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

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

Exercise sheet 10

To be corrected in tutorials in the week from 13.01 to 17.01.2020

Exercise 1 [*Unit tests*]

This exercise is a bit challenging but it is though as a valuable opportunity for you to improve your programmer skills. Take it as Christmas gift ☺ and try to work it out, discussing then its solution with your tutor.

In computer programming, *unit testing* is a way to ensure that the code is working by focusing on single units of source code. In real life, this is nothing but the first step to test a code[1], but for our purposes it is already almost everything we need.

In this exercise we will develop a scheme to make unit tests that can be reused in other codes. The idea is to have a set of functions (our *units*) and to test them separately. In order to do so, we will write a new function for each unit to be tested and, inside it, the original function will be called with default parameters and its output will be compared with some reference values. At the beginning all the input and reference values will be hard-coded, i.e. our test functions will not get any parameter in input. Later on we will try to generalize our code, but only in some particular case.

(i) Your first task is to write some typical testing function, i.e. functions which encapsulate a comparison between numbers.

```
bool isLarger(const double lhs, const double rhs);
bool isSmaller(const double lhs, const double rhs);
bool isEqual(const double lhs, const double rhs, const double epsilon);
```

(ii) Now let us consider some units to be tested. To stay focused on the idea of testing, we will implement some simple mathematical functions, using what is available in the `math.h` library. Define the following functions.

```
double gaussian(const double x, const double mu, const double sigma);
double sine(const double x, const double A, const double B);  // sin(A*x+B)
double exponential(const double x, const double alpha);       // exp(alpha*x)
```

(iii) For each of the functions above, implement a function to test them. Their structure should be similar.

```
    bool /*<function name>*/_test(void){
        // 1 - define input and reference values
        // 2 - call mathematical function
        // 3 - make test and return
    }
```

Check, for example, that the Gaussian distribution $N(x,0,1) > 0$ at some value of x, $e^{3x} < 1$ for some negative value of x and $\sin(4x + \frac{\pi}{4}) = -\frac{\sqrt{2}}{2}$ at $x = \frac{\pi}{4}$.

(iv) Now we should focus on implementing a `runAllTests` function which runs all the tests and gives information to the user. In this function you should, then, call all the `_test` functions and keep track of successes and/or failures. It should now be clear, why we decided before to make a single test return a `bool` variable. We would like to have a complete report. Delegate the printing of a single test outcome to a separate function.

```
void printOutcomeSingleTest(const bool passed, const char* testName, /*...*/);
```

[1] Usually, depending on the software, *integration tests*, *component interface tests*, *system tests* or *operational acceptance tests* are needed before a release. Refer to `https://en.wikipedia.org/wiki/Software_testing` to read more on the topic.

```
 --------------------------------------
  Running 3 tests...
    1 Gaussian distribution    OK
    2 Sine function            FAILED
    3 Exponential function     OK
  ...done!
 --------------------------------------
   2 test(s) passed, 1 test(s) failed!
 --------------------------------------
```

(v) So far so good. Now, the interesting part comes in. Suppose to have a new unit to be tested, and to have already implemented its `_test`. We would like to minimize the effort to add this test to our report. At the moment, to add it, you should probably implement an `if` case, and some printing. It would be nicer to just call a function `runSingleTest`, which takes over the job. Indeed, since the signature of all the `_test` functions is the same, it is not so difficult. Try to implement it, passing a function-pointer as parameter (and a `const char*` to print the test name). Now, your `runAllTests` function can be rewritten in a much easier to generalise way, using e.g. calls like `passed+=runSingleTest(sine_test, "Sine function");`.

(vi) Despite the fact the code looks already good at this point, we cannot be completely satisfied. Each single `_test` function has the test input and reference value(s) hard-coded and this makes the code hard to maintain and generalize. Moreover, our `_test` functions need to prepare input and reference values for the test as well as they have to make the test themselves. This means we violate the single-responsibility rule, according to which each function should not have more than one responsibility[2]. We would like, then, to delegate the testing responsibility to a new function, which we will try to implement in a way such that we can use it in all our `_test` functions. A possible signature could be

```
bool makeTest(/*function-ptr*/, const double x, const double inputParams[],
              const double refValue, const char testType);
```

where the first argument is the function to be tested, the array contains the input parameters to be passed over to the function and the `const char` is used to decide whether to call `isLarger` (`'l'`) or `isSmaller` (`'s'`) or `isEqual` (`'e'`). Add this function to your code.

(vii) In order to use the `makeTest` function in our `_test` functions, we need to slightly change our mathematical functions. In task (ii), we used signatures with `const double x` as first argument and a set of `const double` parameters. We would like, instead, to have only two arguments, the function independent variable `const double x` and an array of parameters `const double params[]`. Rewrite your code accordingly.

(viii) Now you are ready to rewrite your `_test` functions which should set up the test parameters (i.e. prepare the array of parameters for the mathematical function) and call the `makeTest` function. For example, to test that $e^{3x} < 1$ for $x = -\pi$, we could write

```
bool exponential_test(void){
    const double x = - M_PI,  refValue = 1;
    const double params[1] = {3};  const char type = 's';
    return makeTest(exponential, x, params, refValue, type);
}
```

which is very readable and maintainable.

As you may have already noticed, we would get in trouble if we had to add a mathematical function with integers parameters,

```
double binomialDistribution(const double p, const unsigned int n, const unsigned int k);
```

or, even more, if we had to add a mathematical function with parameters of different types. You will learn soon how to approach this problem and how to extend the implemented technology.

[2]This rule comes from common *clean code* habits that programmers daily strive to apply. If you want to know more about the topic, you can refer e.g. to the ≪*Clean Code*≫ book by R.C. Martin.