}$ can be skecthed as follows[1]:

(i) Start with a positive initial value (the closest to the final result, the better), let us call it x_0;

(ii) Take as n-th estimator of the result the following: $x_n = \frac{1}{2}\left(x_{n-1} + \frac{S}{x_{n-1}}\right)$;

(iii) Repeat step (ii) until the desired precision has been achieved.

Make sure that your error is smaller than 10^{-6} by smply comparing two consecutive iterations.
Testing is a very healthy habit when programming. As a rule of thumb, *never* consider that a program is finished if it was not tested. Never. Therefore, you should do a couple of tests in order to finish the exercise. As a first test you should consider the obvious procedure of squaring your result to check the validity of the computation. You should also apply your program to a couple of particular cases, like $\sqrt{9}$ (starting with $x_0 = 2$), $\sqrt{143}$ and $\sqrt{2}$, and/or some other values, and compare with the actual values.
A further simple test for your code is making another program that uses the standard `sqrt` (or `sqrtf`) function, available once the `math.h` header is included, and verify that your own program works properly.
As usual in programming, there are many ways to complete this task, but we recommend to practice with the usage of loops and `if/else` clauses along this exercise. We also recommend to use functions to improve the readability, reusability and mantainability of your code.

Excercise 2 [*Binomial coefficients*] Write a `C` program that computes the binomial coefficients. Please, use functions to improve the quality of your code. Use also *loops* (either `for` or `while` loops) and `if` statements.
Note: Make use of the following expression

$$\binom{a}{b} = \frac{a!}{b!\,(a-b)!}\,,$$

but check that it can be applied to your input. Also think of ways to test your program, and test it.

Excercise 3 [*Machine precision*] In this exercise we will compare the machine precision of the variable types `float`, `double` and `long double`. First, you should note that precision does not refer to the absolute smallness of numbers one can store in the respective variable types, but rather the relative size difference between the numbers one can store.

(i) To get an idea of what we are dealing with, start by storing 1.0 in a `float` and $1.0 + 10^{-10}$ in a different `float`. Compare them using the different comparison operators (`<`, `>`, `==`, ...) and see whether the computer can tell the difference.

(ii) Redo the previous step with `double` instead of `float`.

With this in mind, write a program which finds the smallest number you can add to 1.0 so that the computer sill recognises that they are different. Do this for variable types `float`, `double` and `long double`. Compare your results with the corresponding constants defined in `float.h`. Remember that even floating-point numbers are stored as bits on a computer. You should therefore not check with increments of powers of 10, as would be natural to us, but rather powers of 2. Lastly, skim through the Wikipedia articles for `float`[2] and `double`[3] to improve your understanding of how floating-point numbers are stored on a computer. Does your program give you the expected result?

[1] See, e.g. the Wikipedia for more details.
[2] `http://en.wikipedia.org/wiki/Single-precision_floating-point_format`
[3] `http://en.wikipedia.org/wiki/Double-precision_floating-point_format`

Excercise 4 [*Precedence and associativity*] In C, as in many other computer languages, there are some rules for precedence and associativity of operators. These rules can vary from language to language, and they might not be totally intuitive at first, so it is perhaps a good idea to practice a little bit with them. That is precisely the goal of the present exercise. Let us assume that the following statements are executed before each of the cases we are going to analyze:

```
double a, b, c, Y;
int i, j, X;
a = 1.0;
b = -3.0;
c = 5.5;
i = 5;
j = -6;
k = 10;
```

First evaluate each of the following expressions "analytically" (using pencil and paper at most) and also tell what is the value stored in the variable `X`, when relevant. In a second step check your results "numerically" by writing a corresponding C program.

(i) `a + b*c`

(ii) `a + (b*c)`

(iii) `(a + b)*c`

(iv) `i/j`

(v) `(double) i/j`

(vi) `((double) i)/j`

(vii) `(double) (i/j)`

(viii) `X = i >= 0 && i <= 10`

(ix) `X = i >= 0 && i <= 10 || i == 23`

(x) `X = i >= j && i <= k`

(xi) `X = i <= j && i >= k`

Consider now the logical operators `&&` and `||` and the ***post***-increment operators `++` and `--`. Given the following code,

```
int main(){
  int i=-1, j=0, k=3, l=1, m;
  <conditional_line>
  printf("i=%d\tj=%d\tk=%d\tl=%d\tm=%d\n",i,j,k,l,m);
}
```

try to figure out which is the output in the following cases without using the computer. Then check it running the code.

(xii) `m = j++ || k--;`

(xiii) `m = i-- && l++;`

(xiv) `m = i++ || l++;`

(xv) `m = j-- && k--;`

To tackle the last two expressions, you should recall *the shortcut nature of the logical operators*:

`&&` The logical-AND operator produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.

`||` The logical-OR operator performs an inclusive-OR operation on its operands. The result is 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

Now you should be able to forsee the output of the above code also when the `<conditional_line>` is substituted by:

(xvi) `m = k-- && i-- && j++ && l--;`

(xvii) `m = k++ && j++ || i-- || l--;`

(xviii) `m = j++ && k-- || i++ || l--;`

(xix) `m = i++ || j++ && k-- || l--;`

Which is the lesson of the last two expressions?