High-Performance Computing
Prof. Dr. Ivan Kisel
Robin Lakos, Akhil Mithran, Oddharak Tyagi

High-Performance Computer Architectures
Practical Course
SoSe 2025

Issued: 30.04.2025 / Due: 07.05.2025

GOETHE UNIVERSITÄT FRANKFURT AM MAIN
Institute for Computer Science

Exercise Sheet 1

Introduction to C++ Programming

The goal of this exercise is that every group member learns how to work with Unix-shell and C++. Please remember, every group member should be familiar with all exercises and be able to answer questions about the source code and your solutions.

Please solve the following exercises as a group and upload your solutions compressed in a ZIP-file to OLAT (Please only one submission per group). Your solutions should include all required source code files with meaningful comments and a PDF file. In the PDF, unless requested otherwise, answer all questions and always explain the problem, your solution as well as your results. Additionally, provide screenshots of your results after running your application and add your compilation command including all compiler flags. Do not forget to add the names and enrollment numbers of your group members.

For this exercise sheet, it is sufficient to work with source files (e.g. `.cpp`) only.

Please find the additional material for examples at the OLAT course website.

Exercise 1.1: Hello World? Hello World!

Create a C++ source code file `HelloWorld.cpp` with a simple console output of `"Hello world!"`. Compile and run your source code.

Submitting your source code is sufficient.

Exercise 1.2: Conditional Statements and Loops

Create a C++ source code file `ConditionalStatements.cpp` with a console application that expects a user input n (integer type). The input n should be checked for the following conditions and limited to these boundaries otherwise: $0 \leq n \leq 9$. Afterwards, print ”Hello world!” to console n times.

Submitting your source code is sufficient.

Exercise 1.3: Troubleshooting

i) Factorials

Open the C++ source code `Factorial.cpp` and think about the expected output.

Will the code compile successfully? If yes, will the application run successfully and print the expected output? If not, explain the problem, fix it and explain your solution.

ii) Pointers and Functions

Open the C++ source code `PointersAndFunctions.cpp` and think about the expected output.

Will the code compile successfully? If yes, will the application run successfully and print the expected output? If not, describe the problem and provide a possible solution and explanation.

iii) Function Arguments

Open the C++ source code `FunctionArguments.cpp`.

First, complete the source code. Then, for the cases 1-4 (see comments), explain why the output is as it is.

Compile and run the code and compare the output to your expectations.

iv) Templates

Compile and run the source code `Templates.cpp` and compare the results to the code. For each data type, explain the results in detail.

Hint: For some results, it might help to cast the results into another data type and have a look at an ASCII table.

Exercise 1.4: I Also Like to Live Dangerously

Open the C++ source code `Arrays.cpp` and think about the expected output.

Compile the source code and run the application. Compare the results to your expected output. What happened?

Comment lines 15 and 17, then compile and run again. Is something different? Explain why. Is it safe to use?

Exercise 1.5: Fun with Pointers

Open the C++ source code `PointerWalk.cpp`. The small array initialized in the beginning is printed twice. Explain the differences between the two printing approaches. In both, the memory addresses are printed together with the array elements.

i) Memory Locations

Explain your output and explain the the memory addresses. More precisely, note the memory address of the first element in the array and analyze its value. Consider why it is located at that particular address. Similarly, investigate the memory addresses of the other elements in the array. Are there any patterns or differences between the addresses of adjacent elements? Finally, examine the memory address of the pointer used in the code.

ii) Size of Arrays

The `size` is calculated by using the built-in `sizeof` function. Explain how the size is calculated. Also explain why this approach does not work for dynamically allocated arrays or arrays that decayed into a pointer; what is calculated in these cases when the same approach would be used?

Exercise 1.6: The Power of References

The `PowerOfReferences.cpp` contains source code that compiles into an application with a quite long run-time.

Please reduce the run-time of the application significantly by only adding/deleting/changing only a single character/symbol in the source code and compare the results. Do not change the number of iterations or the size of the data structure. Additionally, explain why the modified code runs faster.

Why is this improvement not always an option? What concept do you know that can provide a similar speedup here?

Exercise 1.7: Debugger

Debuggers are invaluable tools for developers. On many Linux-based systems, the GNU Debugger (GDB) is already pre-installed, and almost every larger IDE incorporates a debugger tailored to its supported programming languages.

Begin by researching the compilation flags necessary for debugging with a C++ compiler of your choice (e.g., clang++, g++). Additionally, gather information on how to use GDB effectively.

i) Write a How-to Guide

Write a brief "how-to" in bullet points on how to compile your code with debugging flags and how to run applications using GDB. Include at least:

- One compilation flag used for enabling debugging features
- Basic steps to initiate GDB and load a program
- Instructions on how to set and describe the utility of breakpoints

ii) Run the Debugger

Open `Arrays.cpp` again and make sure lines 15 and 17 are commented. Compile `Arrays.cpp` with the debug flag `-g`, run the executable using GDB and provide a screenshot of your results. Embed the screenshot in your PDF and explain the results. More precisely:

- What did you expect to happen when you ran the debugger?
- How did the actual results compare with your expectations?
- What kind of behavior is observed, and what might be causing any discrepancies or unexpected results?