Einführung in die Programmierung für Physiker

WS 2017/2018 – Marc Wagner

Francesca Cuteri: cuteri@th.physik.uni-frankfurt.de
Alessandro Sciarra: sciarra@th.physik.uni-frankfurt.de

Exercise sheet 4

To be corrected in tutorials in the week from 13/11/2017 to 17/11/2017

Exercise 1 [*Triangles taxonomy*]

Write a program which takes as input 3 floating point numbers, stores them in variables of type `double`, and checks whether they could correctly represent the length of the three sides of a triangle based on the constrains that

(i) all given values are positive numbers;

(ii) the length of every side is less than the sum of the lengths of the other two sides;

(iii) the length of every side is more than the difference of the length of the other two sides.

What kind of triangle can be drawn with the sides read in input? Print to the user

(i) whether it is an equilateral triangle;

(ii) whether it is an isosceles triangle;

(iii) whether it is a scalene triangle;

(iv) and whether, being it isosceles or scalene, it is a right-angled triangle.

Time to Test! This exercise, despite its harmless appearing, hides some profound aspect of computer programming. Checking if two floating point numbers are equal is a very complicated business and, strictly speaking, there is not a correct way of doing that. Whaat? We have the == operator, isn't it?! Well, yes and no. Let us test together your code to understand why it is complicated to compare non-integer numbers. If we choose easy cases like 3, 4 and 5 as input values, we should get what we expect. Run your code and check it. But what happens in a trickier case, like with 3/7, 4/7 and 5/7? Try to give `0.4285714285714143`, `0.5714285714285714285` and `0.71428571428571428571` as input and check if you obtain a right triangle as output. Are you able to explain what is going on?
If everything is as you expected, then you should be able to explain the output of the following code

```
#include<stdio.h>
int main(){
    float x = 0.1f, sum = 0, product = x*10;

    for(int i=0; i<10; ++i) sum += x;

    if (sum == product)
        printf("10*0.1f is equal to 0.1f+...+0.1f ten times\n");
    else
        printf("10*0.1f is NOT equal to 0.1f+...+0.1f ten times\n");

    printf("sum-product = %1.15f\n", sum-product);
}
```

which gives the following output on on most of systems.

```
10*0.1f is NOT equal to 0.1f+...+0.1f ten times
sum-product = 0.000000119209290
```

Exercise 2 [*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?