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

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

Exercise sheet 3
To be corrected in tutorials in the week from 04.11 to 08.11.2019

Exercise 1 [*Knowing your machine*]
There are two important libraries, which are not very often used, but which it is important to know:

- `limits.h`
- `float.h`

Online there are many good reference websites. Most of them are reference for `C` and `C++` at the same time. A nice, not extremely technical one is `http://www.cplusplus.com/reference/clibrary/`.

(i) Have a look to the content of the above mentioned header files.

(ii) Write a short computer program which prints the minimum and maximum numbers which can be stored inside variables of the following types on your machine.

- `short int`
- `int`
- `long int`
- `short unsigned int`
- `unsigned int`
- `long unsigned int`
- `float`
- `double`
- `long double`

You do not find everything in the above library, because some limits are…trivial!

(iii) Based on the output of your program, can you deduce how many bits are used for `int` variables on your machine? And for `unsigned int`?

Exercise 2 [*4 ways to swap variables*]
To *swap* means to *exchange* and, for our purposes, it is just about

- having two variables initialized with own values;
- assigning to each of the two variables the value originally assigned to the other.

You can work with either `int` or `float` numbers and start with two variables, which we may call `a` and `b`. Your task is to write a program to achieve the swap in 4 different possible ways:

(i) The most naive way is

1. to make use of a third auxiliary variable `c`;

(ii) Somehow less naive ways, that is by **ONLY** using `a` and `b` and

2. Implementing a chain of addition/subtraction (and, of course, assignment) operations;
3. Implementing a chain of multiplication/division (and, of course, assignment) operations;
4. Implementing a so called "`XOR` swap algorithm" that relies on boolean algebra.

Time to Test! Can you identify any pitfall in the various implementations? Said in other words, can you make your swapping algorithm fail depending on the values of the variables to swap and find a reason for the failure?

Exercise 3 [*Operators precedence and casts*]

In `C`, as in most of programming languages, there are some rules for precedence and associativity of operators. Knowing them in detail helps to avoid bugs that may be not so easy to be found, especially in more complex codes. Set up a small program and put the following statements at the beginning of the `main`.

```
double a, b, c, X;        int i, j, k, N;
a=1.0; b=-3.0; c=5.0;     i=5; j=-6; k=10;
```

For each of the assignments here below, try to understand which value is stored in `N` or in `X` using pencil and paper at most. Only when you have an idea of the outcome together with a convincing motivation, add a couple of lines of codes to your program to check if you were right.

(i) `X = a+b*c;`

(ii) `X = a+(b*c);`

(iii) `X = (a+b)*c;`

(iv) `X = a/b*c;`

(v) `X = a/(b*c);`

(vi) `X = i/j;`

(vii) `X = (double)i/j;`

(viii) `X = i/(double)j;`

(ix) `X = (double)(i/j);`

(x) `N = (a>b);`

(xi) `N = a>b;`

(xii) `N = i>k+j;`

(xiii) `N = (i>k)+j;`

(xiv) `N = i>=j && j<=k;`

(xv) `N = k/i && i/k;`

(xvi) `N = i+j+1 || k/(2*i);`

(xvii) `N = a/b;`

(xviii) `N = X = a/c;`

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

```
#include <stdio.h>
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.

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

Then check it by replacing `<conditional_line>` with each of the corresponding assignment statements and by running your code. To tackle the last two expressions, you should recall *the short-cut nature of the logical operators.*

`&&` The logical `AND` operator produces the value 1 if both operands have non-zero 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 non-zero value, the result is 1. If the first operand of a logical `OR` operation has a non-zero value, the second operand is *not* evaluated.

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

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

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

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

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

Which is the lesson of the last two expressions?