Otto-von-Guericke-University Magdeburg
Max Planck Institute for Dynamics of Complex Technical Systems
Computational Methods in Systems and Control Theory

Dr. Jens Saak, Dipl.-Math. Martin Köhler
Website: http://www.mpi-magdeburg.mpg.de/csc/teaching/20ws/sc1/

Winterterm 2020/2021

Scientific Computing 1
2nd worksheet for online events
11/19/2020

*"When reading the code in about six months and asking yourself: who wrote this crap?
The answer should not be: YOU!"*

Basically that means:

- Try to always use meaningful names for functions, variables, ...
- Write documentation wherever necessary.
- Use indentation to increase readability of the code.
- Add a short statement describing its purpose and basic behavior to each function.
- ...

Exercise 1:

Write a C program which reads two integers a and b and computes $\frac{a}{b}$.

a.) Find out what happens if b is zero.

b.) Does the compiler recognize if there is a hard-coded division by zero?

c.) Rewrite the program to floating point numbers. What happens now if b is equal to zero?

d.) Modify the program such that $b = 0$ is detected and avoided before an error occurs.

e.) Is the modulo-operator also affected?

Exercise 2:

Write a C function which converts a temperature given in degrees Fahrenheit to degrees Celsius. The conversion is done with

$$T_C = (T_F - 32) \cdot \frac{5}{9}.$$

Demonstrate the function with two examples:

a.) Read a temperature in degrees Fahrenheit from the standard input and print out the corresponding degrees Celsius.

b.) Read a lower and a upper bound from the standard input defining an interval in degrees Fahrenheit. Print a table containing the temperatures in degrees Fahrenheit and degrees Celsius to the screen. In the table use steps of 1 Fahrenheit.

Exercise 3:

Examine the following program and try to find all memory related errors using `valgrind`. Describe the problems and their solutions. The code is also available at http://www2.mpi-magdeburg.mpg.de/mpcsc/lehre/2020_WS_SC/demo_valgrind.c

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

int main(int argc, char **argv)
{
        int vars[2];
        double sum;
        int i;
        double data[10] = {1,22,23,344,5,34,75,36,89,540};
        double *data2;

        data2 = malloc (sizeof(double) * 10);

        for (i = 0; i <= 10; i++) {
                data2[i]=data[i];
        }

        sum = 0.0;
        for (i = 1; i <= 10; i++) {
                if ( i == 1 ) {
                        vars[i] = data2[i];
                }
                if ( i == 2 ) {
                        vars[i] = data2[i];
                }
                sum = sum + data2[i];
        }


        printf("Sum of all elements: %lg\n", sum);
        printf("var1: %d var2: %d\n", vars[0], vars[1] );

        return 0;
}
```

Hints:

- Switch the generation of debugging symbols on when you compile the program.
- Check the output of `valgrind` to get additional command line options for a deeper analysis of the problems.

Exercise 4:

`Makefiles` support the developer to build large projects easily. Write a `Makefile` which has a target for each source code of the exercises above. For the `valgrind` problem create an additional target for building with debug symbols.
Calling `make all` all programs should be compiled and calling `make clean` should clean up all binary files created by the make process.
Revisit your C files and augment them for the use of `doxygen`. Then add another target `doc` that generates the documentation.