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 5

To be corrected in tutorials in the week from 20/11/2017 to 24/11/2015

Exercise 1 [*Leaplings*]

In the Gregorian calendar, February in a leap year has 29 days instead of the usual 28. A person born on February 29 may be called a *leapling*. Technically, leaplings will have fewer birthday anniversaries than their age in years.
In Gilbert and Sullivan's 1879 comic opera "*The Pirates of Penzance*", Frederic the leapling pirate apprentice, born in 1856, discovers, at the age of 21, that he is bound to serve the pirates until his 21st birthday rather than for 21 years, meaning he has to serve for much much longer. We want to help Frederic finding out in which calendar year will he be allowed to retire according to his indenture, and how old will he really be at his retirement.
To accomplish the task you are requested to write a program in `C`, which you can run to:

(i) Establish whether any year provided as input by the user is or is *not* a leap year (input in this case is just a year);

(ii) Count leap years within any range of years provided as input by the user (input are two years);

(iii) Establish the year at which a leapling, born in some user provided year, will have celebrated the user provided number of birthdays and what will be his/her real age in years by then (input are a year and a number of birthdays).

Hint 1: you can first ask the user in which mode he/she wants to run the code and use some

```
if (/*<condition to enter mode 1>*/) {
    /* ... */
} else if (/*<condition to enter mode 2>*/) {
    /* ... */
} else if (/*<condition to enter mode 3>*/) {
    /* ... */
} else {
    /* ... */
}
```

construct to enter various modes, e.g. reading a `char`.

Hint 2: If you would write a function `bool IsYearLeapYear(unsigned int year)` not only you could use it in the solution of (i), but you would reuse it in the solution of (ii). Note that, strictly speaking, the `bool` data type did not belong[1] to the `C` language and it was introduced in `C++`. However, it is very handy to use it and we encourage you to do so (remember to use, then, e.g., `g++` as compiler). A `bool` variable can either contain the logical value `true` or `false` and, then, statements like `bool isLeap=false;` or `return true;` are perfectly allowed. Moreover, you can test the content of a `bool` variable `x` with an `if`-clause as `if(x==true)` but also simply as `if(x)`.
Also a function `unsigned int CountLeapYearsInRange(unsigned int year1, unsigned int year2)` might turn out to be useful more than once!

Time to Test! Note that you can test your solution for item (iii) against your solution for (ii)!

[1] The `C99` standard of `C` introduced the `stdbool.h` header to allow the user to use `bool` as in `C++`.

Exercise 2 [*Bug hunting*]

In a previous exercise sheet you were asked to write a program to simulate darts game played blindly by your computer. Here you find a somehow **broken** solution for that task, because your task, this time, is to *debug* a code! Pay attention because you may encounter different kinds of errors:

(i) **Syntax errors** that you will be able to detect at compile-time;

(ii) **Logic errors** that you *might* only be able to detected at run-time.

There are *at least*[2] 8 bugs in the following code.

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main(){

    unsigned short int nPoints;
    unsigned short int nTrials;

    printf("Enter the total number of darts you want
            me to throw (<%hu):\n", USHRT_MAX / 4);
    scanf("%hu", nTrials);

    if (nTrials == 0) {
        printf ("Throwing 0 darts makes no fun!\n");
        exit(0);
    }
    printf("Ok user, I'll throw %hu darts for you...\n", nTrials);

    for (short int i = 0; i <= nTrials; i++) {
        double x = (double)rand() / RAND_MAX * 10 - 5;
        double y = (double)rand() / RAND_MAX * 10 - 5;
        double d = sqrt(x * x + y * y);

        if (d <= 5); {
            if (nPoints > USHRT_MAX - 4) {
                printf("Adding up 4 points would bring you to %hu
                        points, which means...\n", nPoints + 4);
                printf("Sorry, I cannot count your score any more:
                        You run into overflow!\n");
                exit(0);
            }
            else
                nPoints =+ 4;
        }
    }

    printf("%1.8f\n", nPoints / nTrials);
    return 0;
}
```

Rather than rewriting the code or copying it from the pdf, which are both not so good ideas, you can simply download the above code from the web by e.g. running the command
`wget https://th.physik.uni-frankfurt.de/~mwagner/teaching/C_WS17/c_II_bugHuntingDarts.c`
in your Linux terminal. Keep hunting *not* just until the code compiles, but until it does what expected!

[2] You can always introduce new errors while debugging.

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 to the user;

(ii) checks if it is a prime number;

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

Implement the check 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`.

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

Change now the goal of your code, such that it counts the number of prime numbers up to a given number N. Use a pre-processor directive to make N known at compile time.

Time to Test! For $N = 1000$, your code should tell that there are 168 prime numbers up to N.

Once your code is ready for production, tackle the following tasks.

(i) Run your code with $N \in \{5 \cdot 10^4, 10^5, 5 \cdot 10^5, 10^6, 5 \cdot 10^6, 10^7\}$ 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.

(ii) Set up a data file with two columns, the first containing the values of N you used and the second containing the execution time in ms.

(iii) Use `gnuplot` to plot your data, using a logarithmic scale on the x- and y-axis and connecting your points with segments. To make the plot look nicer, you can use a command like

```
plot "data.dat" u 1:2 pt 6 with linespoints title "times"
```

which plots both the points and a line connecting them.