Basic C++ Course

M. Ohlerich
Leibniz Rechenzentrum der Bayerischen Akademie der Wissenschaften

February 28, 2024
## Contents

1 License Terms and Conditions ................................................. 11
   1.1 License ........................................................................ 11
   1.2 Disclaimer ................................................................... 11

1 Introduction to Software Engineering / Basic Syntax Elements of C++ 13

2 Introduction ........................................................................ 15

3 Software Development Projects .............................................. 17
   3.1 A first Project – the first C++ Program ............................. 17
       3.1.1 The first C++ Program ............................................. 17
       3.1.2 Basics of CMake – the Build-Tool ............................ 18
   3.2 Some Words about Editors .............................................. 21
   3.3 Git – a Versioning Tool .................................................. 21
   3.4 Test-Driven Development .............................................. 24
   3.5 Documenting Programs using Doxygen ............................ 26

4 Basic Elements of C++ ......................................................... 29
   4.1 Expressions and Statements .......................................... 29
   4.2 Variables ..................................................................... 30
       4.2.1 Types .................................................................. 30
       4.2.2 Identifier Names .................................................. 30
       4.2.3 Scope and Namespaces ......................................... 30
       4.2.4 Arrays .................................................................. 32
   4.3 Control Structures ....................................................... 33
       4.3.1 Conditional Branches ............................................ 33
       4.3.2 Loops .................................................................. 34
   4.4 Functions ................................................................. 36
       4.4.1 Function Signature – Declaration and Definition .......... 36
       4.4.2 Function Overloading, Name Resolution, and Implicit Casting 37
       4.4.3 The main Function ............................................... 38
       4.4.4 Function Parameters – Call-by-Reference vs. Call-by-Value; References and Pointers 39
       4.4.5 Lambda-Functions ............................................... 41

5 Adding Subfolders and Files to the Project ................................. 43

6 Introduction to Vector, String, Algorithms, and I/O – STL Basics 47
   6.1 Basic I/O .................................................................... 47
       6.1.1 Standard Output .................................................. 47
       6.1.2 Standard Input ..................................................... 48
       6.1.3 File I/O .............................................................. 49
# CONTENTS

6.2 Strings ................................................................. 50
6.3 Vector ................................................................. 54
  6.3.1 Creation and Handling ......................................... 54
  6.3.2 Loops over a Vector ........................................... 56
6.4 Algorithms – Introduction ......................................... 58

7 Hands-On Exercises for Part 1 ............................... 61
  7.1 Statistics ................................................................. 61
    7.1.1 RNG – Random Number Generation .......................... 61
    7.1.2 MCMC – Markov Chain Monte Carlo .......................... 62
    7.1.3 Mean, Standard Deviation, Median, Quantiles with Algorithms ............................................... 63
    7.1.4 Integrals using MC Methods ................................... 64
    7.1.5 Gambling ............................................................... 65
  7.2 Algebra ................................................................. 65
    7.2.1 Vector-Vector-Multiplication .................................. 65
    7.2.2 Matrix-Vector-Multiplication .................................. 66
    7.2.3 Matrix-Matrix-Multiplication .................................. 66
  7.3 Cryptography – Vigenère-XOR Encryption .................. 66
  7.4 Numeric ................................................................. 67
  7.5 STL Containers and Algorithms – Sieve of Eratosthenes .... 69
  7.6 Computer – Unix Tools ............................................ 69
    7.6.1 wc – word count .................................................. 69
    7.6.2 grep – text search in files ...................................... 70
    7.6.3 More Unix Tools? .................................................. 70

II User Defined Data Structures, Classes, and Object-Oriented Design 71

8 User Defined Types ................................................. 73
  8.1 Union ................................................................. 73
  8.2 Enum ................................................................. 73
  8.3 Struct ................................................................. 74

9 Classes ................................................................. 79
  9.1 Struct as a special Class ........................................... 79
    9.1.1 Member Functions ............................................... 79
    9.1.2 Access Specifier – a first Glimpse ......................... 80
    9.1.3 Object-Orientation at the Pulse – Abstraction ............ 81
    9.1.4 Structs are Classes? ............................................ 82
    9.1.5 Project Plans – Where should I put my Classes? .......... 82
    9.1.6 Function Overloading and Default Parameters ............ 84
  9.2 Constructors/Destructors – special Class Member Functions 84
    9.2.1 The Default Constructor ....................................... 84
    9.2.2 Initializing Constructors ....................................... 86
    9.2.3 The Copy Constructor ........................................... 87
    9.2.4 Constructors and Access Specifiers ......................... 90
    9.2.5 The Destructor .................................................... 90
  9.3 Lifetime of Objects in Value Semantics ..................... 91
  9.4 The "this" Pointer ................................................... 92
  9.5 Const Correctness ................................................... 92
  9.6 Some Words about Class Design .................................. 93

10 Dynamic Memory Management ................................ 95
  10.1 Operators new and delete ....................................... 95
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2 Operators new[] and delete[]</td>
<td>96</td>
</tr>
<tr>
<td>11 The Static Keyword</td>
<td>99</td>
</tr>
<tr>
<td>11.1 Reference Counting</td>
<td>99</td>
</tr>
<tr>
<td>11.2 Object only on Heap</td>
<td>100</td>
</tr>
<tr>
<td>12 Lambda Expression – Capture Clause; Function Pointer and Function Objects</td>
<td>101</td>
</tr>
<tr>
<td>13 CMake Configuration</td>
<td>105</td>
</tr>
<tr>
<td>13.1 CMake Variables and Configuration Adaptation</td>
<td>105</td>
</tr>
<tr>
<td>13.1.1 POSIX Tools to investigate Executables and Libraries (mostly Linux)</td>
<td>105</td>
</tr>
<tr>
<td>13.1.2 Libraries (Linux)</td>
<td>106</td>
</tr>
<tr>
<td>13.2 GUI Tool: ccmake</td>
<td>106</td>
</tr>
<tr>
<td>14 Refactoring</td>
<td>107</td>
</tr>
<tr>
<td>15 Hands-On Exercises for Part 2</td>
<td>109</td>
</tr>
<tr>
<td>15.1 C++/Python Coupling – PyBind11 (Tutorial)</td>
<td>109</td>
</tr>
<tr>
<td>15.2 Statistics</td>
<td>111</td>
</tr>
<tr>
<td>15.2.1 The CERN ROOT C++ Library (Tutorial)</td>
<td>111</td>
</tr>
<tr>
<td>15.2.2 Full-Text Analysis – Maps</td>
<td>117</td>
</tr>
<tr>
<td>15.2.3 A Histogram Class</td>
<td>117</td>
</tr>
<tr>
<td>15.2.4 Physics: MC Toy Model for Energy Straggling</td>
<td>119</td>
</tr>
<tr>
<td>15.2.5 Bayes MCMC Parameter Fitting</td>
<td>121</td>
</tr>
<tr>
<td>15.3 Algebra</td>
<td>122</td>
</tr>
<tr>
<td>15.3.1 Square Matrix Class</td>
<td>122</td>
</tr>
<tr>
<td>15.3.2 Operations with Matrices and Vectors</td>
<td>123</td>
</tr>
<tr>
<td>15.4 Geometry – Geometric Objects and Operations</td>
<td>123</td>
</tr>
<tr>
<td>15.5 Refactoring</td>
<td>123</td>
</tr>
<tr>
<td>15.6 STL Containers and Algorithms</td>
<td>123</td>
</tr>
<tr>
<td>16 Operator Overloading</td>
<td>127</td>
</tr>
<tr>
<td>16.1 Operator Overloading in Action</td>
<td>127</td>
</tr>
<tr>
<td>16.2 Operator Overloading by Example</td>
<td>128</td>
</tr>
<tr>
<td>16.2.1 Assignment Operator, =</td>
<td>128</td>
</tr>
<tr>
<td>16.2.2 Arithmetic Operations: +=,-,=,/,%=</td>
<td>129</td>
</tr>
<tr>
<td>16.2.3 Arithmetic Operations: +,-,*,/,%</td>
<td>130</td>
</tr>
<tr>
<td>16.2.4 Type Conversion Operators aka User-Defined Conversion</td>
<td>132</td>
</tr>
<tr>
<td>16.2.5 Comparison Operators: ==, !=, &gt;, &lt;, &gt;=, &lt;=</td>
<td>133</td>
</tr>
<tr>
<td>16.2.6 Pre-/Post-In-/De-crement</td>
<td>134</td>
</tr>
<tr>
<td>16.2.7 Index-Operator</td>
<td>135</td>
</tr>
<tr>
<td>16.2.8 Iterators</td>
<td>136</td>
</tr>
<tr>
<td>16.2.9 Function Objects aka Functors</td>
<td>139</td>
</tr>
<tr>
<td>16.2.10 Streaming Operators</td>
<td>141</td>
</tr>
<tr>
<td>16.2.11 Overloading Operator new, and Advanced Memory Management</td>
<td>143</td>
</tr>
<tr>
<td>17 Advanced I/O</td>
<td>145</td>
</tr>
<tr>
<td>17.1 std::cout, std::cerr and std::clog</td>
<td>145</td>
</tr>
<tr>
<td>17.2 Stringstreams</td>
<td>145</td>
</tr>
<tr>
<td>17.3 Binary I/O</td>
<td>146</td>
</tr>
<tr>
<td>17.4 User-Defined Stream Objects</td>
<td>148</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.4 Boost – Named Function Arguments (Tutorial)</td>
<td>212</td>
</tr>
<tr>
<td>24.5 Algebra – Square Matrix and Vector</td>
<td>213</td>
</tr>
<tr>
<td>24.6 Statistics – CERN ROOT Fitting (Tutorial)</td>
<td>214</td>
</tr>
<tr>
<td>24.7 Optimization Session (Tutorial)</td>
<td>217</td>
</tr>
</tbody>
</table>

## V Templates, Generic Programming and Design Patterns

### 25 Templates

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.1 Motivation</td>
<td>221</td>
</tr>
<tr>
<td>25.2 Function Templates</td>
<td>222</td>
</tr>
<tr>
<td>25.3 Class Templates</td>
<td>223</td>
</tr>
<tr>
<td>25.4 Template Non-Type Arguments</td>
<td>225</td>
</tr>
<tr>
<td>25.5 Template Specialization and Default Template Parameters</td>
<td>226</td>
</tr>
<tr>
<td>25.5.1 Template Specialization</td>
<td>226</td>
</tr>
<tr>
<td>25.5.2 Default Template Parameters</td>
<td>226</td>
</tr>
</tbody>
</table>

### 26 Generic and Template Meta-Programming

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1 Generic Programming</td>
<td>229</td>
</tr>
<tr>
<td>26.1.1 Simplification of Algorithm Functions</td>
<td>229</td>
</tr>
<tr>
<td>26.1.2 Example: max Function for Containers</td>
<td>230</td>
</tr>
<tr>
<td>26.2 Template Meta-Programming</td>
<td>231</td>
</tr>
<tr>
<td>26.2.1 Example: Factorial Numbers in Template Meta Language</td>
<td>231</td>
</tr>
<tr>
<td>26.2.2 Short Remark on Compile-time Execution/Computation</td>
<td>232</td>
</tr>
<tr>
<td>26.3 Application: Variables with Physical Units</td>
<td>232</td>
</tr>
</tbody>
</table>

### 27 A Short Glimpse on Design Patterns and Idioms

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.1 Smart Pointers</td>
<td>235</td>
</tr>
<tr>
<td>27.2 Singleton</td>
<td>236</td>
</tr>
<tr>
<td>27.3 Factories</td>
<td>237</td>
</tr>
<tr>
<td>27.4 PIMPL</td>
<td>239</td>
</tr>
<tr>
<td>27.5 Curiously Recurring Template Pattern (CRTP)</td>
<td>240</td>
</tr>
</tbody>
</table>

### 28 C++ – Advanced Topics

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.1 Visualization – Simple Ray Tracing</td>
<td>245</td>
</tr>
<tr>
<td>29.2 Algebra</td>
<td>248</td>
</tr>
<tr>
<td>29.2.1 Templated Matrices and Vectors</td>
<td>248</td>
</tr>
<tr>
<td>29.2.2 Eigen Library (Tutorial)</td>
<td>248</td>
</tr>
<tr>
<td>29.3 Template Meta-Programming and Type Traits (Introduction)</td>
<td>250</td>
</tr>
<tr>
<td>29.4 Named Parameters without Boost (Tutorial)</td>
<td>251</td>
</tr>
</tbody>
</table>

## VI Appendix

### A Setup of Development Environment

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 General Considerations</td>
<td>257</td>
</tr>
<tr>
<td>A.2 Setup of Development Environment under Linux</td>
<td>257</td>
</tr>
<tr>
<td>A.2.1 Basic System</td>
<td>257</td>
</tr>
<tr>
<td>A.2.2 GCC Compiler</td>
<td>257</td>
</tr>
<tr>
<td>A.2.3 Boost</td>
<td>259</td>
</tr>
<tr>
<td>A.2.4 CERN ROOT</td>
<td>261</td>
</tr>
<tr>
<td>A.2.5 Eigen</td>
<td>262</td>
</tr>
</tbody>
</table>
B  Extended CMake-Tutorial  263
   B.1  Tutorial: How to use CMake to build Software  ...........................................  263
      B.1.1  The simple CMake Configure and Build Process  ....................................  263
      B.1.2  Adaptation of Configuration  .................................................................  264
      B.1.3  Testing  .....................................................................................................  264
      B.1.4  Software Development Process  ................................................................  264
   B.2  Tutorial: How to write CMakeLists.txt Files ....................................................  265
C  Cheat Sheets, Web-Links, Literature  267
   C.1  Bash/Linux Tools ............................................................................................  267
   C.2  CMake ...........................................................................................................  267
   C.3  Git ..................................................................................................................  267
   C.4  Doxygen .........................................................................................................  267
   C.5  C++ .................................................................................................................  267
D  Proposals for Solutions for the Hands-On Exercises of Part 1  269
   D.1  Solution: Statistics .........................................................................................  269
      D.1.1  Solution: RNG .........................................................................................  269
      D.1.2  Solution: MCMC ......................................................................................  269
      D.1.3  Solution: Mean, Standard Deviation, Median, Quantiles with Algorithms ......  270
      D.1.4  Solution: Integrals using MC Methods ......................................................  272
      D.1.5  Solution: Gambling ..................................................................................  274
   D.2  Solution: Algebra ............................................................................................  275
      D.2.1  Solution: Vector-Vector-Multiplication .......................................................  275
      D.2.2  Solution: Matrix-Vector-Multiplication ......................................................  276
      D.2.3  Solution: Matrix-Matrix-Multiplication .....................................................  277
   D.3  Solution: Cryptography – Vigenère-XOR Encryption .......................................  277
   D.4  Solution: Numerics .........................................................................................  279
   D.5  Solution: STL Containers and Algorithms – Sieve of Eratosthenes ..................  281
   D.6  Solution: Computer – Unix Tools .......................................................................  281
      D.6.1  wc – word count .......................................................................................  281
      D.6.2  grep – text search in files ...........................................................................  282
E  Proposals for Solutions for the Hands-On Exercises of Part 2  285
   E.1  Solution: C++/Python Coupling – PyBind11 .....................................................  285
   E.2  Solution: Statistics .........................................................................................  286
      E.2.1  Solution: The CERN ROOT C++ Library ...................................................  286
      E.2.2  Solution: Full-Text Analysis – Maps .............................................................  287
      E.2.3  Solution: A Histogram Class ....................................................................  288
      E.2.4  Solution: Physics: MC Toy Model for Energy Straggling .............................  294
      E.2.5  Solution: Bayes MCMC Parameter Fitting ..................................................  295
   E.3  Solution: Algebra ............................................................................................  298
      E.3.1  Solution: Square Matrix Class ...................................................................  298
      E.3.2  Solution: Operations with Matrices and Vectors .........................................  299
   E.4  Solution: Geometry – Geometric Objects and Operations ...............................  299
   E.5  Solution: STL Containers and Algorithms .......................................................  301
F  Proposals for Solutions for the Hands-On Exercises of Part 3  305
   F.1  Solution: Visualization ......................................................................................  305
      F.1.1  Solution: Image Class – PPM File Format ...................................................  305
      F.1.2  Solution: Image Class – Color Map, Mandelbrot Set ...................................  306
      F.1.3  Solution: Paraview File I/O ..........................................................................  309
   F.2  Solution: Algebra .............................................................................................  311
      F.2.1  Solution: Operator Overloading for Matrices and Vectors ..........................  311
CONTENTS

F.2.2 Solution: CERN ROOT Linear Equation Solver ........................................ 317
F.2.3 Solution: Function Object Algebra ......................................................... 318
F.3 Solution: STL Algorithms – Functors instead of Lambda Functions ............ 319
F.4 Solution: Numeric ..................................................................................... 319
  F.4.1 Solution: Boost.odeint – ODE Solver ................................................. 319
  F.4.2 Solution: Aerodynamics and Complex Numbers .................................. 322
F.5 Solution: Debugging Session .................................................................... 325

G Proposals for Solutions for the Hands-On Exercises of Part 4 ................. 329
  G.1 Solution: Cryptography – RSA Encryption .............................................. 329
  G.1.1 Addendum: Boost Program Options .................................................... 335
  G.2 Solution: Visualization ........................................................................... 336
    G.2.1 Solution: Visualization – Simple 2D Drawing (Tutorial) ................... 336
    G.2.2 Solution: Visualization – Image Processing ........................................ 336
  G.3 Solution: Numeric – PDE Solver (Tutorial) ............................................. 338
  G.4 Solution: Boost – Named Function Arguments (Tutorial) .................... 340
  G.5 Solution: Algebra – Square Matrix and Vector ....................................... 340
  G.6 Solution: Statistics – CERN ROOT Fitting (Tutorial) .......................... 343
  G.7 Solution: Optimization Session (Tutorial) .............................................. 344

H Proposals for Solutions for the Hands-On Exercises of Part 5 ................. 347
  H.1 Solution: Visualization – Simple Ray Tracing ......................................... 347
  H.2 Algebra .................................................................................................. 354
    H.2.1 Templatd Matrices and Vectors ........................................................ 354
    H.2.2 Solution: Eigen Library (Tutorial) ..................................................... 357
  H.3 Solution: Template Meta-Programming and Type Traits (Introduction) .... 357
  H.4 Solution: Named Parameters without Boost (Tutorial) ........................ 358

Bibliography ................................................................................................ 361
Chapter 1

License Terms and Conditions

1.1 License

This document is published under a Creative Commons License (CC BY-SA 4.0).

1.2 Disclaimer

This course is not approved or endorsed by any C++ Language Authority.
CHAPTER 1. LICENSE TERMS AND CONDITIONS
Part I

Introduction to Software Engineering
/ Basic Syntax Elements of C++
Chapter 2

Introduction

Programming languages such as C++ are devices to help you think and plan workflows in a systematic, logical, and consistent – say structured – way. C++ contains many language features, and comprises different paradigms such as procedural, object-oriented, or functional programming. As with many other contemporary programming languages, the difficulties of writing code lie in making design decisions – which language features to use, and in which way.

This course covers an introduction to the C++ language features (types and data structures, basic keywords, flow control structures, templates, ...), some of the paradigm ideas, the software development process including program design, project management, debugging, and more (cmake, doxygen, git, ...).

The straightest way to learn a programming language is probably to have the need to apply it to a specific problem. If you have a specific problem or project in mind, this would be perfect! However, we offer Hands-on tutorials with application examples out of the scientific, engineering, and mathematical fields, which are usually met in the university environment. In this way, we want to disburden the entry and access to the skills of programming for prospective and advancing scientists and engineers.

Often, your will be confronted with the complex situation of programming together with several programmers – possibly on different levels of experience and skills – in a development team. There might be third party interests (customers, bosses, project partners, ...), and other circumstances, under which you try to keep sustainability, reliability (correctness, stability), maintainability, efficiency, etc. of your code. We will introduce some tools commonly used in software engineering (build tools, versioning tools, debugging tools, ...), but also some useful approaches and methods for software design.

Some remarks may be in order before we start out. The programming novice usually will not comprehend all the ideas, concepts and necessities of C++ right from the outset. That is normal! Even programmers being in business for many years still discover new things in the course of their work while facing new challenges. So, learning a programming language is a long process, and will probably never finish. Be happy! Because you can look forward to learn always something new until the end of your life!

C++ is extremely versatile, and allows to do things in many different ways. Bjarne Stroustrup (the creator of C++) once formulated it approximately so, that "C++ shall support programmers to simplify their work; and does not prohibit things". This versatility of ideas and concepts in C++ is very overwhelming for beginners as they usually are unsure which language feature to use for which task or problem. At the same time, you can be rather sure that you did not find the most efficient way, which a C++ guru with all his experience would recommend. And if you found, for instance, five ways to solve a problem, and present them at the next meeting or conference, somebody will come up with a sixth solution you have not thought of – or understood before. That is fully okay and normal! Don’t worry! Don’t be upset! Just be encouraged to stay patient, and focus on your problem at hand. You will learn in the course of practicing C++!

That is the reason why we designed this course in the form as it is – with a lot of exercises (to become familiar with the language features) and hands-on sessions with real world problems (which
were formerly already solved, such that you can compare to those solutions if you want). Instead of our hands-on problems, you can but choose your own tasks and problems if you have some. The more important a problem is for you to solve, the better is your motivation to understand the concepts of 

C++. And being able to solve problems via a programming language makes you access also easier the fields of applications such as mathematics, physics, engineering, etc., because a programming language is not a matter of self-sufficiency. So at least is our experience.

You may take this introduction as sort of a mantra – for motivation and comfort when you despair, or as guideline. You are never alone! So, help is always at hand if needed.

Otherwise, we wish you pleasure and fun in learning and experiencing progresses in this course.

How to work with this Tutorial ...

We explicitly do not strive for a reading book! Practicing programming is the fastest way of learning, after all. There are inline examples you can use to investigate certain language features or idioms. Some are in the shape of a more extended exercise. But that’s up to you whether you want to go deeper or not. There are but also hands-on exercises at the end of each part which offer the chance to test the learned skills in applications. Nonetheless, it is not intended that you will pass through all of these hands-on either! Some topics are simply not interesting for you! Rather it may be that you have your own software project intents or applications, and you should try to apply your programming skills there!

Disclaimer

There might still be typos and errors (code errors) inside the tutorial. We did not leave them intentionally inside the script – and for sure not to annoy you! Also, things may change unnoticed by us. And, after all, also we are far from being perfect. We are aware that we do not know everything about the C++ standard. So, some things we announced as C++ standard might not be so, and thus fail when using other compilers. We apologize for such occurrences.

On the other hand, we consider it as positive, when you recognize or discover such dissonances. This means you approached a level to be able to do so ;)

Last, but not least, you can determine what you take out of the course! We cannot cover all the participants’ requirements or didactical needs. We just offer you the chance to learn. But the learning process must be accomplished by you. And only you may know how you can learn best!
Chapter 3

Software Development Projects

It is usually a good habit to consider each programming enterprise as a software development project that has to be managed – that is, which must be planned, tested, documented, etc. The larger a project grows, i.e. the more developers might be involved, or the more submodules and functionalities are added, the more important is this management to avoid unnecessary problems or even failure.

But for the moment, we start small, just to overview the principles of software management.

3.1 A first Project – the first C++ Program

A C++ software project consists of several subfolders, containing several source code (with ending .cpp, .C, .cxx, .hpp, .H, .h) and build configuration files (all in ASCII).

3.1.1 The first C++ Program

In the simplest case, we have just one such ASCII file, e.g. with the ending .cxx. Endings are not really essential. But build tools are usually sensible to them, to work in an automatic fashion. So, better stick with the conventions!

Let’s start within a new project folder, we call it ”01” (in lack of creativity), and create the following file main.cxx:

```c++
// main.cxx
#include <iostream>
int main() {
    std::cout << "Cheers!\n";
}
```

Once compiled, the program only prints ”Cheers!” (and a new line) to the screen. You can compile this via

```
g++ -o main main.cxx
```

and executed via (Linux)

```
./main
Cheers!
```

Let’s have a look on the single lines of the file main.cxx. The first line contains a comment. Each character after // until the end of the line is ignored by the compiler. So, you can also put a comment after each valid C++ statement. The second line contains an include macro that includes all of iostream into the file during the first step of compilation. We need it to include the std::cout object for printing in line 4. Details about printing and header file inclusion will be postponed until a bit later.
In that .cxx-file, also the function main is defined (function declarations and definitions, specifically that for main, are also postponed for a while), which is the entrance point for EACH C++ program. Everything must go through this function.

The braces { and } embrace a list of statements. This holds not only for functions, as we will see later. Each valid statement finishes with a semicolon ;. C++ is designed such that you could write the full code (all statements semicolon-separated) in only one long line (exception are the #include and other so-called pre-processor directives, which do not directly belong to C++). That this is not nicely readable, becomes obvious immediately. So, it is a good idea to start from the beginning to structure the code also visually.

What a statement is, will be explained in the following sections.

Nota Bene

We have prepared a Virtual Machine with Debian for Virtual Box. Mac OS users have essentially the same environment on their system, and can take over the workflows and tools presented here. Similarly, the Windows Subsystem for Linux (WSL) under Windows 10 (together with the Debian subsystem) allows users to essentially work with all the Linux tools as presented here.

There are but more and different build tools and tool chains out there, as well as compilers, we cannot cover here. We focus dominantly on GCC’s C++ compiler and the Linux (CMake-Make) tool chain. Also important to note is that the C++ source code can be ported to different systems (as long as you do not use system specific headers). But not the binary executables! They are mostly executable only on the system (OS and hardware architecture) for which they were built!

For those, not having a compiler at hand, there are several web pages available that can be temporarily used for smaller programs and/or for testing:

- [cpp.sh](https://cpp.sh)
- [wandbox.org](https://wandbox.org) (contains also boost)
- [godbolt.org](https://godbolt.org) (compiler explorer - for optimization)

### 3.1.2 Basics of CMake – the Build-Tool

As you probably don’t want to permanently remember the compile command string, which will become very cumbersome when your projects grow larger, we introduce here the build tool CMake [20, 4]. Under Linux, CMake creates Makefiles, which can then be executed via make. So, you only need to remember essentially make.

There are dozens of other configuration and build tools. But CMake has some advantages as outlined later.

If you have pip available, a pip install --user --upgrade cmake ninja can provide you with a rather complete build tool chain.\(^1\)

#### Basic Setup

Because you have just one file in your (current) project folder, the corresponding setup file – which is always CMakeLists.txt – is rather simple. So, create an ASCII file with the name CMakeLists.txt, and the following content!

```cmake
cmake_minimum_required (VERSION 3.5)
project (first)
add_executable(first main.cxx)
```

The first line specifies the minimum CMake version necessary to build the project. Although older versions might work still, it is a good idea to specify here a version number, which was successfully

\(^1\)Ninja is a tool comparable to make. See below.
tested. You can use

```
$ cmake --version
```
to figure out, which version of CMake you are currently using.

The second line specifies the project name – and also the name of the later executable – `first`! The last line just states that the file `main.cxx` shall be added to the project. The CMakeLists.txt file needs to be changed only when you change something in the project settings – add more files or folders, change names of files, etc. We will step-by-step extend the capabilities of CMakeLists.txt in the subsequent chapters where needed to include own or third-party libraries, tests, etc..

### How to configure and build Projects

The most essential steps of building a project are to configure and to build the executable. Configuration – if successful, and no changes are made to the project – only needs to be done once in the beginning.

The so-called "out-of-source" build is accomplished by creating a folder (traditionally `build`), change to it and call `cmake`. So let’s assume to be in the project folder:

```
$ mkdir build && cd build
$ cmake ..
```

CMake checks for the compiler and possibly other required tools, libraries, environments, and so on. Once finished successfully, the configuration is complete, and the executable can be build. This is done by executing `make`.

```
$ make
```

Now, an executable `first` should have been created in the current build directory. Each time you change something in the original file `main.cxx` file, you can re-issue `make` (inside the build folder) to update the executable. Note that a re-configuration via `cmake` is NOT necessary (as long you do not change something in the project structure, i.e. adding new files to the project, renaming files, changing CMakeLists.txt contents, etc.)!

---

2Meaning that file in the source tree!
Remarks

1. If something goes massively wrong during the build process, you can always delete (remove) the build folder and restart. This is a big advantage of out-of-source builds. Furthermore, you can have several build folders for each with different build options (e.g., different optimization flags, debugging, etc.).

2. This might appear rather complex at the very beginning. And sure, for a "Hello World" example it is. But your projects will gain more complexity, as you will see in the coming chapters. Project management is then of paramount importance.

3. *Make* comes with some nice extra features. First of all, you see the advancement of the build process—in most cases rather colorful. *Make* is furthermore capable of building (i.e., compiling and linking) the source files in parallel (check option `-j`), which is of utmost importance for large projects. *Make* furthermore compiles only necessary files, i.e., those that were changed since the last build (and depending files). This saves a lot of time during repeated builds.

4. *Make* by itself is actually an independent tool with its own syntax. But *CMake* makes it unnecessary to learn this syntax. Furthermore, *CMake* is platform-independent, and can also be used in conjunction with Visual Studio under Windows. So, learning *CMake* simplifies the process of creating platform-independent software projects.

5. *CMake* configures *Make* in a way convenient also for debugging output during the build process. Simply issue the following.

   ```bash
   $ make VERBOSE=1
   ```

   *CMake* builds also POSIX-conform Makefiles, with certain default targets. So, for instance,

   ```bash
   $ make clean
   ```

   cleans out the build (removes object and other temporary files). This is but not that invasive as to remove the build folder.

6. *CMake* is not bound to *make* in order to build the code. There are other so-called *generators*. Using `cmake -G` lists which are available on your system (but usage depends on that you in fact also have installed those tools then). If you don’t want to care about that, *CMake* also allows one to be generic. The above multi-step process of creating a build directory, configuring the project, and building the code, can be accomplished in the rather following short and generic two step process.

   ```bash
   $ cmake -B build
   $ cmake --build build
   ```

   Furthermore, you can also build out-of-source. If your source code is located under some path, you can also execute

   ```bash
   $ cmake -B build -S <path-to-TopLevel-CMakeLists.txt>
   $ cmake --build build
   ```

   In this way, you can keep the source tree clean. The final executable is then located as usual in the `build` directory.
3.2 Some Words about Editors

The editor you use – can also be an IDE (Integrated Development Environment) as Visual Studio – should have some features you should know of and must learn.

1. It must be capable of writing ASCII files.
2. It should support language specific syntax highlighting, parenthesis matching, and tab structuring (possibly user configurable).
3. It should have short-key support (possibly adaptable), which enhances coding speed.
4. It needs capabilities for copy/paste and search/replace. Copying/Moving/Deleting should be capable to be done word, line and block-wise.
5. It should be capable of opening several files at the same time (different tabs/buffers) – possibly show two of them in parallel (for comparing).
6. For working remotely (files are on a remote system), there should be either a terminal mode for the editor (running then via SSH on the remote system), or have a SSH client support.
7. Syntax checking and lookup, as well as autocompletion are accelerators for software development.

Common editors (with in part long tradition) for C++ are Vim, Emacs, Gedit, Jedit, Notepad++, Kate. IDEs can be Visual Studio (MS or Intel), Eclipse, or KDevelop. Not all are supported on all operating systems, however. Testout what suits you best!

Exercise:

a) Copy the complete `std::cout` line from `main.cpp`, and change the text printed in this second output line.

b) Search for all occurrences of semicolon `;`!

c) Find all "`std::cout`" and replace them by "`std::cerr`"! Finally, replace the "\n" by "`<< std::endl`" (be careful with the quotation marks!!)

d) Remove the second output statement (as efficiently as possible)!

"`std::cerr`" is the output stream to standard error – this is unbuffered output, usually used for error messages. "`std::endl`" is the same as "\n" + "`std::flush`", where the latter flushes the buffers. As flushing buffers is usually a performance killer, we propose to use "\n" for a normal line break. Once the output buffer is full, it will get flushed automatically.

3.3 Git – a Versioning Tool

Programming will in most cases be incremental. That is, you will add some functionality in one part of your source code. Or, you change (remove a bug) in a certain place. So, your code, as your project, evolves over time. One part of documenting is "What you have done, and when?".

Another issue is that you might have changed something that afterwards reveals not to work as you wanted. How do you come back to the last working code base?

Finally, there might be different things you would like to try out – say, different development directions. As software development is a design process, you would like to be flexible. Of course, you could copy your folder again and again. But the chance is good that you lose the overview.

Git [19, 11] can help you accomplishing all these tasks. There are several other versioning systems on the market (SVN – subversion – being one also wide spread) – each with different strengths and weaknesses. But be aware, that none is actually a backup system!
How does (g)it work?

At the start of a project, or also during a running project, you need to initialize Git once. This is done by executing the following inside the project folder.

```
$ git init
```

Next, also necessary only once, it is a good habit to specify some contact information.

```
$ git config user.name "My Name"
$ git config user.email "MyName@bla.com"
$ git config --list
```

Instead of the quoted placeholders, you of course enter your own name and e-mail address. These will be printed whenever you look into the logs about changes in your project. If you want to specify your name and e-mail address globally for all projects (as default), you can add a `--global` after `git config`.

You can check the status of your Git project via

```
$ git status
```

After the pure initialization, you will obtain a message that `CMakeLists.txt` and `main.cxx` are not versioned, yet. To help this, we add these files to our project:

```
$ git add main.cxx CMakeLists.txt
$ git commit -m "My first commit"
```

In the second line, we accomplished a so-called commit. This means that the files in the current state are recorded by Git. Another `git status` tells you that the working directory is up-to-date. This means, the source code in the versioned files, and the information in the Git database are consistent. You can watch the logs of your project evolution via.

```
$ git log
commit a68961cbd214472375937fedaf053e892569fe16
Author: My Name <MyName@bla.com>
Date: Fri Jun 21 14:54:11 2019 +0200

My first commit
```

You can see when a commit was done, and by whom, and the comment tells you possibly what was changed. Each commit gets a unique hash label, by which it can be identified.

Okay. Let’s change something – say "Cheers" by "Hello" inside `main.cxx`. When you enter `git status`, Git will tell you that you changed `main.cxx`. A `git diff` also tells you what was changed.

```
$ git diff main.cxx
diff --git a/main.cxx b/main.cxx
index ee9e82b..548172b 100644
--- a/main.cxx
+++ b/main.cxx
@@ -1,6 +1,6 @@
// main.cxx
#include <iostream>
int main() {
```
3.3. GIT – A VERSIONING TOOL

- std::cout << "Cheers!\n";
+ std::cout << "Hello!\n";
}

If you are content with your changes, you can submit another commit,

```
$ git commit -a -m "Changed Cheers by Hello"
```

```
[master 029b1f1] Changed Cheers by Hello
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
$ git log
```

```
commit 029b1f106a9567bf5ad6d70970069e2d5b95abe9
Author: My Name <MyName@bla.com>
Date: Fri Jun 21 15:04:30 2019 +0200

Changed Cheers by Hello
```

```
commit a68961c2db214472375937fedaf053e892569fe16
Author: My Name <MyName@bla.com>
Date: Fri Jun 21 14:54:11 2019 +0200

My first commit
```

The `-a` option adds the changed files. That is the way how git handles changes: first add the files that changed, second commit to the "data base". You could separate both steps by using `git add ...` and `git commit -m "..."` in sequence. The following `git log` now shows two commits – as we have done two commits.

You can also use `git show <HASH>` or `git diff <HASH1> <HASH2>` to investigate changes.

**Exercise:** Familiarize yourself with the Git workflow. Make changes to `main.cxx` and commit them. Look for the Git status, log, and differences of different commits!

You can rename and remove files and folders using `git rm ...` and `git mv ...`, respectively. We will extend the features of Git further in the coming sections when needed.

**The dark Corners**

You will usually create also files and folders you don’t want to have versioned. For instance, backup files from editors, or the build folder for CMake. That’s fine for Git. Just enter them into a file called `.gitignore` – each line a file or folder name. Do not forget to add `.gitignore` to your Git project.

**Reversing to former Commits, or Removing Versioning**

Imagine your last commit was wrong – or buggy – and you want to return to your former commit. This can be done via

```
$ git checkout a68961c2db214472375937fedaf053e892569fe16
```

or, the hash corresponding to the commit you want to return. Be careful in doing this as all your changes done after that commit might be lost!!

For a single file, you can also return to HEAD, via

```
$ git checkout HEAD -- file
```

---

3 The leading dot makes this file invisible. You need `ls -a` to see it. But otherwise, it’s a normal file!
If you messed up anyhow with Git, you can simply remove the .git folder via

```
$ rm -rf .git
```

and reinitialize Git via `git init`. Be careful to have a working set of source code files! It can be annoying if everything you programmed got lost!

This is now the right place to mention that you should also think about a backup strategy! Git may offer something, when you use GitHub or GitLab. Here, the Git repositories are located on remote servers with own backup solutions. But that’s a different topic. Just keep backup in mind!

### 3.4 Test-Driven Development

Git allows us to "clone" a repository which afterwards becomes (more or less) completely independent of the former. So, let us clone the first project folder,

```
$ git clone 01 02
$ cd 02
```

This folder should have exactly the same content as 01 (except for the ignored, non-versioned files and folders).

#### Unit Tests

We now want to extend this project by adding some files, and changing the present files. Before doing so, we would like to introduce a program design strategy called Test-Driven Development (short, TDD). The background of this philosophy is that we first write test code that uses the code (functions, classes, ...) in the way we imagine – BEFORE we develop the actual production code. But this usage is in terms of tests, i.e. statements that result in either true (if we did a good job), or false. In the latter case, we need to improve our implementation (or at least do debugging and find the cause for the problem).

As an example, say we would like to introduce our own square-root function, implemented in terms of Heron’s method. In short, that is a iterative procedure that takes a number as input, \( a \), an initial guess, \( x_0 \), and gives out an approximation of the square-root of \( x_1 \approx a \). Starting from \( x_0 \), we iterate via the following prescription until convergence (whatever this means) is reached.

\[
x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right).
\]

After convergence, \( x \approx \sqrt{a} \).

Let us start by changing the main function to include a test function. This function has no parameters (empty parameter list, ()), as it serves solely as test container. And it does not return anything. This is expressed in C++ by the return value void. As C++ insists on return types, that is the only way to say no to return anything.

```c++
// main.cxx
#include <cassert>

void test() {
    assert((1.0==sqrt(1.0)));
}

int main() {
    test();
}
```

---

4 Here is a hardcopy as backup legitimate.

5 Can be derived by solving \( f(x) = x^2 - a = 0 \) via Newton’s iterative method for root-finding [30], \( x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \)
This will not compile, of course, because inside main.cxx, sqrt is completely unknown to the compiler. There are now different ways to cure this. This simplest is to define this sqrt function before the test function.

```cpp
// main.cxx
#include <cassert>
double sqrt(double a) {
    double x=a;
    for(int i=0; i<10; ++i)
        x=0.5*(x+a/x);
    return x;
}
void test() {
    assert(1.0==sqrt(1.0));
}
int main() {
    test();
}
```

Now, it compiles, and executes. This means the tests are successful, and we made thus a successful development step! From now on, we try to continue the development on the line keeping all tests passing – no matter whether we change something or add new functionality. If a test once passed does not pass after a change, we have to cure this before the next commit.

Some explanations. double is another word for real valued floating point number, and int stands for integer. C++ works with types. Each object (even the functions) represents a type.

The next thing is the for-loop. How it works will be explained later, as well as the C++ built-in types, and functions. Suffice it that it performs iterations, and that everything within the loop is executed repeatedly, until a stopping criterion is reached.

Let us extend the test, because \( \sqrt{1} = 1 \) is really not much saying. So, we extend test() by

```cpp
assert(1.41421356237309504880==sqrt(2.0));
```

The left-hand value was taken from another calculator. It compiles. Fine! But execution fails with

```
first: [...]/Tutorial/Day1/02/main.cxx:13:
  void test(): Assertion '1.41421356237309504880==sqrt(2.0)' failed.
Aborted
```

Okay. This was to be expected as comparing floating point numbers is really a bit brittle. So, let us try a more stable test,

```cpp
assert(fabs(1.41421356237309504880-sqrt(2.0))<1.e-10);
```

In order that this works, we have to #include <cmath>, where fabs is defined (mathematical function to return the absolute value of a real number). Where to get such an information, will be described later.

This time it compiles and runs without errors. Success!

Okay! Fine! We defined "success" for our own good. We could now extend this test for each number between \( 10^{-14} \) to \( 10^{14} \), or so. And although cautious testing in the range of interest is essential, not every number needs to be tested unless you are sure that it will be of relevance. The above test will fail at about \( \sqrt{5500} \) and larger. If these numbers are relevant, we should improve the sqrt function, e.g. by changing the start value \( x_0 \) to \( a/2 \). Also, if the precision of \( |x_n - \sqrt{a}| < 10^{-10} \) is not sufficient, we would need to change something.

Square-root is of course a silly example to test, as such a mathematical function is certainly already available in some well-tested library (std::sqrt in <cmath> btw.). But as explanation of the TDD approach it is sufficient. In future, to choose meaningful tests will be of paramount importance to convince yourself of the correctness of your code. In ideal cases, it alleviates or even avoids totally any debugging.
Exercise: Catch the case of $\sqrt{0}$ and for negative numbers (give out a warning and return 0), and add tests for the numbers 0.1, and 0.04! Add and commit the changes via Git! Check via git status and git log!

Integration and Acceptance Tests

What we showed up to now were so-called unit tests. That is tests for small pieces – called units. To rely solely on these would be foolish, because the interplay and cooperation of the different units also matter. So, the next level are integration tests, that is, tests of the more complex system of all integrated units. To define such tests is of course more difficult, as we now test a more (and maybe too) complex system.

Therefore, a reduced test facility are acceptance tests. Here, the “customer” (the later user) defines the desired overall behavior, which is to be tested. The advantage is that the customer does not need to know about implementation details – he usually is not interested in. And, the customer does not require unnecessary tests in all details, but mostly what’s important for him. CMake supports such tests via the ctest tool. You can find an example in section B.1.

3.5 Documenting Programs using Doxygen

On Comments and Good Coding Styles

Documentation of code is a matter of many debates. Some prefer a complete documentation of everything. Others think that sparse usage of documentation is better – up to the extreme that “well-written source code is self-explanatory”. A final agreement is probably never achievable.

But let us try a minimal agreement about that some documentation for certain purposes is certainly meaningful. For instance, for your own memory and understanding of your code, some comments are advisable. However, what should you comment? Some ideas can be collected by reading different Coding Styles, enacted for diverse software projects such as the Linux kernel. One good hint is section 8 of that coding style [17], which we’d like to cite.

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it’s much better to write the code so that the working is obvious, and it’s a waste of time to explain badly written code.

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 6 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

Another coding style commonly used is the Google C++ Coding Style [13], or that of OpenFOAM [31]. In large projects, style guides are inevitable in order to keep the project’s code base maintainable. Those coding styles not only prescribe comments. So, reading them thoroughly is generally a good idea in order to write readable code, because according to the Pareto Principle $^6$ [32], we can reasonably assume that code will be much more often read than written. Violating against such generally agreed style rules can seriously jeopardize the success of your software projects. Some humorous essay of doing things wrong intentionally can be found in [15]. But one can learn from it, in spite of this. More on reasoning why and how writing good code can be found in [22, 21].

More extensive documentation will become more important when you want to supply your own code to other users, or when you are working in a collaboration. So, others will have to make sense out of what you did. Please, keep this in mind! Once you made the experience of using a badly documented library, you sadly know why!

$^6$ aka 20/80 rule
3.5. DOCUMENTING PROGRAMS USING DOXYGEN

Introduction to Doxygen

We follow the description of [5]. In your project directory, issue

```
$ doxygen -g
```

This creates a file named `Doxyfile`. This file contains lots of documented key-value pairs – although a lot of values might be missing. You should set a reasonable project title `PROJECT_NAME`, a project release version number `PROJECT_NUMBER=1.0` (which should probably correlate with the CMake and Git release tag numbers), `PROJECT_BRIEF`, etc. Please, parse through the Doxyfile – most of it should intuitively clear. One good option is `OUTPUT_DIRECTORY = doc` to put all documentation into a folder called `doc`. Next, running

```
$ doxygen
```

creates the `doc` folder. Inside of it, you find two folders – `html` and `latex`. You can, for instance, open a browser with `doc/html/index.html`. Of course, there is still not much to see – just the project title, and a version number, and some things more. We should set `EXTRACT_ALL = YES`. Re-running doxygen, and refreshing the browser. Now the files item should appear in the menu, and all files found in the project are listed. That is only one, yet – `main.cxx`. But more will come.

Doxygen would be largely useless if that’s everything it can do. So, the programmer can add more information. Using a markup language – largely comments in C++ – you can add a lot of information to your source files, which Doxygen uses and presents – even in a nicely formatted manner.

```c
#include <iostream>

int main(int argc, char * argv[]) {
    std::cout << "Cheers!\n";
    return 0;
}
```

This is, of course, overkill in this extremely simple example. But more complex projects with possibly many users not being involved in those projects, but just being applicants, documentation will be their only chance to use the project’s software (mostly libraries). Doxygen has the capability also to parse the source code files and represent their relations (also the classes) graphically.
**Exercise:** Add a Doxygen documentation to your project. Add comments as shown above to your own code, and view the result via a web browser. Add the Doxygen file also to your Git-versioned files. Exclude (.gitignore) the resulting *doc* folder!
Chapter 4

Basic Elements of C++

In the following sections, we introduce the most basic C++ language features. These are expressions and statements, the built-in type system, control structures, and functions.

4.1 Expressions and Statements

Loosely formulated, the C++ compiler accepts a program as correct, if it is a sequence of syntactically correct statements. Statements can be expressions, compounds, selections, iterations, etc. (see cppreference: Statements). Each statement must be completed by a semicolon ";"! White-spaces and tabs are irrelevant and are ignored!

Expressions (cppreference: Expressions) can be assignments, arithmetic, or logical expressions. Also function calls are expressions. In the simplest and most intuitive way, mathematical expressions like 1+2, for instance, are easy to comprehend. If you don’t know what it means, programming is probably not an option for you, yet! So, C++ tried to setup the expression system in the most intuitive way on which commonly can be agreed. Addition +, subtraction -, multiplication *, division /, modulo (%), etc. should be familiar – and also how to use them (e.g. 1.3*5*6.2 is not equal to (1.3+5)*6.2). Operator precedence (cppreference: Operator Precedence) is built into C++!

An expression like 3==5 might be new for some of you. This is a comparison – a Boolean expression (being false or true). This expression gives false, as 3 ≠ 5, obviously. Also <, <=, >, >=, != for less than, less or equal, greater than, and greater or equal, not equal, are comparison operators. So, 3<5 is true. The logical conjunction or disjunction are expressed by && and ||, respectively. So, (3<5)&&(4>1) is a conjunction (logical "and"), meaning both sub-expressions must be true to become true in total. For disjunction (logical "or"), only one of the sub-expression needs to be true. That is, (3<5)||4<1) is true.

C++ knows quite some operators (cppreference: Operators). And you can learn them when needed. Bit operations like ^, <<, >>, ˜, for instance, are needed mostly in quite specific areas such as close-to-hardware programming.

4.2 Variables

You can write a program like

```cpp
#include <iostream>
int main() {
    std::cout << ((2+5)*7/2.3-5%3 == 10.3) << "\n";
}
```

It compiles and executes ... successfully, i.e. without a crash. But that is largely uninteresting in modern programming. (Maybe as misuse as old fashioned micro-calculator ...)

First of all, we want to remember and reuse results of computations inside the program. This is accomplished by variables, something like
a = (2+5)*7/2.3-5%3 ;

But before we can use a "label" (aka variable), we must declare it to the C++ compiler. Specifically, we must declare also its "type".

4.2.1 Types

Types determine the desired meaning of a variable. That is how the computer interprets them – and handles them internally. Some of the basic types are int for (at least 32-bit) integer number representation, and double for (64-bit) floating point numbers. Both are to some extent default in C++. So, the literal 2 is interpreted as int, 2.0 as double. For the above, we could naturally choose

\[
\text{double } a = (2+5)*7/2.3-5\%3; 
\]

or, we can also write

\[
\text{double } a;  \\
\text{a} = (2+5)*7/2.3-5\%3; 
\]

C++ performs also internal conversions automatically. So,

\[
\text{double } a = 1; 
\]

is correct, and the integer 1 will be converted into a floating point 1.0. Also

\[
\text{int } a = 1.3; 
\]

is valid. But 1.3 will be "narrowed", i.e. converted to 1. The compiler even does not give a warning per default!\footnote{Built-in into C++.}

C++ knows several types – fundamental types such as int and double, but also bool, float, long, char etc. \(\text{(see cppreference: Fundamental Types)}\), but also compound and user-defined types \(\text{(see cppreference: Type)}\). We will introduce them later when needed.

You can declare variables at any place in the code – before you use it (writing, reading). Modern practice is to declare and initialize variables where you need them the first time (and in the smallest possible scope – what will become clear also later).

4.2.2 Identifier Names

Variables names have to obey some rules – they can contain letters \(\text{(C++ is case-sensitive!!)}\), numbers, underscores \(\text{(see cppreference: Identifiers)}\). That is, we can write the following.

\[
\text{#include } \text<iostream> \\
\text{int } \text{main}() \{ \\
\text{double } a = 2.5*2-10./3.3+4.; \\
\text{int } A = 50\%3; \\
\text{std::cout } \ll \text"a=" \ll a \ll \text"A=" \ll A \ll \text"\n"; \\
\} 
\]

But you should not do so. Give variables reasonable names – easy to read, to comprehend their meaning, and to search for!

4.2.3 Scope and Namespaces

Curly braces \{\ldots\} can be inserted everywhere. And they can contain any sequence of statements. Such braces are used to logically structure the code in terms of blocks. Thus, function bodies are included in braces, as well as the bodies of the control structures, which are introduced in the next section. But the task of such braced code blocks goes even further. They form a so-called scope \(\text{(see cppreference: Scope)}\). Variables declared in such a scope exist only within this scope. That is a name of a variable cannot be used outside of the scope in which it was declared.

\footnote{But you can switch on warnings. How? is shown later.}
4.2. VARIABLES

```cpp
#include <iostream>
int main() {
    double a = 2.5;
    std::cout << "a= " << a << "\n"; // Error: a not declared
}
```

This will not compile!

Variables declared outside any function body (specifically that of `main`) are global variables (also called static variables), because they are declared in the global scope. Try to avoid them if possible! Something like this,

```cpp
#include <iostream>
double a = 2.5;
int main() {
    std::cout << "a= " << a << "\n";
}
```

is 100% correct. But you will notice soon that it might be easy to generate naming conflicts. Furthermore, the following is also completely correct, but confusing.

```cpp
#include <iostream>
double a = 2.5;
int main() {
    int a = 3;
    std::cout << "a= " << a << "\n"; // 3 is printed
}
```

The (re-)declaration of a within a function (or any nested scope) is legal – even with a different type. And the most inner-scope declaration hides any more outer-scope definitions. This is exactly such a naming collision as mentioned above.

C++ has therefore introduced namespaces – so-called named scopes (see `cppreference: Namespaces`).

```cpp
#include <iostream>
namespace bla {
    double a = 2.5;
}
int main() {
    int a = 3;
    std::cout << "a= " << bla::a << "\n"; // 2.5 is printed
}
```

Namespaces not only can contain variable declarations and definitions, but also functions. By means of the scope-resolution operator ::, we can always address variables and functions of namespaces. For instance, `std::cout` is a function from the namespace `std` – the namespace of the C++ standard library.

Namespaces can be nested. For instance,

```cpp
#include <iostream>
namespace bla { 
    namespace zack { 
        double a = 2.5; 
    } 
}
int main() {
    int a = 3;
    std::cout << "a= " << zack::a << "\n"; // 2.5 is printed
}
```

You should not over-stress this feature.

The global namespace can be addressed, too.

```cpp
#include <iostream>
double a = 2.5;
int main() {
    std::cout << "a= " << a << "\n";
}
```
CHAPTER 4. BASIC ELEMENTS OF C++

```cpp
int a = 3;
std::cout << " a= " << a << " \n"; // 3 is printed
std::cout << ::a << " \n"; // 2.5 is printed
}
```

That is the global namespace has no name.

**Convenience Feature – using**

Beginners often find it frustrating to write always std:: and other namespaces addresses. We can use the using keyword to declare abbreviations and aliases. **Please, do not do this in header files and the global space!!** For instance,

```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Cheers!\n" << endl;
}
```

includes the namespace std:: globally for the file. Therefore, we can use now simply cout instead of std::cout. If you use cout only inside a function (here main), we can restrict the namespace inclusion only to that function.

```cpp
#include <iostream>
int main() {
    using namespace std;
    cout << "Cheers!" << endl;
}
```

And finally, if you only need std::cout, you can make a so-called using declaration (no namespace).

```cpp
#include <iostream>
int main() {
    using std::cout;
    cout << "Cheers!" << std::endl; // endl without std:: is unknown
}
```

The more you can restrict the using declarations to inner scopes, the better, because the less the chance for naming collisions.

**Some Remarks**

You will learn about classes later in that course, and that they make namespaces of their own, to some extent. So, often the question occurs early why extra namespaces in C++. A preliminary and short answer is the following. Classes encapsulate mostly data. Namespaces encapsulate functions. Specifically, if you write software for others, this sort of encapsulation is a good habit from the beginning. It often prevents clashes of function names etc. from the outset, and alleviates possible debugging efforts if something goes awry.

**4.2.4 Arrays**

We will show you a better way than the native C-arrays, because the C++ Standard Library contains convenient, safe, and efficient replacements. But the semantics of arrays is important nonetheless. An array is declared as

```cpp
double av[10];
```

This is an array of 10 doubles (i.e. of 10 consecutive doubles in memory). The array is usually not initialized. Initializing all array elements to zero can be accomplished by

```cpp
double av[10] = {};  
```

If you want other values, it is also possible via

```cpp
double av[10] = {0,1,2,3,4,5,6,7,8,9};
```
The array index runs from 0 to 9 (that’s C legacy, and is related to the underlying pointer arithmetic)!
Access (writing or reading) the first element by means of the index operator [... ] looks as follows.

```cpp
av[0] = 11;
std::cout << av[0] << "\n";
```

Semantically, indexes are integers. So you should use only integer types (variables are also possible) to index an array.

```cpp
int i = 0;
av[i] = 11;
std::cout << av[i] << "\n";
```

Another, and rather natural way to set values of an array are loops. We postpone this until the next section.

### 4.3 Control Structures

#### 4.3.1 Conditional Branches

**Simple Branch – If-Else Construct**

A linear program is boring – and would also render programming largely useless. More common is that a program branches off conditional based on events such as user input or on data. The syntax is simple.

```cpp
if ( condition ) {
    statement block 1
} else {
    statement block 2
}
```

If `condition` is true, `statement block 1` is executed. Otherwise, `statement block 2` is executed. The `else` branch is optional. If not needed, you can be content with just a single branch.

```cpp
if ( condition ) {
    statement block 1
}
```

If the statement block consists of just a single statement, the scoping braces can be omitted.

As `condition`, you can actually use any statement. If its value is not equal to zero (as bit-sequence), it is considered **true**. As example,

```cpp
if ( true )
```

or something that is not null (check for pointers, function return values, etc.), is always resulting in the execution of the `statement block 1`. This means that conditions don’t need to be of type `bool`. They will be interpreted as `bool`. (see `cppreference: if Statement`)

**Multiple Branches – Switch-Case Construct**

It is feasible to nest deeper in `if`-conditions, such as

```cpp
if ( condition 1 ) {
    statement block 1
} else if ( condition 2 ) {
    statement block 2
} else if ( condition 3 ) {
    ...
} else {
    statement block N
}
```

But better you avoid such constructs. They are hard to read. Especially when the statement blocks consist of more than one statement.
If you have an integer variable (char, characters, is such a type, too), you can also use the switch-case structure. Its layout is as follows.

```cpp
switch ( condition 1 ) {
    case constant_expression1 : statement block 1; break;
    case constant_expression2 : statement block 2; break;
    ...
    default: statement block N;
}
```

A usage example makes things easier to comprehend. Imagine, your program requires the user to confirm or deny a certain action, you can do it like this

```cpp
#include <iostream>

int main() {
    std::cout << "Do you confirm? [y/n] : ";
    char answer;
    std::cin >> answer;
    switch ( answer ) {
        case 'y': 
        case 'Y':
            std::cout << "\nConfirmed!\n";
            break;
        default:
            std::cout << "\nAborted!\n";
    }
}
```

You can also use {...} to embrace the statement blocks. But it is not strictly needed here. `std::cin` is the standard input stream, and will be explained later. If you compile and run this program, the user is asked for a confirmation. If characters 'y' or 'Y' are pressed, we get the program output Confirmed! Any other letter – that is the default – leads to the output Aborted!

You might have noticed that after the case 'y' : line, nothing is executed. Also no break. The switch-case structure is constructed such that a fall through happens in that case, i.e. the statements of case 'Y' : are executed, too, and so on – until the first occurrence of break, which interrupts the switch-case execution. (see cppreference: switch Statement)

### 4.3.2 Loops

The next relevant program flow control structure is loops.

#### for-Loops

The prototype of a for-loop is a index-counted loop such as

```cpp
for(int i=0; i<100; ++i) {
    loop body statement block
}
```

The for-loop has an initialization part, int i=0, a condition part, i<100, which needs to result in true in order that the loop proceeds, and the index count part, ++i, which is used to increment the index. ++i has the same effect as i=i+1. The loop index variable i is visible only on the scope of the for-loop! You can increment it also by more than just one. So, i+=2 instead of ++i would increase i by two.

All these parts can be extended or misused. For instance, imagine you need to run through two arrays of length 100, but the one array forward, and the other backward. One can do this for instance by the following.

```cpp
int av[100] = {}; bv[100] = {};
for(int i=0, j=99; i<100 && j>=0; ++i, --j) av[i] = bv[j];
```

The comma operator ”,” is sometimes very useful! This construct copies the values of bv to av in reverse order. Again, if the loop body consists of only one statement, the braces can be omitted. In this case, we wrote the statement also in the same line as the for-clause. Remember that C++ programs can be written completely in just a single line. But visually better is probably the following.
4.3. CONTROL STRUCTURES

```c
for(int i=0, j=99; i<100 && j>=0; ++i, --j)
    av[i] = bv[j];
```

Or even,

```c
for(int i=0, j=99; i<100 && j>=0; ++i, --j) {
    av[i] = bv[j];
}
```

(see [cppreference: for Loop](https://en.cppreference.com/w/cpp/language/for))

For-loops can also be nested. For instance, passing through a matrix (2D array – not mentioned so far) look like

```c
double am[5][5] = {};
for(int i=0; i<5; ++i) {
    for(int j=0; j<5; ++j) {
        am[i][j] = (i*5.+j);
    }
}
```

That is essentially equivalent to

```c
double am[5*5] = {};
for(int i=0; i<5; ++i) {
    for(int j=0; j<5; ++j) {
        am[5*i+j] = (i*5.+j);
    }
}
```

when an 1D array is used.

**while-Loops**

The prototype of a while loop is

```c
while( condition ) {
    loop body statement block
}
```

So,

```c
int i=0;
while( i<100 ) {
    loop body statement block
    ++i;
}
```

more or less resembles the for-loop construct. But while for-loops traditionally are used for a loop through a fixed range of values, while-loops are used for flexible length loops. This might include possibly zero-length, if the condition is not true even initially. (see [cppreference: while Loop](https://en.cppreference.com/w/cpp/language/while))

**do-while-Loops**

If you need a while loop that but runs at least once, do-while is the mean of choice:

```c
do {
    loop body statement block
} while( condition );
```

Please note the finale semicolon! (see [cppreference: do-while Loop](https://en.cppreference.com/w/cpp/language/while))

**break and continue**

`break` is used if you want to leave the innermost loop (for, while, do-while does not matter). `continue` is used when you want to skip the rest of the loop body, but otherwise want to continue the loop to the next count. For instance,
for(int k=0; k<10; ++k) {
    if(k==8) break;
    if(k%2==0) continue;
    std::cout << " loop count " << k << "\n";
}

The output of this loop is
1
3
5
7

The even loop counts, 0, 2, 4, 6, are skipped due to continue. 8 and 9 are removed, because the break stops the loop at count \( k = 8 \). The order of the sequence of statements matters!

4.4 Functions

Functions are THE MOST IMPORTANT basic abstraction of working units. They effectively obviate code reduplication (DRY: Don’t Repeat Yourself). They enhance thus code readability, and therefore maintainability, and in consequence cause less faulty and error-prone software. Each function should serve one and at most one task (SRP: Single Responsibility Principle). Give functions expressive names, which tell you WHAT the functions do! Ideally, you can minimize documenting comments! Same holds for the function parameters.

4.4.1 Function Signature – Declaration and Definition

A function **declaration** is just an announcement of the function signature, and specification of the name of that function. Function names are also variables, and as such have to obey the same rules and restrictions as the names of variables. Furthermore, function declarations are used in header files (next chapter). A function has a return value (possibly void, meaning that it does not return anything), a name, and a parameter list. Examples are,

```cpp
void test();
double sqrt(double);
```

Functions can return any kind of object. And they can have several parameters. For instance, a Gaussian function has a variable parameter \( x \), and two parameters – mean \( \mu \) and standard deviation \( \sigma \). If we want a function that returns the value of the Gaussian function for a value of \( x \), and some parameters \( \mu \) and \( \sigma \), we can declare such a function as

```cpp
double Gaus(double x, double mu, double sigma);
```

The parameter list is a comma-separated list of types. We can later call this function as

```cpp
std::cout << Gaus(0.2) << "\n"; // x=0.2, mu=0.0, sigma=1.0
std::cout << Gaus(0.2,1.) << "\n"; // x=0.2, mu=1.0, sigma=1.0
std::cout << Gaus(0.2,1.,0.6) << "\n"; // x=0.2, mu=1.0, sigma=0.6
```

That is, we can call this function with one, two or three parameter values,

\[3\]The parameter names of the declaration and the definition don’t need to be the same.
Before a function is used, it must at least be declared before – either in a header (to be included), or via forward declaration (i.e. specification of the name and parameter type list as described above).

The definition (say: realization or implementation) of a function can be done at any place (in the global scope) in the same file or in another translation unit (another file). One only needs to care that the function is defined only once – otherwise does the compiler or linker yelp.

Definition of a function means that a function body in braces is added. If you need a function only in the file with the main function, you can combine declaration and definition. For instance,

```cpp
#include <cmath>
#include <iostream>
double Gaus(double x, double mu=0., double sigma=1.) {
    using namespace std;
    return exp(-0.5*(x-mu)*(x-mu)/(sigma*sigma))/(sqrt(2.*M_PI)*sigma);
}
int main() {
    std::cout << Gaus(0.2) << 
    std::cout << Gaus(0.2,1.) << 
    std::cout << Gaus(0.2,1.,0.6) << 
}
```

The return Keyword

Inside a function you can do whatever you like (as long as it is correct C++). At any place inside the function, you can issue a return (+ a value of the functions return type), which makes the program leave the function and return to the function that called it (with the return value). For instance, if we want to have a Gaussian function call result in zero if \( x < 0 \), we could achieve this by the following.

```cpp
#include <cmath>
#include <iostream>
double Gaus(double x, double mu=0., double sigma=1.) {
    if(x<0)
        return 0;
    using namespace std;
    return exp(-0.5*(x-mu)*(x-mu)/(sigma*sigma))/(sqrt(2.*M_PI)*sigma);
}
int main() {
    std::cout << Gaus(-0.2) << "\n"; // 0
    std::cout << Gaus(0.2) << "\n"; // 0.391043
}
```

This function has two return points. And it returns always to the caller – the line where it was called from. We only need to be careful not to inadvertently cut away some program flow paths. Some occasional mistake might be integer comparison,

```cpp
int do_something(int a) {
    if(a==2)
        return 0;
    return 1;
}
```

When you carelessly write in the if-condition \( a=2 \) (assignment, NOT comparison!), do_something you will always return zero, because \( a=2 \ "returns" 2 \) (no matter what a was before), which is not equal to zero – thus, always true.

### 4.4.2 Function Overloading, Name Resolution, and Implicit Casting

A function has a name and a parameter list with a unique sequence of parameter types. The return value does not play a role in the following considerations. C++ allows you to redefine a function, i.e. to reuse a function name, as long as the parameter list is unique, i.e. different from all previously defined functions with that name. This feature becomes quite handy when you need functions doing
the same thing but for different parameter types. But when we talk about function overloading, we also need to talk about name resolution. The business is about the following. Imagine, we have overloaded a function `f` twice as

```cpp
double f(double x, int i) { return x; } // 1
double f(int i, double x) { return x; } // 2
int main() {
    f(0.1,2); // ok: takes variant 1
    f(1,2.); // ok: takes variant 2
    f(1,2);  // Error: does not compile
}
```

The first function call in `main`, takes 0.1 a `double`, and 2 an `int`. There is one exact match – variant 1. The second function call takes 1 an `int`, and 2. a `double`. But in the third function call, we supply two `ints`, such that the compiler cannot and does not decide, which function `f` to call. The compiler error message looks like this,

```text
test.cxx: In function ‘int main()’:
test.cxx:15:12: error: call of overloaded ‘f(int, int)’ is ambiguous
  f(1,2);
  ^
test.cxx:9:8: note: candidate: double f(double, int)
  double f(double x, int i) { return x; }
  ^
test.cxx:10:8: note: candidate: double f(int, double)
  double f(int i, double x) { return x; }
  ^
```

To cure, we can add another function overload like e.g.

```cpp
double f(int x, int i) { return x; }
```

(Return values do not matter. You could return also `int` in some of the variants, while `double` in others.) Now, `f(1,2)` compiles as there is an exact match.

Or, an alternative cure option, you can remove any overloading and keep, for instance, only variant 1. In that case, both `f(1,2.`) and `f(1,2)` do not find an exact match. But the compiler knows only one function `f(double, int)` and can make your calls matching. He does so by casting the types to those needed. In `f(1,2.)`, for instance, 1 is cast into a `double`, and 2. into an `int`. In worst cases, this means also narrowing, which might be unwanted. Set the compiler flag `-Wall`, and the compiler will at least spit a warning. This casting is called implicit cast, in contrast to an explicit type cast, to which we will return later.

### 4.4.3 The main Function

The `main` function is a special function and the entry point of any valid C/C++ program. The standard prescribes that it returns `int`. But an explicit `return` statement can be omitted – meaning implicitly `return 0;` (in POSIX terms: "Everything is fine! Function finished successfully.").

Mostly, up to now, we have used the function signature `int main()` without any function parameter. This is standard-conform. And if we don’t need the parameters, that’s perfectly fine.

The `main`'s function parameters but may have a sense. The mostly used `main` function signature is `int main(int, char**)`, often written as

```cpp
int main(int argc, char** argv)
```

or

```cpp
int main(int argc, char* argv[])
```

About the star, we’ll talk in the next subsection. But in short, it means `argv` is an array of an array of characters, or, in other words, an array of character-strings.
4.4. FUNCTIONS

These parameters can be used to get the command-line parameters of the program call. The following example prints out its command-line parameters.

```cpp
#include <iostream>
int main(int argc, char* argv[]) {
    for(int i=0; i<argc; ++i)
        std::cout << argv[i] << std::endl;
}
```

It compiles, and works:

```
g++ -o main main.cxx
./main
./main
3
```

argv[0] contains the name of the called program, argv[1] the first parameter, and so on.

**NB - for real hackers:** Even shorter than the for-loop is this.

```cpp
while(argc--)
    std::cout << *argv++ << std::endl;
```

**NB - for real hackers:** The most general main function signature a programmer can use is

```cpp
int main (int argc, char *argv[], char *envp[])
```

The last parameter envp is again an array of character-strings – this time containing the environment variables set in the environment from which the program was started.

4.4.4 Function Parameters – Call-by-Reference vs. Call-by-Value; References and Pointers

One can give parameter values to functions via call-by-value. This essentially means that the parameter value is copied into the function. Also argc in the main function is an independent variable with the value of the number of the command-line parameters copied in from outside. We can do with it whatever we like (increment, decrement, as you can see in the NB above) without the side effect of changing anything outside of the function.

Parameters can be but given to functions via call-by-reference. You will largely use it because of efficiency aspects, as becomes clear soon. Copying just a reference or a pointer is much cheaper than copying complex compound objects.

Furthermore, sometimes it is necessary that a function changes a parameter given to it. For instance, in the next chapter, we learn something about std::vector, and we might want to add something to a vector inside a function. This works only via call-by-reference.

Call-by-reference parameters are declared either by

```cpp
ret-type func_name (par-type *par); // pointer
```

or

```cpp
ret-type func_name (par-type &par); // reference
```

In short, a reference is a constant pointer, meaning a pointer the address it points to cannot be changed anymore during the run-time. In order to understand this, we need to introduce some pointer semantics. But we keep this short, because we will try to work mostly in value semantics (for good reasons, as you will learn). Let’s start.
40 CHAPTER 4. BASIC ELEMENTS OF C++

```
int* c = &a;  // pointer to int, assigned to address of a
*c = 3;      // a == 3 now
*c = &b;     // redirecting c to address of b
std::cout << "a=" << a << " b=" << b
            << " c=" << c << "\n";
int &d = b;
d = 7;
std::cout << "b=" << b << " d=" << d << "\n";
```

Output:

```
a=3 b=5 c=0x7fff3a4bdd60
b=7 d=7
```

**Spoiler Alarm: Pointer Logic!**  
a and b are integers. Humans can life with that loose description.  
The details are more complicated. a actually is just a readable alias for a memory address, from where  
the next four bytes (int) represent an integer value. This address, which a aliases, can be obtained  
via &a.

    c is a pointer to an integer variable – again, the details are more complicated: c is an alias for a  
memory address, the next eight bytes represent a memory address to another memory location, the  
next four bytes represent an integer value.  

Headache? It becomes worse! Initially c contains the address of a in line 6. The value of c is not  
fix – we can reassign it, as we did in line 7, where it has the address of b. c=0x7fff3a4bdd60 in the  
output is exactly this value (equally, you could also write &b). Insofar, a pointer is not different from  
a normal variable. Its value is only an address instead of a "normal" value.  

Fed up with pointers? I understand, and agree. Fortunately, you will rarely need this. Because  
there is an alternative – references.

**Next Level!**  
We also defined a reference to an int, d, in line 11, which points to b. We assign a new  
value to d – just as if it were a plain int – but also b changes. So, a reference is in effect a pointer,  
but it wraps itself into a kind of value formulation, which much more resembles the dealing with plain  
values. However to make d point to another address is not possible (const pointer).

Please forget the pointers (everything with a *), and consider the following example.

```
#include <iostream>
void do_smth(int & d) {
    d = 5;
}
int main(int argc, char* argv[]) {
    int a = 2;
    std::cout << "a=" << a << "\n";
    do_smth(a);
    std::cout << "a=" << a << "\n";
}
```

Output:

```
a=2
a=5
```

Do you get the point? In most cases, you will also rarely explicitly define references as such – only  
as function parameters. If it is required, thus, use references and & instead of pointers for function  
parameters!

---

5& is the address-of operator.
6Mnemonic: If c is int *, then *c is int.
**NB - A final Remark:** const correctness is mentioned again and again later in that course. You will see const references in the future – many of them. These are effectively const pointers to const objects. Often, you only want the advantage of not copying objects by value into a function, because this might be expensive in terms of speed and memory consumption. But you also don’t intend to change these objects inside the function. The way to ensure this is const & (const reference).

```cpp
void do_not_change_d(const int & d);
```

or

```cpp
void do_not_change_d(int const & d);
```

The const here says that d is a reference (const pointer) to an integer object that is considered unchangeable. For ints (and basically for all C++ built-in types) that’s sort of an overkill, and copy-by-value is simpler and has no overhead! But soon, you will learn about objects of more complex types. And then using references is essential to obtain any kind of performance!

**Exercise:** Experiment around with function parameters as values, references and pointers. Place also const into the reference and pointer types (for values it does rarely sense), and check what happens if you try to change such variables inside the function. Look onto the compiler error messages!

### 4.4.5 Lambda-Functions

Since C++ 11, there is another kind of function – Lambda functions (cppreference: Lambda Functions). They are sort of inline functions and have therefore usually no name. Their importance and practical relevance will become clear later in the STL algorithms chapter. Their syntax is a construct that consists of three sections, \[]()\} – in that fixed sequence. The braces {\ldots} are the function body. The parentheses (...) will hold the function parameters. The square brackets [...] represent the so-called capture clause. Its usage will become more clear when we introduce the STL algorithms.

As example, let’s use the square-root function from above, and write it as Lambda function.

```cpp
#include <iostream>
int main() {
  std::cout << " sqrt(2.) = " << [](double a){
    double x=0.5*a;
    for(int i=0; i<10; ++i)
      x=0.5*(x+a/x);
    return x;
  }(2.) << "\n";
}
```

It outlines only the principle! Sure! That’s far worse to read than normal function definitions. Probably, square-root is not the best example. Better examples will put in place later.

We can also name it if we want to reuse it – and use it as normal function as before.

```cpp
#include <iostream>
int main() {
  auto f = [](double a){
    double x=0.5*a;
    for(int i=0; i<10; ++i)
      x=0.5*(x+a/x);
    return x;
  };
  std::cout << " sqrt(2.) = " << f(2.) << "\n";
}
```

Two things are remarkable here. First, auto represents a *automatically deduced* type. In this case, the type of f is a function with one double parameter and a double return value. We can use auto as sort of a dummy type place-holder, when it is not relevant for us to write the type explicitly. Examples will come up in later chapters. Second, we defined here a function locally within a scope (not a namespace)! f is not a global function! With normal (not Lambda) functions, that is not possible.
Concerning the parameter list and the function body, the same rules apply as for the normal named functions above. But the capture clause, [ ], is special. In that clause, you can “capture” parameters from the outer scope – but not as function parameters! As example, consider

```c++
#include <cmath>
#include <iostream>

int main() {
    double mu=0., sigma=1.;
    auto f = [&mu,&sigma](double x){
        return exp(-0.5*(x-mu)*(x-mu)/(sigma*sigma))/(sqrt(2.*M_PI)*sigma);
    };
    std::cout << "Gaus(0.2,mu=0.,sigma=1.) = " << f(0.2) << "\n";
    mu=1.;
    std::cout << "Gaus(0.2,mu=1.,sigma=1.) = " << f(0.2) << "\n";
}
```

This gives the Lambda functions kind of a stateful character. This is a very powerful feature, as we will see, and can be used even more flexibly as outlined here. The & before mu and sigma indicates that we use references here. So, you could also change the captured parameters permanently, meaning that the variables in the scope outside the Lambda function change permanently.

**Exercise:** Use the above code, compile and run it! Next, play around with the capture clause parameters – using & and without. Try to change the captured parameters inside the Lambda function’s body!

---

7 & can be left away if you just want to use the value inside. Or, you can use const references.
Chapter 5

Adding Subfolders and Files to the Project

Up to now, we have considered only a single source code file (main.cxx) in the project. However, more realistic projects usually consist of several to many files, possibly in several different subfolders for a better comprehensible project structure. We would like to simulate this now starting from a clone of the Unit Test example from above.

Git Branching

$ git clone 02 04
$ cd 04
$ mkdir MySqrt

We want to introduce another feature of Git – branching. Branches represent parallel development lines of a project.

$ git branch devel
$ git show-branch
  ! [devel] added .gitignore, added sqrt/test to main.cxx
  * [master] added .gitignore, added sqrt/test to main.cxx
--
++ [devel] added .gitignore, added sqrt/test to main.cxx

$ git checkout devel
$ git show-branch
  * [devel] added .gitignore, added sqrt/test to main.cxx
  ! [master] added .gitignore, added sqrt/test to main.cxx
--
++ [devel] added .gitignore, added sqrt/test to main.cxx

One can have several branches at a time. And you can switch back to "master", or any other branch, by git checkout <branch name>.

We now create a new folder and new files in the "devel" branch. They will not be visible when switching to the "master" (or any other) branch.

$ mkdir MySqrt
$ touch MySqrt/mysqrt.h MySqrt/mysqrt.cxx MySqrt/CMakeLists.txt
CHAPTER 5. ADDING SUBFOLDERS AND FILES TO THE PROJECT

```bash
$ git add MySqrt
$ git commit -m "added MySqrt"
```

The new folder `MySqrt`, and its three files are now part of our Git repository – in the "devel" branch. When you switch to "master", the complete `MySqrt` folder has vanished. Switching back to "devel", the folder reappears.

Adding the Library – several Source Files

Now, let us fill the files. `MySqrt/mysqrt.h` is a so-called header file. It is similar to `iostream` above. Its purpose is to declare functions consistently among different (independent) translation units. Those are `main.cxx` and `MySqrt/mysqrt.cxx`, in the latter of which we define the function declared in `MySqrt/mysqrt.h`.

`MySqrt/mysqrt.h` looks as follows.

```c
#ifndef MYSQRT_
#define MYSQRT_
double mysqrt(double a);
#endif
```

#ifndef, #define, and #endif are pre-processor directives. In the compiler’s first steps, include-files are included verbatim into the translation units. These pre-processor directives are used here to prevent double (recursive) inclusion. Headers should always possess such include guards! The name `MYSQRT_` is rather arbitrary. But it should be sufficiently unique. Also often used is `MYSQRT_H`.

The whole construct reads as follows. If `MYSQRT_` is not defined, yet, define it and include the rest of the file. If `MYSQRT_` is already defined when processing the first line, ignore everything of that header file until #endif.

`MySqrt/mysqrt.cxx` takes the following shape.

```c
#include "mysqrt.h"
double mysqrt(double a) {
  double x = 0.5*a;
  for (int i=0; i<10; ++i)
    x = 0.5*(x+a/x);
  return x;
}
```

The `main.cxx` file simplifies somewhat (as intended!).

```c
#include "mysqrt.h"
#include <cassert>
#include <cmath>
void test () {
  assert(1.0 == mysqrt(1.0));
  assert(fabs(1.41421356237309504880-mysqrt(2.0))<1.e-10);
}
int main () {
  test ();
}
```

Before we go on and compile, some explanations should be put forward here. You may wonder why some headers are included in angular brackets `<...>`, and others in quotes "...". This is reasoned historically. The quoted variant searches the headers in the current directory first. The angular bracket variant searches the header first in system folders first. This is not as important anymore as we can tell the compiler, which folders to search through for headers. Here, only the order of folders plays a role. – It is a good idea to name the header files still uniquely, if possible.

Modifying CMakeLists.txt Files

Before trying to compile, we need to adapt the `CMakeLists.txt` files. You have seen that there is now a top-level `CMakeLists.txt` file, and one inside the `MySqrt` folder. These files represent a hierarchy
"cmake" later passes through accordingly. But it needs to know about that. So, let’s adapt the top-level *CMakeLists.txt* first:

```cmake
# Top-level CMakeLists.txt

# Set the CMake version
cmake_minimum_required (VERSION 3.5)
# Set the project name
project (first)
# Add the MySqrt subdirectory
add_subdirectory(MySqrt)
# Add the executable target
add_executable(first main.cxx)
# Link the MySqrt library
target_link_libraries (first PRIVATE MySqrt)
```

*add_subdirectory* adds the *MySqrt* folder to *cmake*’s path tree, and adds thus the target (and library) *MySqrt*, which is then linked into our executable via *target_link_libraries*. *CMake* really simplifies things for programmers!

Finally, *MySqrt/CMakeLists.txt* just adds this library:

```cmake
# MySqrt/CMakeLists.txt

# Add the MySqrt library
add_library(MySqrt mysqrt.cxx)
# Include directories for MySqrt
target_include_directories(MySqrt INTERFACE ${CMAKE_CURRENT_SOURCE_DIR})
```

The last line, *target_include_directories* just familiarizes all depending target where to find the header file (*mysqrt.h* in this case).

We can now configure and build the project. In the top-level of the project:

```bash
$ cmake -B build
$ cmake --build build
```

Scanning dependencies of target MySqrt

Building CXX object MySqrt/CMakeFiles/MySqrt.dir/mysqrt.cxx.o

Linking CXX static library libMySqrt.a

Built target MySqrt

Scanning dependencies of target first

Building CXX object CMakeFiles/first.dir/main.cxx.o

Linking CXX executable first

Built target first

> build/first

A library called *libMySqrt.a* (that’s a so-called static library) is built first. ("cmake --build build --target clean" (make clean) and "cmake --build build --verbose" ("make VERBOSE=1") can provide you more details.) You can investigate the *build* directory, what *CMake* does and creates in the *build* directory.

**Exercise:** Include *iostream* inside the *MySqrt/mysqrt.cxx* and in *main.cxx*, and print out some information to the screen (making the example above more interesting) – e.g. which function is just executed! Recompile (but do not reconfigure), and execute!

**Merging the Branch back to ”master”**

Finally, we want to commit this change – as it appears successful. Go to the project’s top-level folder, and issue

```bash
$ git commit -am "added MySqrt library implementation"
$ git log
```

```
commit 5161549416cdacbc386e6ffe1cb20c2dc33c7495
Author: Martin <ohlerich@lrz.de>
Date:   Sun Jun 23 12:03:07 2019 +0200

added MySqrt library implementation
```
Author: Martin <ohlerich@lrz.de>
Date: Sun Jun 23 11:22:37 2019 +0200

added MySqrt
...

We also want to bring back this new developed branch to the master branch. So, issue

```bash
$ git checkout master
$ git merge devel
Update d40adec..5161549
Fast-forward
  CMakeLists.txt | 6 ++++++
  MySqrt/CMakeLists.txt | 1 +
  MySqrt/mysqrt.cxx   | 8 ++++++++  
  MySqrt/mysqrt.h     | 7 +++++++
  main.cxx             | 9 +--------

5 files changed, 23 insertions(+), 8 deletions(-)
create mode 100644 MySqrt/CMakeLists.txt
create mode 100644 MySqrt/mysqrt.cxx
create mode 100644 MySqrt/mysqrt.h
```

```
$ git log
commit 5161549416cdacbc386e6ffe1cb20cdc3c3db7495
Author: Martin <ohlerich@lrz.de>
Date: Sun Jun 23 12:03:07 2019 +0200

added MySqrt library implementation
```

"master" now contains all developments done in "devel". Also the commit history.

**Why this fuzz about it?** Well, you can do all your developments in "devel"! And if successful, you can merge it to "master". In this way, "master" can always build and is functioning – up to the latest merges.

Furthermore, when you work together with other developers, you can always branch off from the latest definitely working "master", commit your changes, and merge them back. Each developer in your project can do so. In that way, you do not much interfere with the other developers if you messed up.

There is, of course, much more about merging! It might happen that different merge commits of different developers lead to conflicts. That is okay! Git was made for that case, too. But the merger (i.e. the developer who tries to merge) is in charge of resolving this conflict, possibly manually unless Git can do this automatically. We refer here to text books [19] and to Google, if you need to learn more about this scenario, because it is out of scope of this introductory course. For a single developer and single branch-offs merge conflicts should not occur.
Chapter 6

Introduction to Vector, String, Algorithms, and I/O – STL Basics

We introduce here some of the most essential higher-level abstraction concepts of the C++ Standard (Template) Library (STL) [27, 16], as those will be ubiquitous. Using them guarantees in most cases efficient code which is easy to comprehend and to maintain.

One important fact is still to mention. None of the C++ Standard Library objects are built-in, i.e. none are language features – even not the I/O. Therefore, one must include corresponding header files.

6.1 Basic I/O

The I/O library of C++ is rather large and complicated – but usually easy to use. ([cppreference: Input/Output library]

6.1.1 Standard Output

Normal output to a terminal via `std::cout` could be seen in action already. The operator `<<"directs"` any printable object (string literals in double quotes, number literals, build-in variables, and – as we’ll see soon – any object, where a `<<` operator is defined (overloaded), such as strings). The corresponding header file is `iostream` (see [cppreference: iostream]

```cpp
#include <iostream>
int main() {
    char c = 'a';
    int i = 4;
    double d = 2.3;
    std::cout << "Text " << 0.2 << " " << 3 << " " << 'c'
        << "\n" << c << " " << i << " " << d << "\n";
}
```

Output:

```
Text 0.2 3 c
a 4 2.3
```

As is visible from the example above, one can concatenate several output terms via `<<`, and even line breaks to not disturb.

There are lots of extensions for formatting. For instance, floating point numbers can be represented in different forms. Also, a `bool` will normally be printed as 0 or 1. But sometimes, you would like to see really true and false in the output. Please, try the following.

```cpp
#include <iostream>
int main() {
    // ...
}
```
double $d = 3.1415$;
std::cout << std::scientific << $d$ << std::endl;
std::cout << std::fixed << $d$ << std::endl;

bool $b = true$
std::cout << std::boolalpha << $b$ << std::endl;
std::cout << std::noboolalpha << $b$ << std::endl;

The corresponding header is \texttt{ios}. But this header is already included by \texttt{iostream}.

Exercise: Consult \texttt{cppreference: ios} for further stream modifying functions. Rewrite the above program to align the output (left/right in a fixed width column), change the numerical base for number representation, and transform text (double quoted character strings) to lower/upper case.

Another extension are stream manipulators from header \texttt{iomanip} (\texttt{cppreference: iomanip}). As example, compile and run the following.

```cpp
#include <iostream>
#include <iomanip>

int main() {
  double $d = 3.1415$;
  std::cout.width(10);
  std::cout << $d$ << std::endl;
  std::cout << std::setw(10) << $d$ << std::endl;
  std::cout << std::setprecision(10)
        << std::scientific << $d$ << std::endl;
}
```

That way, you can change the output width or the precision of the floating point numbers.

Exercise: Print out in a \texttt{for}-loop abscessa-ordinate-values ($x, f(x)$) of some function. Select a convenient column width and a number representation. You can redirect the program's output to a file via

```
$ ./main > data.txt
```

and view it via a pager (e.g. \texttt{cat data.txt}), or via Gnuplot\cite{gnuplot} (\texttt{gnuplot> plot 'data.txt' u 1:2 w l}).

6.1.2 Standard Input

\texttt{iostream} also contains \texttt{std::cin} – the input stream from keyboard during the program run-time. Its use is similarly simple as the output. But instead of <<, one needs to use the >> operator. That is, optically the input stream is directed “inward”. The target the stream points to must be a variable. Consider the following example.

```cpp
#include <iostream>

int main() {
  int $a$;
  double $b$;
  std::cout << "An integer, please: ";
  std::cin >> $a$;
  if(!std::cin) {
    std::cout << "Oops! \n";
    return 0;
  }
  std::cout << "A double, please: ";
  std::cin >> $b$;
  std::cout << $a$ << "$b" << "\n";
}
```

Some warning here: The input should be checked for errors, because there are erroneous users that supply this input. This is possible by checking the input stream (line 7).

One can use it like this:
6.1. **BASIC I/O**

```bash
$ ./test
An integer, please: 2
A double, please: 2.3
```

**In POSIX compliant shell environments,** one can use the shell piping mechanism to fake sort of a file input.

```bash
$ cat data
2
2.4
$ ./test < data
An integer, please: A double, please: 2 2.4
```

Or, without a file

```bash
$ echo -e "2\n2.3" | ./test
An integer, please: A double, please: 2 2.3
```

**Exercise:** We will later learn the `std::string` object. Without knowing details, use the following code template

```cpp
#include <iostream>
#include <string>
int main() {
    std::string s;
    ...
}
```

Use the same input operators `>>` from `std::cin` to read in text from the terminal standard input into the string. And also print it to the screen.\(^1\)

### 6.1.3 File I/O

Writing to and reading from files is similarly simple. All you need is the `fstream` header (cppreference:: fstream).

**Writing to a file.** The following example writes out the numbers 0 through 4 into the ASCII file `text.txt`, together with a function value of \(x + 0.5\).

```cpp
#include <fstream>
int main() {
    std::ofstream ofs("text.txt");
    for (int i=0; i<5; ++i)
        ofs << i << " " << i+0.5 << "\n";
    // Usually closing file here?
}
```

While `std::cout` is already instantiated, an `ofstream` must be opened (and closed again) "by hand". But else, a file output stream behaves the same as `std::cout`. Generally, this can be combined with `ios` and `iomanip` as for `std::cout`. This is the advantage of abstraction concepts – all output streams should behave equally.

There is something striking here. We did not close the file again. Earlier, closing files and releasing file handles was not only a good idea, but mandatory. In C++, (value-semantics, RAII (Resource Acquisition Is Initialization)), a "owner" is responsible for a resource. In this case, the `std::ofstream ofs`

---

\(^1\)Use intuitively the same syntax as for `int` or `double` variables!
is responsible for the file handle. And this variable is "destructed" when its scope is left – that is at the main function’s closing brace, "}". How that is accomplished will be shown in the coming parts of this course.

You can close a file also manually – and that’s sometimes necessary.

```cpp
ofs.close();
```

Reading from a file is also straightforward. The following example reads from a file the name of which is given as the first program command line parameter.

```cpp
#include <iostream>
#include <fstream>

int main(int argc, char* argv[]) {
    std::ifstream ifs(argv[1]);
    if(!ifs) return -1; // Error Handling
    int a; double b;
    while(ifs >> a >> b)
        std::cout << a << " 
" << b << std::endl;
    ifs.close();
}
```

If the file cannot be opened successfully, the program stops. Otherwise, we read in one integer and one floating point (double) number at a time, and print to the screen. If this fails, the loop is left. Finally, we close the input stream manually.

### 6.2 Strings

When we talk about strings, we mean *character strings* – words, sentences, etc. Strings are rather complicated objects in their implementation. But they are designed to be easy-to-use. In order to use strings, you need to include the header `string` (see `cppreference: std::basic_string`)

#### Creation of Strings

First of let us look at the creation of a string. C++ is object-oriented. And strings are objects. Objects can be created in different ways.

```cpp
#include <iostream>
#include <string>

int main() {
    using std::string;
    using std::cout;
    // native C char array; deprecated: don’t use!
    char *cs = "Hello";
    // assignment, conversion from literal
    string s1 = "Hello";
    // assignment, conversion from char array
    string s2 = cs;
    // constructor from literal
    string s3 ("Hello");
    // constructor from char array
    string s4 (cs);
    // copy constructor
    string s5 (s1);
    // copy assignment
    string s6 = s1;
    // default constructor, empty string
    string s7 ();// or std::string s7;
    // assignment
    s7 = s1;
    cout << cs << "\n";
    cout << s1 << "\n";
```

\(^2\)Don’t be scared that you see `std::basic_string`! `std::string` is an alias to it.
Strings can be created in different ways – from literals, from char arrays, from other strings. Mostly, it is done the same way as you used to create any other built-in variable. New is the constructor nomenclature using braces, {...}. Before C++ 11, the constructor calls were performed using parentheses, (...) (that is still feasible). But writing a default construction resulted occasionally in surprises.

```
std::string s ();
```

Oops! That is a signature for a function named s, taking no parameter, and returning a string. Therefore, using braces is considered more fail-safe in this respect.

Another nice feature of strings is that the programmer actually does not need to care for any memory management. While char arrays are really raw memory pieces interpreted as characters, which have a fixed length (either statically created such as from literals, or, dynamically during run-time), and for which the programmer needs to care to handle the memory resources (allocation, avoiding out-of-range access, release, etc.), for std::string that’s mostly not necessary. A string contains a raw char array (dynamically managed for most cases). And if you really need this raw array, you can obtain it via

```
std::string s {"data.txt"};
const char * c = s.data(); // C string
const char * c = s.data(); // C string with trailing '\0'
```

But otherwise, you will never be concerned with the internals of std::string.

Character-wise Access

Arrays can be accessed via the index operator [int], as we have shown above e.g. when explaining for-loops. For a character array, this is true, too. As a string is a character array, we would like to use [int], too. And so, std::string can be used with an index operator! But string objects also contain meta-information about e.g. the length of the char array they contain. Therefore, there is a way to do bound-checks.

```
std::string s {"Hello!"};
char c = s[3]; // c == 'l', no check
// c = s[4]; // c == 'o', bound check
// std::cout << s[10] << "\n"; // Hmm! Maybe no crash!
// std::cout << s[10] << "\n"; // Oops! Out of range!
// s[4] = 'i'; // character-wise assignment
```

Bound checking all the time is rather expensive – and in most cases not necessary. Therefore, the index operator [...] was implemented not to do these checks. Thus, the semantics of that operator and its speed are comparable of that native array indexing. Beware!! String-indexes start at 0 (as in native-C)!

There is a so-called class member function, at(int), which does the same as [int], except that bound checking is performed. An out-of-range errors are generated during run-time, if an index is out of range. No such error is generated with [int]! So, you might be lucky that your program does not crash when committing this crime. But you are in the realm of undefined behavior!

Generally, calling member functions is accomplished by adding a dot, ".", the name of the member function, and the parameter list embraced by parentheses, (...) (like for functions).

For beginners, this concept of functions "assigned" or "attached" to objects, is often new and difficult to comprehend. In order to understand what these functions do on the object, they got mostly telling names. And documentation is important in order to know how to use these functions. The best platform for STL we know of is CppReference. Use it extensively!

---

3This is true for value and reference objects. Pointers need to be handled differently!
Concatenation

As for Python, you can also concatenate several strings into one string, by means of an operator +, which is semantically associated to addition.

```cpp
std::string s1 = "Hello";
std::string s2 = s1 + ", World!";
s2 += " Bye!";
```

This comes in quite handy if you need to construct valid absolute path names, for instance.

Comparison

Often, you need to compare also strings for equality, non-equality, larger-ness or smaller-ness than (lexicographical). Intuitively, this goes like this:

```cpp
std::string foo = "alpha";
std::string bar = "beta";
if (foo==bar) std::cout << "foo == bar\n";  // false
if (foo!=bar) std::cout << "foo != bar\n";  // true
if (foo< bar) std::cout << "foo < bar\n";  // true
if (foo> bar) std::cout << "foo > bar\n";  // false
if (foo<=bar) std::cout << "foo <= bar\n";  // true
if (foo>=bar) std::cout << "foo >= bar\n";  // false
```

Substring Extraction and Manipulation

The opposite of concatenation is substring extraction. This can be accomplished by means of `substr()`.

```cpp
const std::string s = "This is a string";
std::string s1 = s.substr(10);   // s1 == "string"
std::string s2 = s.substr(5,2); // s2 == "is"
```

This creates a new string containing the substring (the original is not changed). `substr()` has two parameters. The first is the position in the string, from where to start the substring. The second is the number of following characters to be placed in the substring. If omitted, the rest of the string to its end is taken.

Another case is to remove a part of a string in-place. This is done via `erase()`. For example,

```cpp
std::string s = "This is a string";
s.erase(9);  // s == "This is a"
s.erase(4,3); // s == "This a"
```

A string that shall be modifiable in that way must not be constant, of course. Similar to `substr()`, `erase()` can take two parameters – the position from where to erase, and the length of the substring to be erased. If the latter is omitted, again the rest of the original string up to the end is removed. `erase()` changes the original string!

Strings can also behave semantically like a `std::vector` – a container/array of characters – (see next section). Therefore, member functions like `push_back()`, `clear()`, `append()`, `replace()` and `resize()` can be used to append, extend, clear out, or resize the string, which changes the string. We will discuss this later.

Searches

Another operation with strings is to check whether a certain substring is contained. `std::string` has several search functions. The simplest of which is `find`, which accomplishes a forward search.

```cpp
std::string const s = "This is a string";
std::string::size_type n = s.find("is");
if (n != std::string::npos)
    std::cout << s.substr(n) << std::endl;
```
6.2. STRINGS

std::string::size_type is an integer type (most probably a long). If you do not like those type names, use using declarations to abbreviate.

```cpp
using sztype = std::string::size_type;
...
```

Or, if the type is unimportant, you can use auto (type deduction).

```cpp
std::string const s = "This is a string";
auto n = s.find("is");
if (n != std::string::npos)
    std::cout << s.substr(n) << std::endl;
```

Depending on what you actually need, or what your preferences are, you can write this in an even more compact way.

```cpp
std::string const s = "This is a string";
if (s.find("is") != std::string::npos)
    std::cout << s.substr(s.find("is")) << std::endl;
```

Here, we have of course the disadvantage that we search twice.

There are more search member functions of std::string, that do different kind of searches. E.g. rfind() searches for substring occurrences from the string's back – reverse search. find_first_of(), find_first_not_of(), find_last_of(), and find_last_not_of() search for first occurrences of a character out of a specified character sequence – or not-occurrence. Both, forward and reverse. They all return the position in case of a match, or std::string::npos (consider npos simply as a "invalid position"!).

Conversions

How to make a number – e.g. an int or a double – a string? This might be desirable e.g. for nice formatting of numbers (e.g. replace decimal "." by German decimal comma ",", or similar). In principle, you can convert almost every built-in type into a string.

```cpp
int i = 1;
double d = 2.71;
std::string si = std::to_string(i);
std::string sd = std::to_string(d);
```

The opposite is to convert a string into a numbers. For instance, you read in number values as character strings from the command-line, and you want to convert them to numbers used inside your program.

```cpp
std::string s = "2.71";
double e = std::stod(s);
s = "e = 2.71!";
e = std::stod(s[4]);
e = std::stod(s.substr(s.find("=")+1));
```

There is almost for each type such a conversion function. But they have some limits. We therefore recommend not to use these rather old-fashion conversion tools, and better stick to so-called string-streams, which we introduce later in this course.

Exercise: Write a program that reads the name of the user, his/her age and body height from the terminal standard input or a file. Also, let the program decide (based on input parameters), whether the input is via std::cin or from file. Consider input error checking, and re-request the information if an error occurred! Print the information in nice text format to the screen!

Exercise: By means of strings, it is easy now to handle command-line parameters of a program.

```cpp
#include <iostream>
#include <string>
int main(int argc, char * argv[]) {
    using std::cout;
```
using std::string;
for(int i=0; i<argc; ++i) {
    string cmd(argv[i]);
    cout << " parameter " << i << " is " "
    << cmd << " the length of " << cmd.length() << "\n"
    if(cmd.find("-") == 0)
        cout << " Seems to be a switch! " "\n"
    
} 

Extend this code to read in flags (character or string with leading dash(s), but no values – e.g. -h or --help for printing a help message), dedicated parameters (with leading dash(s), and comma or = separated values, e.g. --name="Johnny Cash" or --name "Johnny Cash"), and unspecified parameters (like a list of files, e.g. *.txt), which are sorted and printed to the screen. To distinguish them on the command-line, it makes sense to have the unspecified parameters trailing to the flags and the dedicated parameters. But can you conceive a way where the order of the command-line parameters is more or less unimportant?

Exercise: A simple request-driven program with a loop can be realized by std::getline.

```cpp
#include <iostream>
#include <string>
int main() {
    std::string line = "";
    do {
        if(line.length()>0)
            std::cout << " You entered: " "
        << line << "\n"
        std::cout << " Enter sthg (or 'Ctrl+D'):
";
        while(std::getline(std::cin,line));
        std::cout << "\n"
    } 
    
Check its syntax at cppreference: getline. You can use std::getline also for file streams. Modify the above program to read lines from an ASCII text file, the name of which you specified as command-line parameter of the program.

6.3 Vector

A vector – std::vector (see cppreference: std::vector) – is one of several container classes in the STL. It is useful and convenient for handling larger amounts of data, of which you but don’t know how many entries there might be. For instance, when you read data from a file, you usually don’t know in advance how many data are stored there. But there is also a very basic use of vectors (and other containers): Imagine you want to give an array of data to a function. With native arrays, the information of the length of this array is lost, because you can submit a native array only as pointer (call-by-reference). Furthermore, you immediately face the problem of the memory comprising this array. This is more pressing if you need to change the size of the array because e.g. more data will be added later. To avoid all these issues of dynamic run-time memory management, std::vector was designed. And more advantages become obvious, e.g. when dealing with algorithms.

6.3.1 Creation and Handling

The vector container type has a somewhat strange new syntax element, the sense of which becomes clearer in a moment. So, let us look at a simple example of usage.

---

4 Of course, this problem can be solved by file meta data, stored insight the file together with the data. Or you can check the file’s content first, to calculate how many data entries there are – this requires the files to be formatted in a special way.

5 Correct! Designed!!
This code initializes the two vectors $vi1$ (with integer numbers 2,5,4,7,9,2) and $vi2$ (empty). In the loop all values of $vi1$ are copied to $vi2$. The design of std::vector was made such that this above code appears intuitively comprehensible to the programmers, although the back-end implementation of vector is rather complicated. Here, for instance, you probably never thought about how large $vi2$ was after its initialization, did you?

Let us start at the beginning. First, we need to include the vector header to make sure that the compiler knows what std::vector is, and which member functions are available.

Second, we declare and define vector containers in a similar fashion as we did with built-in type variables, and as we did with std::string (remember: std::string is to some extent a std::vector<char>). {} means that the vector is initialized as empty. \{2,5,4,7,9,2\} is just a so-called direct list initialization, which we know also from classical arrays.

Third, at the type specification, we must put a type in angular brackets, <int>. This tells the compiler, that we wish a vector of integers.

Fourth, a default-constructed vector ({} or no braces) is an empty vector. Its size is zero, but its capacity does not need to be zero.

As for string, you can access the different vector element by means of the index operator [int] (or at(int)). And as for strings, there is no bound check when using this index operator (different from at(int)).

If you want to fill something into the vector, you have different options. You can initialize the vector using a initializing list. You can successively fill the vector from the rear by means of push_back(value). When doing this, you don’t need to care for the memory (except you have requirements beyond the computer’s capability). std::vector automatically manages the memory. Another possibility are algorithms, which are introduced later.

You can also check whether a vector is empty, using the member function empty(). If you know the size of a vector in advance, you can use reserve(int) to set the vector’s capacity. You can also clear out a vector, using clear. std::vector also allows insertion, erasure, and more of the manipulation procedures which we already saw for strings.

Finally, as for strings, there also some operators (assignment, comparison). For instance,

```cpp
#include <vector>
int main() {
    using ivec = std::vector<int>; // using type alias
    ivec vi1 {2,5,4,7,9,2};
    ivec vi2 {vi1}; // copy construction
    ivec vi3 = {vi1.begin(),vi1.end()}; // range constructor
    ivec vi4 = vi1; // copy assignment
}
```

The range constructor can become very handy when you need only a sub-range of another vector, or read in from files. Assigning a vector to another is yet another possibility to fill it. The example from above can be largely simplified.

```cpp
#include <vector>
int main() {
    std::vector<int> vi1 {2,5,4,7,9,2}, vi2 {};
    vi2 = vi1;
}
```

---

6 This is called “template instantiation”. As short background, std::vector is a class template. That means that it is written generically, and type independent. We will see later that we can create vectors of all possible types. Templates are introduced here purely from the users perspective. The details will be explained much later in the course. But note that different types in template instantiations create different types. Therefore, std::vector<int> and std::vector<double> are really two different types, and conversions between both is not possible (by default)!

7 Capacity == size of internal memory already acquired (allocated). Size == size of effectively used memory. $\text{Size} \leq \text{Capacity}$!
Vectors as Function Parameters

Sometime, you wish to give a vector to a function. We show here two ways. First, call-by-value.8

```cpp
#include <iostream>
#include <vector>
using ivec = std::vector<int>;
void print(ivec v) {
    v[1] = 10;
    v.push_back(213);
    std::cout << "Element 1 is " << v[1]
               << " size of v is " << v.size() << "\n";
}
int main() {
    ivec vi1 {2,5,4,7,9,2};
    print(vi1);
    std::cout << "Element 1 is " << vi1[1]
               << " size of vi1 is " << vi1.size() << "\n";
}
```

The full vector is copied into the function. Changes on the vector made inside of the function stay local in the function. Outside, after the function finishes, the outer vector is still the same as before. Contrast this with the following call-by-reference (but no pointer!).

```cpp
#include <iostream>
#include <vector>
using ivec = std::vector<int>;
void print(ivec& v) {
    // <-- here happened the change!!!
    v[1] = 10;
    v.push_back(213);
    std::cout << "Element 1 is " << v[1]
               << " size of v is " << v.size() << "\n";
}
int main() {
    ivec vi1 {2,5,4,7,9,2};
    print(vi1);
    std::cout << "Element 1 is " << vi1[1]
               << " size of vi1 is " << vi1.size() << "\n";
}
```

We only added a reference sign, &, to the function's vector parameter. A small change with a big effect. Now, all changes made inside the function are persistent also outside. The big advantage in any case is that the vector does not need to be copied into the function – only the reference to it is copied.

You are afraid that a function changes your vector? No problem. Write `void print(ivec const& v)` or `void print(const ivec& v)`!

**Exercise:** Change the function signature from `void print(ivec& v)` to `void print(ivec const& v)` or `void print(const ivec& v)`, and try to compile!

Call-by-const reference, is kind of an idiom you simply must learn! Use it whenever possible and useful, as it makes up most of C++'s performance.

### 6.3.2 Loops over a Vector

Quite natural for vectors is that you loop over them. All normal "basic" loops from above – for, while, do-while – with integer indexing, can be used. For instance,

```cpp
#include <iostream>
```

8Try this with native arrays!! ;)}
```cpp
#include <string>
#include <vector>
using svec = std::vector<std::string>; // gets really handy, right?
int main() {
    svec sv {};
    sv.push_back("Text No 1");
    sv.push_back("Text No 2");
    sv.push_back("Text No 3");
    for(int i=0; i<sv.size(); ++i)
        sv[i].erase(5,3);
    for(int i=0; i<sv.size(); ++i)
        std::cout << " String " << i << " is now " << sv[i] << "\n";
}
```

However, the index is actually not really needed in the erase loop. C++ knows the concept of so-called **iterators**. We introduce them here as they are needed in part also for the algorithms. Consider the following code, and try to understand it.

```cpp
#include <iostream>
#include <string>
#include <vector>
using svec = std::vector<std::string>
int main() {
    svec sv {};
    sv.push_back("Text No 1");
    sv.push_back("Text No 2");
    sv.push_back("Text No 3");
    for(auto p = sv.begin(); p!=sv.end(); ++p)
        (*p).erase(5,3);
    for(int i=0; i<sv.size(); ++i)
        std::cout << " String " << i << " is now " << sv[i] << "\n";
}
```

Although that's ugly, it is valid and – up to recently – state-of-the-art C++. `sv.begin()` returns an iterator, `p`, pointing to the begin of the vector. `sv.end()` points to one place behind the last valid vector entry. Iterators can be incremented, `++p` or `p++`, and behave much like pointer. Therefore, we need to dereference them when accessing the member functions of the string the iterator points to. This is done by `*p` (parenthesis are necessary here because of the operator precedence, and `*` has a higher one than `*` as your compiler might confirm). Alternatively, you can also use `p->erase(5,3);`

Here we are back at pointer semantics ... But iterators are no pointers. Indeed, they complex user-defined objects, i.e. no built-in types. They only resemble pointer semantics, and are thus also more general than pointers.

**Modern C++ 11** There is another loop construct (see **cppreference: Range-based for loop**) that is a bit shorter, and much more convenient – although maybe even more mysterious than iterators.

```cpp
#include <iostream>
#include <string>
#include <vector>
using svec = std::vector<std::string>; // gets really handy, right?
int main() {
    svec sv {};
    sv.push_back("Text No 1");
    sv.push_back("Text No 2");
    sv.push_back("Text No 3");
    for(auto & p : sv)
        p.erase(5,3);
    for(int i=0; i<sv.size(); ++i)
        std::cout << " String " << i << " is now " << sv[i] << "\n";
}
```

`p` serves here as the temporary object – to be precise, the reference to an object in the vector. Therefore, `auto&`!

**Exercise:** Remove the reference &, re-compile and re-run! What do you observe?
Remark: Don’t be upset or frustrated if you do not understand iterators or the range-for-loops, yet. With the time of seeing them in action and using them, the understanding will come.

6.4 Algorithms – Introduction

Sorting an array according to some criterion, or finding an element in an array, other things are most often problems you have to solve on data arrays. The STL algorithm library (see cppreference: Algorithms) is providing help. Even better, this library is designed to work on any container. This is not by accident! Furthermore, by means of lambda expressions (and function objects that are subject later in this course), algorithms are extremely flexible and extensible!

Assume, you have a vector of strings, and you want to sort them. Here is how it works.

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using svec = std::vector<std::string>;

void print(svec const & sv) {
    std::cout << "--------------------------\n";
    for(auto & p : sv) std::cout << p << "\n";
}

int main() {
    svec sv {"Dale","Pluto","Chip"};
    print(sv);
    std::sort(sv.begin(),sv.end()); // <-- here is the actual work
    print(sv);
}
```

This appears rather simple and almost intuitive, right? All what is a bit disturbing are the begin and end iterators! But you see immediately that the concept works. Can we for instance do this also with a string? Sure! Why not! A string is in this respect just a std::vector<char>! And it supplies begin and end iterators!

Or, what is about reverse the vector?

```cpp
std::reverse(sv.begin(),sv.end());
```

Randomize? That is, change the order randomly.

```cpp
std::shuffle(sv.begin(),sv.end());
```

That’s rather straightforward – almost too easy, right?9

But algorithms are much more powerful. Imagine, you need to remove something from a vector, e.g. the first three elements (assuming there are at least as many!),

```cpp
std::remove(sv.begin(),sv.end(),"Dale");
```

Some words are here in order. std::remove does not really remove the elements. They are just placed to the end of the vector (potentially empty), while the rest is reshuffled to the front. But the size of the vector is still the same. If you really want to get rid of the elements you removed, you also need to erase them.

```cpp
sv.erase(std::remove(sv.begin(),sv.end(),"Dale"), sv.end());
```

This is the so-called erase-remove idiom. Now, your vector is really shorter (size is smaller).

Next Level - Reading from a File

Suppose, we have a file (ASCII) with either words (strings), or numbers (doubles), or whatsoever – data.txt. Simply white-space-separated. We want to read and write it to the screen. Here, we use a combination of std::vector range constructor, iterators, and algorithms together. (Don’t be afraid if you cannot understand it immediately!)

---

9Well, indeed it is a bit more complicated: see cppreference: std::shuffle!
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>

int main() {
    using dvec = std::vector<double>;
    using diiter = std::istream_iterator<double>;
    using doiter = std::ostream_iterator<double>;
    std::ifstream dfile("test.txt");
    dvec rv {(diiter(dfile)), diiter()};
    // range c'tor
    std::copy(rv.begin(), rv.end(), doiter(std::cout, "\n"));
}

Don’t be scared about the iterators! They are just types, and will be explained later in more details!

Effectively, there are only three lines. 1 - We open an input stream. 2 - We use the constructor of vector to read in from a start-iterator (here that of the file) to the end-iterator (end-of-file). This fills the vector. 3 - Finally, the vector is copied to std::out – an algorithm.

`std::copy` is usually used to copy one container into another. Therefore, it takes a begin and end iterator of the container FROM WHERE is copied, and just a start iterator (can point to any position other than the begin) of the container WHERE TO is copied. It is in the responsibility of the programmer that the second container is large enough!

Obviously, a output stream can be considered as a container – at least it behaves like such in that respect that characters can be “filled” into it. So we just misused the copy algorithm in order to ”copy” into the standard output (supplemented by a new-line after each element).

In this way, we can really fast read in data from a file – or, actually, from any stream we like (could be also network stream)! By using user-defined types, later, we can extend this to read in actually everything we like. Only the memory of the machine is the limit! And all this with only few lines of code – but to the prize of a high level of abstraction.

**Next Level - Tuning**

Maybe you want to remove all elements with a certain property. Or, you want to check whether there are elements with a certain property. How would you approach this issue?

Here is an example. Assume again, we have an array of strings, read in from a file. And, just for fun, imagine you want to remove all words starting with a letter a or A. How would you proceed? Again, the algorithms are of help. There is, for instance, `std::remove_if`. It requires a start and an end iterator, as we already know, but also a predicate – a function that returns `true`, if the condition is fulfilled. The simplest object to use here are lambda expressions. So, consider the following code snippet.

```cpp
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

using svec = std::vector<std::string>;
using siiter = std::istream_iterator<std::string>;

void print(svec const& sv) {
    std::cout << "--------------------------\n";
    for (auto& p : sv) std::cout << p << "\n";
}

int main() {
    std::ifstream sfile("test.txt");
    svec sv {(siiter(sfile)), siiter()};
    sv.erase(std::remove_if(sv.begin(), sv.end(),
        [](const std::string& s){
            return (s[0] == 'a' || s[0] == 'A');
        }),
    sv.end());
    print(sv);
}
This is already a very complex exercise. Nonetheless, by means of abstraction, we can accomplish this in effectively one line of code. This is probably not always possible. But you should strive for such solutions — i.e. strive to think in abstract concepts, and then look whether there is an algorithm in the STL matching this purpose. Please, have a special eye on `std::rotate`, `std::transform`, `std::generate`, `std::partition` and `std::stable_partition`, `std::stable_sort`, `std::all_of`, `std::none_of`.

You but might also rediscover `std::find` and its if-variants. You may ask: What about reverse search? Fortunately, vector and string have a reverse iterator. You only need to specify `rbegin()` and `rend()`, respectively, instead of `begin()` and `end()`.

But enough for the moment! Please continue now with the hands-on exercises!
Chapter 7

Hands-On Exercises for Part 1

In the following Hands-On Exercises, you will have the chance to apply what you have learned so far. We supply problems with a more application-like character. You will be guided. And we will also introduce more of the STL features, where it makes sense.

But you have the freedom to extend or change the tasks if you like. Or, if it makes sense, you can use your own issues for practicing C++, and acquire help, when needed.

7.1 Statistics

7.1.1 RNG – Random Number Generation

The generation and usage of (pseudo) random numbers is the basis of so-called Monte Carlo methods. To say the least: they are really versatile and powerful! We would like to use them here for simple statistics exercises – and random data are created artificially in lack of data from e.g. experimental measurements.

The STL supplies several features to create sequences of pseudo random numbers (see cppreference: Pseudo-random number generation). We just spice them up somewhat.

First, we need to include `random` header. Second, we create a random number generator (RNG) – here, it is a Mersenne Twister `std::mt19937`. Third, we create a distribution – here a real (`double`) valued uniform distribution on the interval `[0,1)`. Finally, in the `for`-loop, we generate new (and different) numbers randomly according to the uniform distribution.

```cpp
#include <iostream>
#include <random>
int main() {
    std::mt19937 e2(1234);
    std::uniform_real_distribution<double> uniform_dist(0,1);
    for(int i=0; i<10; ++i)
        std::cout << uniform_dist(e2) << " ";
    std::cout << "\n";
}
```

Well, randomness is "relative"! Because the seed (1234) is fix, to re-run the program would reproduce the same sequence of numbers! But that is essentially wanted! "True randomness"\(^1\) is not reproducible! Only statistically (via histograms e.g.), distributions can be compared. "True randomness" is but not needed for what follows.

**Task:** The STL provides several random distributions – continuous ones as well as discrete ones. Create random numbers from some of them, and write them in a column into a file!

---

\(^1\)If you don’t know what it is, don’t worry!

\(^2\)If it seriously should be definable!
CHAPTER 7. HANDS-ON EXERCISES FOR PART 1

Hint: You can use the following script to plot a file named \textit{data.dat}.

```bash
#!/usr/bin/gnuplot --persist
reset
n=100       # number of intervals
max=1.      # max value
min=0.      # min value
width=(max-min)/n # interval width
# function used to map a value to the intervals
hist(x,width)=width*floor(x/width)+width/2.0
set xrange [min:max]
set yrange [0:] set xtics min,(max-min)/5,max
set boxwidth width
set style fill solid 0.5      # fillstyle
set tics out nomirror
set xlabel "x"
set ylabel "Frequency"
plot "data.dat" u (hist($1,width)):1.0 smooth freq w boxes lc rgb"brown" notitle
```

Adapt \(n\), \(\text{min}\), and \(\text{max}\) as needed! Gnuplot is not part of this course. But you can find the Gnuplot User Guide on the web.

7.1.2 MCMC – Markov Chain Monte Carlo

STL does not supply all possible distributions that might be needed in a simulation. There are several methods to create other distributions from a uniform random number generator. The most general one is the Metropolis-Hastings Algorithm – based on Markov Chains (therefore the alternative name MCMC). It is even applicable on finite regions and in many dimensions without difficulties. It is even not necessary to know the normalization factor of the probability density function (pdf), in order that this algorithm works! This is the driving reason for applications in the field of Bayes Estimation methods.

We outline the algorithm in general, and specialize it later for an exercise in one dimension.

Assume, you have a (possibly not normalized) probability density function, \(f(r)\), where \(r\) is a vector of an \(n\)-dimensional Euclidean space.\(^3\) The algorithm now goes as follows.

1. Choose a valid start vector \(r_0\), i.e. it must lie with the range of possible values.
2. Dice a vector \(p\) using a transition probability \(T(r_n \rightarrow p)\).
3. Accept \(r_{n+1} = p\) as new position according to the probability
   \[
   R = \min \left( 1, \frac{f(p)T(p \rightarrow r_n)}{f(r_n)T(r_n \rightarrow p)} \right)
   \]
4. If \(p\) is rejected, then \(r_{n+1} = r_n.\)

The sequence of \(r_n\) \((n = 0, 1, 2, \ldots)\) is then distributed according to \(f(r)\). The transition probability function \(T\) is rather arbitrary – you can make a choice of your own. In the special case, that \(T\) is symmetric, i.e. \(T(p \rightarrow r) = T(r \rightarrow p)\), it cancels out.

Enough theory! An example! We would like to sample values from \(f(x) = xe^{-x}\).\(^5\) The algorithm to implement is now the following (called Gibbs Sampling due to the choice of \(T\)).

\(^3\)Don't try to imagine it!!
\(^4\)It is important to account this double counting to obtain the correct distribution \(f(r)\)!
\(^5\)Although there is certainly a direct method, let us use it to play with MCMC!
7.1. STATISTICS

1. \( x_0 = 1 \) (arbitrary, but ok).

2. Transition probability function:

\[
(x_n \rightarrow p) \sim \text{Gaus}(x_n - p, \mu = 0, \sigma = 1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(x_n - p)^2}.
\]

As \( \mu = 0 \), it is symmetric, \( T(x_n \rightarrow p) = T(p \rightarrow x_n) \).

3. Accept \( x_{n+1} = p \) as new position according to the probability

\[
R = \min \left( 1, \frac{f(p)}{f(x_n)} \right) = \min \left( 1, \frac{pe^{-p}}{x_ne^{-x_n}} \right)
\]

4. If \( p \) is rejected, then \( x_{n+1} = x_n \).

Remark: In the above algorithm, \( \sigma \) is still a free parameter. Setting it to 1 here is a possible, and good choice. But if you choose it differently too small or too large, then the steps \( x_n \rightarrow x_{n+1} \) will be stagnant (small or even zero). That is, your random process is not much advancing. And being in one region, the chance is high to stay there still after several to many steps. A good guidance can be to have a rejection rate of about 50\% – but in a range of 30\% to 70\% is certainly also acceptable. Simply count how often you try, and how often it is accepted!

Task: Realize this algorithm for the Gibbs sampler! Check the result again by looking on the histogram (it should look like \( f(x) \))!

Hint: For the Gaussian (transition), you can use \texttt{std::normal_distribution}. You throw randomly \( z \sim \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}} \). \( p \) is simply obtained by \( p = x_n + z \).\(^6\)

Hint: For the acceptance, use \( R \sim \texttt{std::uniform_real_distribution<double>} \) on \([0, 1)\), and accept if \( R < \frac{f(p)}{f(x_n)} \).

Hint: Maybe it makes sense to store \( f(x_n) \) for the next iteration (need to be changed only if a new step is accepted), in order to increase the program’s efficiency.

7.1.3 Mean, Standard Deviation, Median, Quantiles with Algorithms

Task: Use one of the random distributions (one of the generators) from above, and feed 100000 (double) numbers randomly into a \texttt{std::vector<double>}! By means of STL algorithm library, determine the mean, standard deviation, median, and some quantiles (say 5\%, 10\%, 25\%, 75\%, 90\%, 95\%).

Hint: If your RNG supplies a Gaussian, you can approximately check the correctness of your program, as the characteristics of a standard Gaussian can be found on Google.

For those not familiar with estimates of mean, \( \bar{x} \), standard deviation, \( s_x \), quantiles from random samples, \( \{ x_i \}_{i=1}^N \), some help here.

\[
\mu_x \approx \bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i , \quad \sigma^2_x \approx s_x^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2.
\]

A quantile is defined as that position on \( x \) axis, up to which the area under the density function is the percentile (of 100\%). For instance, the median is the 50\% quantile, because left of it 50\% of all samples should lie (the rest of course right of it). So, estimating quantiles is essentially counting.

\(^6\)If \( \sigma \neq 1 \), then \( p = x_n + \sigma z \) (simple scaling).
7.1.4 Integrals using MC Methods

We are looking for the value of integrals of the following form.

\[ I = \int_a^b f(x) \, dx = (b - a) \int_a^b f(x) \frac{dx}{b - a} = (b - a) \cdot E_{\text{unif}(a,b)}(f(x)) \]

and

\[ \int_a^b dx = b - a = 1. \]

That is, we can express the integral as the interval size times an expectation value of \( f(x) \) with respect to a uniform distribution of \( x \in [a,b) \). But expectation value we know already – that’s the arithmetic mean. Thus, using a Monte-Carlo method, one creates uniformly random numbers \( x_i, i = 1,\ldots,N \), in the interval \([a,b)\), and forms the following sum, which should approximate \( I \).

\[ I \approx (b - a) \frac{1}{N} \sum_{i=1}^{N} f(x_i). \]

\( N \) should be sufficiently large in order to gain substantial precision.

**Task:** Try this method on functions like \( \sin(x) \) on \([-3, 3]\) (\( \int_{-3}^{3} \sin(x) \, dx = 0 \)), and \( xe^{-x} \) on \([0, \infty)\) (\( \int_{0}^{\infty} xe^{-x} \, dx = \Gamma(2) = 1! \equiv 1! \))

**Hint:** The uncertainty could be estimated via the standard deviation estimation.

\[ s_f = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (f(x_i) - I)^2}. \]

I should be precise with a standard deviation of \( s/\sqrt{N} \) – i.e. the larger \( N \), the more accurate the value of the \( I \) estimate. This is an option to investigate whether it makes sense to continue (i.e. to increase the statistics) in order to obtain a more precise result. True accuracy is obtained anyway only for infinitely many samples!

The real advantage of MC integration becomes significant for integrals in five and more dimensions, where the curse of dimension kills any performance of deterministic methods.

**Remark:** Playing with the interval limits of function \( xe^{-x} \) reveals some shortcoming – the result for \( I \) fluctuates largely with the upper limit. And a upper limit seems necessary because infinity is bad to cover! This is related to the fact that the uniform distribution also samples a lot of \( x \) values at locations, where the function is already close to zero – and thus does not contribute reasonably. The above approximation for \( I \) is strict only in the limit \( N \to \infty \). And for finite \( N \), it is bad if the dominant regions are under-represented by the sampling \( x_i \). This is the reason why importance sampling techniques need to be involved.

**Task:** Repeat the integration of \( xe^{-x} \)! But this time, sample \( x_i \) from the exponential distribution\(^7\), \( e^{-x} \), and determine

\[ I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) e^{-x_i} = \frac{1}{N} \sum_{i=1}^{N} x_i. \]

What do you find?

\(^7\)std::exponential_distribution<double>
7.1.5 Gambling

Throwing a Die

The numbers the throwing of a die results in is a random sequence of the numbers 1 through 6 ... each with a probability of 1/6.

**Task:** Imagine, you are a Game programmer, and you are in charge to program the RNG for this dicing! The user shall see a new (not predictable) number from such a dicing event on each pressing the enter-key.

Random Shuffling (Permutations)

This time, we have a sequence of e.g. letters (could be with repetition).

**Task:** On each enter-key pressing a (new) random sequence out of the given sequence shall be created and printed.

Lottery – 6 out of 49

Every week, many people play lotto, meaning they bet on a set of 6 numbers (out of 49 possible ones) in the hope to make money. Good players do exercises ;)

**Task:** Write such a RNG which spits a random set of 6 out of 49 numbers, without repetition (a number can occur at most once) on each pressing the enter-key.

7.2 Algebra

A $n$-dimensional vector is a one-dimensional array of numbers $v = (v_0, \ldots, v_{n-1})$. And a square $n$-by-$n$ matrix a two-dimensional array

$$
A = \begin{pmatrix}
a_{00} & a_{01} & \cdots \\
a_{10} & a_{11} & \cdots \\
& \ddots & \ddots \\
& & & a_{n-1,n-1}
\end{pmatrix}
$$

The vector and matrix components ($v_i$ and $a_{ij}$, respectively) are real (double) numbers.

To represent a vector structure in C++, one could take a native array, or e.g. `std::array<double, n>`, `std::valarray<double>` or `std::vector<double>` from the STL. As a matrix is a 2D array, one could use a native 2D array, or also a 1D array. Using a so-called *multi-index*, $k(i, j) = n \times i + j$, one can map $a_{00} \rightarrow a_0$, $a_{01} \rightarrow a_1$, ..., $a_{n-1,n-1} \rightarrow a_{n^2-1}$.

Select one structure and fill it (maybe also from a file) with numbers. For the beginning, a size of $n = 4$ or $5$ shall suffice.

7.2.1 Vector-Vector-Multiplication

One can do operations on these structures. The first and simplest is the *scalar product*. Two vectors, $v$ and $w$, are scalar multiplied by

$$
s = \sum_{i=0}^{n-1} v_i w_i .
$$
CHAPTER 7. HANDS-ON EXERCISES FOR PART 1

**Task:** Write a function which takes two vectors, and returns their scalar product!

**Hint:** Think about the function parameter types. Is `const` reference an option here? What problems do you see in this simple approach? How would you cure it?

### 7.2.2 Matrix-Vector-Multiplication

One can also multiply a matrix and a vector, where another vector results. Considering the input and output being from more or less the same state, this multiplication is often referred to as “applying a matrix (operation) on a vector”. In detail, having vector of length \( n \), \( v \), and a matrix of size \( n^2 \), \( A \), this multiplication results in \( w = Av \), or in components

\[
    w_i = \sum_{j=0}^{n-1} a_{ij} v_j \quad \text{for} \quad i = 0, \ldots, n - 1.
\]

**Task:** Write a function which takes a matrix and a vector, and returns their product (another vector)! Again, what issues might be problematic?

### 7.2.3 Matrix-Matrix-Multiplication

Finally, two \( n \times n \) matrices, \( A \) and \( B \), can be multiplied to a new matrix, \( C \).

\[
    c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \quad \text{for} \quad i, j = 0, \ldots, n - 1.
\]

**Task:** Write a function which takes two matrices, and returns their product (another matrix)! Also write a function, which prints a matrix to the screen in a formatted way!

### 7.3 Cryptography – Vigenére-XOR Encryption

Cryptography is about encoding and decoding text for securing information. This is accomplished by means of an obscuring algorithm, and some secret keys.\(^{11}\)

With computers, encryption is already simplified by the fact that characters are already encoded – Wiki: ASCII. As a `char` is one byte wide, i.e. eight bits long, one can encode \( 256 = 2^8 \) characters. For letters and special punctuation characters, we need much less. So, is, for instance, letter A (upper case) ASCII code 65, as can be checked via the following small program.

```cpp
#include <bitset>
#include <iostream>

int main() {
    char p = 'A';
    std::cout << (short int)p << "\n";
    std::cout << std::bitset<8>(p) << "\n";
}
```

65 is in binary (base 2) format 01000001 (we introduced here also STLs `bitset` (see cppreference: `std::bitset`), and a application for it).\(^{12}\)

If we now had a rather random bit sequence, e.g. 01111000 (ASCII ‘x’), we can easily encode the letter A above by means of XOR (exclusive OR). As 01101001 also corresponds to one ASCII character, we can easily encode ‘A’ by the following.

\(^{11}\)In contrast, cryptology also comprises additionally the methods how to crack codes, and to extract secret information without the knowledge of a key.

\(^{12}\)If this is strange for you, don’t panic! The mathematical explanations are not really necessary to do cryptography!
#include <bitset>
#include <iostream>

int main() {
  char p = 'A'; // plain text
  char k = 'x'; // key
  char c = p ^ k; // cipher text
  std::cout << p << "\n" << std::bitset<8>(p) << "\n" << std::bitset<8>(k) << "\n" << std::bitset<8>(c) << "\n";
  return 0;
}

The result of the XOR (c = p ^ k) is 00111001, the ASCII representation of the digit 9. So, we encrypted!

How can we reverse this process – decryption? With XOR, it is pretty easy. Just XOR the cipher text again, i.e. c ^ k results in the former bit pattern of p (just try it!).

Well! We can now encrypt, and decrypt text. But to use just a single character as key (‘x’, above), is not really clever or safe. It is too short for a password (key)! And testing 256 ASCII characters is probably no big deal for a modern code cracker to get access to the secret information!

The Vigenère cipher is a so-called poly-alphabetical cipher. The basic idea is to have a password as character sequence (the longer the better). And the first plain text character is now XOR-ed with the first password character, the second plain text character XOR-ed with the second password character, and so on. And when you reached the end of the password, you simply start again with the first password character. The encryption is finished, when all plain text characters have been XOR-ed. The decryption is done with the same procedure. But instead of the plain text, the cipher text is used as input.

**Task:** Realize such a Vigenère-XOR-Encoder!

**Hint:** Read the plain text from a file, and the password from the command line (or std::cin). Check the correctness of your implementation by viewing the cipher text (it should be cryptic). And decrypting this cipher text should result back in the original plain text.

### 7.4 Numeric

In school, you learned (probably) to analyze functions, \( f(x) \), and to characterize them by determination of zeros (roots), minima and maxima, etc. As we can realize functions \( f(x) \) in C++ by

```cpp
double f(double x) {
  ...
  return ...;
}
```

we can try to automate this process (approximately). For the following, chose one nice function to investigate. If you have none, take \( \sin(x) \) on the interval \([-3, 3]\).

First, we would like to find the zeros of \( f \) within a chosen interval \([a, b]\). The method of choice (and assuming a sufficiently well-behaved function) is to first put a fine equispaced grid over the interval, e.g. \( \varepsilon = \frac{b-a}{N} \) and \( x_k = a + k\varepsilon \ (k = 0, \ldots, N, \text{ and } N \text{ sufficiently large } \ldots \text{ say, } 100) \). Now, find those intervals \([x_k, x_{k+1}]\), where \( f(x_k)f(x_{k+1}) < 0 \). They should contain a zero of \( f \). As next, we

And also the bitsets are only to illustrate the underlying effects.

13In most cases, the result of XOR will result possibly in non printable characters. But that is OK for cryptography, because we anyway want to obscure!
14This is similar to the so-called Caesar cipher from about 2000 years ago (yes: that Julius Gaius Caesar!), which was such a mono-alphabetical code. In times of illiteracy and lack of general education, but specifically with computers, this was probably even safe enough.
15Make a sketch to illustrate this scenario!
only need to refine. For simplicity, we choose bisection, which works as follows. For the interval in question, determine the middle point by \( x_m = \frac{x_l + x_r}{2} \) (where \( l = k \) and \( r = k + 1 \) initially), and calculate \( f(x_m) \). If now \( f(x_l)f(x_m) < 0 \), then set \( x_r = x_m \) and restart. Otherwise set \( x_l = x_m \) and restart. This is interval bisection, approaching with the interval limits closer and closer to the zero.

One needs to stop this algorithm eventually. One criterion could be \( |x_r - x_l| < 10^{-10} \) (or some other number).

**Task:** Program this bisection function based on an interval \([x_k, x_{k+1}]\)! Inside the main, just loop over the grid to identify these candidate intervals before calling the bisection function.

**Hint:** As double f(double) is a global function, you can use the interval limits as parameters for the bisection function. More keen programmers can try to give also a function pointer to the bisection function (see below).

Next, we would like to find the minima/maxima of \( f \) (within the chosen interval) – and preferably all of them! There are several ways to do this. But the simplest is to look for zeros of the first derivative of \( f \) (and check the second derivative for the sign). Now, there is a problem, as we only have \( f(x) \). You can now either do the derivative by hand, and write two functions \( df \) and \( d2f \). Or you determine the derivative numerically. One way of doing so is to choose a small value \( \kappa > 0 \), and calculate the derivative value of \( f(x) \) at position \( x \) by

\[
f'(x) = \frac{4D(\kappa/2) - D(\kappa)}{3}
\]

where \( D(\kappa) = \frac{f(x + \kappa) - f(x - \kappa)}{2\kappa} \).

Similarly, the second derivative of \( f \) at position \( x \) can be calculated via

\[
f''(x) = \frac{4F(\kappa/2) - F(\kappa)}{3}
\]

and here \( F(\kappa) = \frac{f(x + \kappa) - 2f(x) + f(x - \kappa)}{\kappa^2} \).

**Task:** Implement these functions, double \( df(double) \) and double \( d2f(double) \)! As before put a grid over the interval and find the zeros of \( f'(x) \), i.e. \( f'(\hat{x}) = 0 \)! If \( f''(\hat{x}) > 0 \), \( \hat{x} \) is a minimum. Otherwise, a maximum (unless it is exactly zero, where we have a saddle point).

**Hint:** When implementing these functions, it may come to mind that it would be useful to be able to give functions as parameters to other functions. In this way, we could e.g. reuse the bisection function. This is definitely possible. With the following using declaration, we can rewrite the bisection function’s signature.

```c
// you recognize the function signature?
using Fprt = double (*)(double);
double bisection(double xl, double xr, Fprt g) {
  ...
  // here all things with g
  return ...
}
...
double zero = bisection(xl,xr,f);
...
zero = bisection(xl,xr,df);
```

**Remark:** You can also give Lambda functions to bisection, as long as no capture clause is included. Only the signature is important.

```c
auto f2 = [] (double x) { return x*x*x; }
...
double zero = bisection(xl,xr,f2);
```

---

16 Keen people could try Newton’s method! But it might not converge – a case that must be captured!
17 We will later learn more object-oriented ways to do the same thing!
Task: By this mean, try to write the above analysis more generically (i.e. also \( \text{df} \) and \( \text{d2f} \)), such that you can easily change or add a function only in a single place, and can analyze it!

7.5 STL Containers and Algorithms – Sieve of Eratosthenes

Some 2000 years ago, a Greek mathematician conceived an algorithm on how to generate the prime numbers – i.e. those that can be divided by one or by itself without a rest – from the natural numbers. It goes like this. Starting from 2, cut out all even numbers (numbers divisible by 2). From the remaining, cut out all numbers divisible by 3. Next natural number would be four – but it was already removed before, because it is divisible by 2. So, next is number 5 – cut out all numbers divisible by 5. Then comes 7, 11, 13, etc. The principle is clear?

Task: Write a program that prints out the prime numbers up to 100000!

Hint: Look for the so-called erase-remove idiom!

7.6 Computer – Unix Tools

Under [21 Best Linux Command Cheat Sheets](#), you can find Cheat Sheets for Unix-Tools. Many of these tools are part of the POSIX standard, and all are diligent little helpers. It doesn’t matter if you don’t know them, yet. That’s the perfect time to get to know them.

Tools like `wc` (word count), `grep` (global/regular expression/print), `sed` (stream editor), `awk` (acronym for Aho, Weinberger, Kernighan), etc. can be used to achieve complex tasks on the shell command line. Their power lies in the fact that they can be combined within `bash` programs for instance with `find`, in order to e.g. search for and modify certain information in files.

7.6.1 wc – word count

`wc` is a tool to count lines, words, characters, and some things more of an ASCII file.

```bash
$ wc text.txt
17 56 403 text.txt
```

Here, 17 lines, 56 words, and 403 characters are counted.

Task: Realize `wc` in a simplified Version! Create a program `wc`, which counts lines, words, and characters of an ASCII file. You can read from `stdin`, and supply the file content to be parsed via piping.

```bash
$ ./wc < text.txt
```

Or you use `std::ifstream` to open a file, the name of which you supply via command line.

```bash
$ ./wc text.txt
```

In order this to get working, you can write your `main()` function as follows.

```cpp
#include <string>
int main(int argc, char * argv[]) {
    std::string filename(argv[1]);
    ...
}
```
CHAPTER 7. HANDS-ON EXERCISES FOR PART 1

Hint: You can use any of the STL containers and algorithms and streams (and iterators) you understand and know how to get them working.

7.6.2 grep – text search in files

grep parses one or more ASCII files for a certain search pattern.

```bash
$ grep "text" text.txt
3: In this file is text.
```

This indicates that text.txt contains the word text in line three. (Often the found search pattern is even colored.)

Task: Realize a simple grep program which takes a search pattern and one or more file names

```bash
$ grep <pattern> <file list>
```

Print out the filename, the line number and the line, where the pattern matched!

Hint: If you are really keen, you can try the cppreference: regular expression library of the STL to perform the pattern matching.

7.6.3 More Unix Tools?

Select another tool from the list of many tools from a cheat sheet, and try to understand and realize it in C++! cat, tac, sort, uniq, diff, etc. are probably easier to realize than sed or awk.

Always try to find a way to use STL containers, algorithms, or any other library. The more you reuse, the less you need to write!
Part II

User Defined Data Structures, Classes, and Object-Oriented Design
Chapter 8

User Defined Types

User defined types are types, where the programmer has some freedom (sometimes considerable) to determine the layout (also in memory). We start with the classic user defined types, which are already available in native C – `struct`, `union`, `enum`. However, some of the behavior was extended in C++, as we will see. For Classes – THE objects in C++ – we dedicate an own chapter.

8.1 Union

Union is kind of an obscure structure, not to say a special type (see [cppreference: union]). The idea is to have a “variable” that can be of different desired types at the same time and during run-time. This is achieved by that the same memory is interpreted in different ways for different types. Declaration and usage is accomplished as follows.

```cpp
#include <iostream>

union U {
    int i;
    double d;
    char c;
};

int main() {
    U u {97};
    std::cout << " u.i= " << u.i << " u.d= " << u.d
              << " u.c= " << u.c << "\n";
}
```

Lines 2-6 are the declaration of the union. Inside, we declare three types – `int`, `double`, and `char`. In line 8, we create an object of that type, and initialize with a number. In the following output, we interpret this number as `int`, `double`, and `char`, respectively. The type is labeled by the name of the member variable – `i`, `d`, or `c`.

The way to use an initialized list as in line 8 here (instead of assignment, `=`), is feasible since C++ 11. It has some advantage it terms of performance and clarity.

As of C++ 17, there is a STL replacement `std::variant` (see [cppreference: variant]), which is much more flexible. Still, unions have their fields of application.

8.2 Enum

Enumerations can be considered as labeled integer type. It is used for more readable code as the following example might illuminate.

```cpp
#include <iostream>

enum Color { red, green, blue };

int main() {
    Color r = red;
}
```
CHAPTER 8. USER DEFINED TYPES

```cpp
switch(r) {
    case red : std::cout << "red\n"; break;
    case green: std::cout << "green\n"; break;
    case blue : std::cout << "blue\n"; break;
}
```

Line 2 declares an enumeration with three states – red, green, and blue. You can imagine to replace them directly by 1, 2, and 3. But any other numbers are also possible. If you need, you can even assign specific numbers,

```cpp
#include <iostream>
enum Color { red=20, green=30, blue=40 }
int main() {
    Color r (red);
    std::cout << "Color's number is " << r << "\n";
    switch(r) {
        case red : std::cout << "red\n"; break;
        case green: std::cout << "green\n"; break;
        case blue : std::cout << "blue\n"; break;
    }
}
```

The way to use an initialized list as in line 4 here (instead of assignment as done above), is feasible since C++ 11. red, green, and blue can be used synonym to literal 20, 30, and 40, respectively. Therefore, you can use them in switch-case constructs.

For modeling finite state automate, enumerations can be very useful. However, those can be now modeled also by means of classes. So, also enumerations are therefore like unions only of special use.

### 8.3 Struct

Structs are in its simplest form just structured compounds of other variables (see [cppreference: Struct]). They can contain built-in type variables, but also other user defined and compound types.

For example, we can define a 2D point "Point_2D", containing two coordinates – \( x \) and \( y \).

```cpp
struct Point2D {
    double x, y;
};
```

This is just the declaration of a type – a pattern if you want – not more and not less. When you wish an object of that type (with real memory acquisition), you must create and instance. And as with any other type, you can create and handle many independent instances (objects) at the same time. An example illustrates the usage.

```cpp
#include <iostream>
struct Point2D {
    double x, y;
};
int main() {
    Point2D p {1.0, 2.0};
    std::cout << " 2D Point (" << p.x << "," << p.y << ")\n";
    p.x = 4.0;
    p.y = 5.0;
    std::cout << " 2D Point (" << p.x << "," << p.y << ")\n";
}
```

In line 6, we create (instantiate) a Point2D object, called \( p \), and initialize it via an initializer list (C++ 11). The order of the numbers in braces is of the same as in the declaration of the member attributes above in line 3. Access (read/write) to these class members \( x \) and \( y \) is accomplished by object name, a dot ".", and the member name – as we did already for unions.\(^1\)

\(^1\)The member access operator \( . \) is correct for reference objects. For pointers, you must use \( \ast \) instead. See below!
### 8.3. STRUCT

**Handling**

Imagine, you would like to create a second point, and initialize it to be the first point.

```cpp
#include <iostream>
struct Point2D {
    double x, y;
};
int main() {
    Point2D p {1.0, 2.0};
    std::cout << "2D Point p(" << p.x << "," << p.y << ")\n";
    Point2D q = p;
    q.x = 3.0;
    std::cout << "2D Point q(" << q.x << "," << q.y << ")\n";
}
```

The members of `p` are copied element-wise. But otherwise, `p` and `q` are independent objects. Changing `q` afterwards does not change `p`.

**References (Reference Objects)**

As for built-in types, you can also create references and pointers to structures. Here, an example with references.

```cpp
#include <iostream>
struct Point2D {
    double x, y;
};
int main() {
    Point2D p {1.0, 2.0};
    std::cout << "2D Point p(" << p.x << "," << p.y << ")\n";
    Point2D &q = p;
    q.x = 3.0;
    std::cout << "2D Point p(" << q.x << "," << q.y << ")\n";
}
```

In the last line of `main()`, we print the same point’s coordinates, after we change the `x`-coordinate of `q`, being a reference (remember, kind of `const` pointer) to `p`.

**Structs as Function Parameters**

As should be familiar from the first part of that course, we can also transfer structs as function parameter — again in the three flavors: call-by-value (copies the struct completely into the function), call-by-reference (as `(const)` reference; address of the object is copied into the function, but object is handled like by value), or call-by-reference as pointer. For illustration, here is call-by-reference.

```cpp
#include <iostream>
struct Point2D {
    double x, y;
};
void change_point(Point2D& p) {
    p.x=4.0;
    p.y=5.0;
}
int main() {
    Point2D p {1.0, 2.0};
    std::cout << "2D Point p(" << p.x << "," << p.y << ")\n";
    change_point(p);
    std::cout << "2D Point p(" << p.x << "," << p.y << ")\n";
}
```

Please note that we used the label `p` several times — also inside the function. That is okay, because the scopes of the function bodies are disjunct. Surely, it may not help for better readability of the code.
Exercise: Remove the address-of operator &, re-compile and re-run. What do you observe (which call-by-... is it now)?
Or, instead, add a const before the address-of operator &; re-compile and re-run. What do you observe (which call-by-... is it now)?

In pointer semantics, you would write it like this.

```cpp
#include <iostream>
struct Point2D {
    double x, y;
};
void change_point(Point2D * q) {
    q->x=4.0;                      // <- change
    q->y=5.0;                      // <- change
}
int main() {
    Point2D p {1.0, 2.0};
    std::cout << "2D Point p(" << p.x << "," << p.y << ")\n";
    change_point(&p);              // <- change
    std::cout << "2D Point p(" << p.x << "," << p.y << ")\n";
}
```

Semantically, this is exactly the same as before with references. However, as you can see, pointer semantics causes some clutter. But still, you can easily decipher this. Point2D * q can be read either as Point2D* q (q is a pointer to a Point2D) or as Point2D *q (*q is a Point2D). The latter is called dereferencing, you remember?.

Where you used a dot . to access the member attribute of a reference object, you have to use -> now, when doing the same with a pointer. (Alternatively, you can also dereference. So, q->x and (*q).x do exactly the same thing – access to the x-coordinate attribute of the pointer q).

Finally, as change_point requires an address to a Point2D object, but p is the object itself, we need to get its address by &p.

I guess, now it becomes more clear, why value semantics is so much better to comprehend. Stick to it and avoid pointers whenever you can!

More Complex Structures

So far, this struct is only a kind of container. You can also combine different data types as shown in the following example.

```cpp
struct Person {
    int age;
    std::string first_name;
    std::string last_name;
    std::vector<std::string> hobbies;
};
```

The member attributes can be accessed again as shown before.

```cpp
Person me {10, "Pippi", "Longsocks", {"living in a villa",
    "riding a horse",
    "having a monkey",
    "adventures",
    "bothering teachers"}};
std::cout << "My name is " << me.first_name << "",
    " and I'm " << me.age << " years old.\n"
    "My hobbies are:\n"
for(auto & h : me.hobbies)
    std::cout << h << "\n";
```

---

2Parentheses around (*q) is often important because of operator precedence!
Exercise: Create a struct called DB_Item, representing an item in a literature database consisting of books, booklets, CDs, DVDs, ..., with different genres (e.g. science fiction, fantasy, adventure, western, ...). Each item also has a title, an author (director), year of publication. You can use all user defined types learned so far. Create some objects of items, and put them into a vector. From a function void list(const std::vector<DB_Item> & db), print all items in your database to the screen.

Help for Solution

The following code is thought as an example on which you can orient yourself.

```cpp
#include <iostream>
#include <string>
#include <vector>

enum item_type {book, booklet, CD, DVD};

struct DB_Item {
    int year;
    std::string title;
    std::string author;
    item_type type;
};

using DB = std::vector<DB_Item>;

void list(DB const & db) {
    for(auto & it : db) {
        cout << "Title: " << it.title << "\n";
        cout << "Author: " << it.author << "\n";
        cout << "year of publication: " << it.year << "\n";
        switch(it.type) {
            case book : cout << "Book\n"; break;
            case booklet : cout << "Booklet\n"; break;
            case CD : cout << "CD\n"; break;
            case DVD : cout << "DVD\n"; break;
        }
    }
}

int main() {
    DB my_db;
    my_db.push_back({1848,"Manifest der Kommunistischen Partei", "Karl Marx",booklet});
    my_db.push_back({1989,"Leningrad", "Billy Joel",CD});
    list(my_db);
}
```

Next Level – Unleash the Power of STL Algorithms

Exercise: Pick up the last example (maybe add more items), and sort them according to Title and according to year of publication. Hint: std::sort and lambda expressions are very useful. If you want to know whether there is at least one DVD in that collection, which algorithm would you use?

For those that are bored by that task, you can put all the items into an ASCII text file, and implement to read into the my_db from the file. Hint: As this is a dedicated task, you can create a function doing this for you. This also meets the needs of the Single Responsibility Principle.
Chapter 9

Classes

Classes and Structs are at the heart of object orientation. The basic idea of this programming concept is that we express everything in terms of objects – actors, if you like – that do something. We saw already this concept in action with STL strings and vectors. In order that this works, an object not only has data attributes, as we had in the examples for structs above. Much more must it be possible to hide these attributes from the direct access from outside. The Single Responsibility Principle (SRP) says that the responsibility to handle the data, which are represented by the data attributes (also called data members), must lie on the object itself.

This raises, of course, a problem now. If nobody can access directly the attributes, how would you change them? The answer to this are class member functions (sometimes also called methods, but that’s more Java slang). That is, functions specifically assigned to classes, and which can only be executed by an instance (a concrete realization, or a real object) of a class. The size of a std::vector, for instance, was obtainable via the class member function size(). But we could call it only for a vector object – std::vector v; v.size(). In this respect, size() is not a free function.

9.1 Struct as a special Class

9.1.1 Member Functions

Let us start with the simple example of the Point2D structure from above, add two member functions.

```cpp
#include <iostream>
struct Point2D {
    double x, y;
    double getX() { return x; }
    double getY() { return y; }
};
int main() {
    Point2D p { 2., 3.};
    std::cout << "Point p(" << p.getX() << "," << p.getY() << ")\n";
}
```

We also used these methods immediately to obtain the coordinate values for printing them. Of course, we had could use p.x and p.y instead. This would be much shorter. A bit patience, please!

Now, imagine that you don’t want the point in Cartesian coordinates, but e.g. in polar coordinates. You could, of course, externally write sqrt(p.x*p.x+p.y*p.y). But that is actually violating the SRP, as it is in the responsibility of Point2D to supply this information. So, we extend the declaration of Point2D.

```cpp
#include <cmath>
struct Point2D {
    double x, y;
    double r() { return sqrt(x*x+y*y); }
    double theta() { return atan2(y, x); }
};
```

1It turns out that struct is a special class.
CHAPTER 9. CLASSES

double x, y;
double getX() { return x; }
double getY() { return y; }
double getR() { return std::sqrt(x*x+y*y); }
double getPhi() { return std::atan2(y,x); }
};

This could be a solution. But now, who says that x and y are Cartesian coordinates? A programmer might have the idea to misuse x and y as radius and angle, and initialize a point instance accordingly. Then calling getR() would definitely give the wrong result – something you would not expect from the name of this function. Semantically, a programmer would expect to get the radius of the point in polar coordinates. So, here is a great potential of errors.

One of the most important rules in C++: Stick to intuitive semantics!

9.1.2 Access Specifier – a first Glimpse

So, let us prevent direct access to the data members! We introduce an access-specifier private, which makes x and y private (i.e. not accessible from outside via p.x or p.y).

#include <cmath>
struct Point2D {
  double getX() { return x; }
  double getY() { return y; }
  double getR() { return std::sqrt(x*x+y*y); }
  double getPhi() { return std::atan2(y,x); }
private:
  double x, y;
};

Okay! But now another problem occurs. Creating an instance of Point2D as before via

Point2D p { 2., 3. };

fails to compile. Also accessing p.x and p.y directly is obviated by the compiler, as intended. So, we cannot assign anymore values to x and y. That’s bad! We need to change the strategy to create Point2D and access its coordinates – that’s now enforced by the design.

First of all, we can instantiate a point without initialization (for the moment – we’ll show later how you can cope with that limitation by use of constructors).

Point2D p;

We could also already get the points coordinate via p.getX() and p.getY(), or via p.getR() and p.getPhi(), respectively. But that’s not much of a help as long these coordinates are uninitialized. Initialization is rather straightforward since C++ 11.

#include <cmath>
#include <iostream>
struct Point2D {
  double getX() { return x; }
  double getY() { return y; }
  double getR() { return std::sqrt(x*x+y*y); }
  double getPhi() { return std::atan2(y,x); }
private:
  double x=0, y=0;
};

Well, ... no! That’s not exactly what we want. Because all points we instantiate have the coordinates (0, 0). This can at most be sort of a default. We but also want to change the coordinates!

So, let us introduce some more member functions. We also rename the internal coordinates to X and Y, as this is convenient.\(^2\)

\(^2\)Do you remember the exercises with the editor in the last part?
9.1. STRUCT AS A SPECIAL CLASS

```cpp
void setXY(double x, double y) { X = x; Y = y; }
    double getX() { return X; }
    double getY() { return Y; }
    double getR() { return std::sqrt(X*X+Y*Y); }
    double getPhi() { return std::atan2(Y,X); }
private:
        double X=0, Y=0;
};
```

```cpp
int main() {
        Point2D p;
        p.setXY(2.0,3.0);
        std::cout << "Point p(x=" << p.getX() << ",y=" << p.getY() << ")\n";
}  
```

Perfect! That works! We can create a point with default Cartesian coordinates (0,0). Via member functions, we can now set these coordinates. And by the names of the so-called setters (setXY(double,double)), semantically it is clear that these are Cartesian coordinates. We could now also add setRPhi(double, double). At the latest now, every programmer should know, which coordinate representation is used from the member function name she uses.

9.1.3 Object-Orientation at the Pulse – Abstraction

But we gain really more! In the way we designed Point2D, we hide also the internal representation of the point’s coordinate. That is without changing the accessible member function interface (names of the member functions and the respective function parameters), we can change the internal way of representing the coordinates. For instance, some programmer might prefer polar coordinates.

```cpp
#include <cmath>
#include <iostream>
struct Point2D {
        void setXY(double x, double y) {
            R = std::sqrt(x*x+y*y);
            F = std::atan2(y,x);
        }
        void setRPhi(double r, double phi) { R = r; F = phi; }
        double getX() { return R*std::cos(F); }
        double getY() { return R*std::sin(F); }
        double getR() { return R; }
        double getPhi() { return F; }
private:
        double R=0, F=0;
};
int main() {
        Point2D p;
        p.setXY(2.0,3.0);
        std::cout << "Point p(x=" << p.getX() << ",y=" << p.getY() << ")\n";
}  
```

The program compiles and gives exactly the same result as before – the usage of Point2D is still the same!

This is exactly that kind of abstraction that object orientation wants to achieve. Furthermore, will C++ help us in accomplishing this abstraction in a consistent way. You can try as an exercise to set the coordinates independently, that is via setX(double) and setY(double)! You will realize that this is not possible for arbitrary internal coordinate representations (like that for polar coordinates). This should make you contemplate, what a point is, and why you cannot specify it with only the one

---

3You may ask whether we should set x and y independently. That’s possible if you stick with Cartesian coordinate representation. As we’ll see in a moment, for representation-independent points, that’s not so easy anymore.
or the other coordinate!

### 9.1.4 Structs are Classes?

C++ know the keyword `class`. And expressing the `Point2D` struct as a class would look like this.

```cpp
class Point2D { // <-- here is a change!!
public: // <-- here is a change!!
    void setXY(double x, double y) {
        R = std::sqrt(x*x+y*y);
        F = std::atan2(y, x);
    };
    void setRPhi(double r, double phi) { R = r; F = phi; };
    double getX() { return R*std::cos(F); };
    double getY() { return R*std::sin(F); };
    double getR() { return R; };
    double getPhi() { return F; };
private:
    double R=0, F=0;
};
```

That's really everything! The rest is completely the same. `struct → class`, and introduction of the access specifier `public`, which means that all following members listed are freely accessible from outside — until the end of the class declaration, or until the change by another access specifier.

So, the real "big" difference between `struct` and `class` is that `struct` is per default `public`. While `class` is per default private. And because this is what you should mostly strive for — keep as many things as possible private (keep them encapsulated) — most of C++ books deal with `class` instead of `struct`.

There is a rather crude rule to make the member functions public, and the class data members (attributes) private. Well, there will be many exceptions! But for the beginning that is a good starting point. Encapsulate the data, and write public class member functions to act on the class's data.

Some might intervene that all the nice things we achieved above by means of structs and vectors and algorithms by mean of easy (public) access to the data members of a struct. That is true! But on contemplation, you will notice that the direct aka public access to data members is not necessary. Access via class member functions can mostly do the same thing. Abstraction is only more complicated on a first gaze, as it is necessary to handle complexity. It is actually the complexity that is harder to grasp. You may have already experienced this effect when using vectors, strings and algorithms. Without the abstraction, the complexity of those objects would be hardly manageable!

For you as an ongoing programmer, it is not necessary to start out fully in abstraction. C++ does not force you to program in a certain style. There are just principles that from experience over time prove reasonable and supporting in software design. Guided by this, also the C++ language evolved and evolves further.

### 9.1.5 Project Plans – Where should I put my Classes?

We now just used only one class for the beginning. Of course, you will never be content with just one class. And programming complex programs will sooner or later require more classes — many more classes — not to say complete class hierarchies. Putting all the class declarations and definitions/implementations into one single file, where the `main` function is also included, will cause the loss of overview definitely.

Therefore, it has prevailed that the class declarations are put into header files, which are then included a) into the files where the classes are used (as kind of forward declaration), and b) into files where the class implementation happens. So, what is the difference between declaration and implementation? As this is more of a theoretical dispute, let’s reformulate it differently. From a programmers perspective, there are programmers (creators of classes) and users (those who instantiate the classes as physical objects). Both can of course be the same person. But for instance in the case of the STL, you are mostly a user. Then, what is of interest to you in the header you include, is just the interface — which class data members and member functions are accessible by you.
9.1. STRUCT AS A SPECIAL CLASS

So, put everything in a header which is necessary for a user to know or use from your class. This statement is general enough to also include implementation code in headers such as static data members and member functions, inline member functions, templates, and more.4

For instance, the above point example code can be separated into three files.

```cpp
// Point2D.h - Declaration
#ifndef POINT2D_ 
#define POINT2D_ // include protection
class Point2D {
  public:
    void setXY(double x, double y);
    void setRPhi(double r, double phi);
    double getX();
    double getY();
    double getR();
    double getPhi();
  private:
    double R=0, F=0; 
  };

// Don't forget ; !!
#endif
```

The class declaration must be finished by a semi-colon! And also use the include protection! The names of the class member function arguments are not strictly necessary, as they represent just a forward declaration. But as the header contains the interface for the user of the class, and the order of the parameters plays a role when calling these functions, it is a very good habit, to label them reasonably (Semantics!!).

```cpp
// Point2D.cxx - the Point2D Implementation
#include "Point2D.h" // <- header include
#include <cmath>
void Point2D::setXY(double x, double y) {
  R = std::sqrt(x*x+y*y);
  F=std::atan2(y,x);
}
void Point2D::setRPhi(double r, double phi) {
  R = r;
  F = phi;
} 
double Point2D::getX() {
  return R*std::cos(F);
}
double Point2D::getY() {
  return R*std::sin(F);
}
double Point2D::getR() {
  return R;
}
double Point2D::getPhi() {
  return F;
}
```

Look at this carefully, where we placed semi-colons! In the class declaration, they are important! After implementation, they are not.

You may also wonder whether `Point2D::...` in front of the member function implementation means the same as the namespace scope resolution we saw in the first part. And the answer is to some extent "Yes!". Each class makes up its own namespace. But while namespaces can be split and spread over several files even, a class declaration cannot. If you want to define or implement a declared class member, you need to do this by the scope resolution operator.5

```cpp
// main.cxx -- Use of Point2D
#include "Point2D.h" // <- header include
#include <iostream>
```

---

4You may ask about private class data member and member functions. You are right! They should be put somewhere else. But for this, we forward you to the advanced C++ course.

5Static class members can only be called like this as we will see later.
int main() {
    Point2D p;
    p.setXY(2.0, 3.0);
    std::cout << "Point p(x= " << p.getX() << ", y= " << p.getY() << ")n;"
    std::cout << "Point p(r= " << p.getR() << ", phi= " << p.getPhi() << ")n;"
}

As a user this is what you want and expect. A short, clear, readable, comprehensible application code.

You may have noticed that we need most (here all) of the #includes ONLY in the implementation files (.cxx files). This is intended. You should stick to the guideline to include as few as possible header files inside your own headers – as much as possible, put them only into the implementation files (which are distinct translation units). The same holds true for using namespace declarations. Don't do this in your headers, unless you have no other choice!

CMakeLists.txt

If you have all three files in one project folder, you can add a CMakeLists.txt file with the following content.

```cmake
cmake_minimum_required (VERSION 3.5)
project (PointTest)
set(CMAKE_CXX_STANDARD 11)
add_executable(PointTest main.cxx Point2D.cxx)
```

Exercise: Alternatively, you can put the .cxx files in one subfolder (e.g. src), and the header in another (say include). Write a corresponding CMakeLists.txt file. Think about include_directories and PROJECT_SOURCE_DIR (see the extended CMake tutorial of part 1)!

9.1.6 Function Overloading and Default Parameters

As for normal free functions, it is also possible to overload class member functions and to give (from right to left in sequence) default parameters. The same overload rules apply.

9.2 Constructors/Destructors – special Class Member Functions

In most cases, you probably want to instantiate a class (as object) with parameters. This is the even more true if you want to create a const instance of a class. Furthermore, there might be the need to acquire, manage, and release resources within a class. The acquisition should be done at the time of instantiation, and the release at the time of destruction of the objects.

9.2.1 The Default Constructor

There are special kinds of member functions, constructors, that do the job. There are some rules on how to define c’tors (short for constructor).

- Constructors have no return value!!
- Constructors must have the same name as the class itself.
- There are different constructors possible, i.e. they can be overloaded (as the name is the same, they differ in the parameter list only).

Some constructors have a special signature and a definite semantic. Therefore, they are called by a special name.
Let us look at the example of the Point2D. We add some constructor here, the rest staying the same.

```cpp
class Point2D {
public:
    Point2D () {}
...
};
```

This constructor has an empty parameter list. It is therefore called a **Default Constructor** (see [cppreference: Default Constructor](#)). The compiler can create one for you, either when it will be needed (you call one), or if you specify

```cpp
Point2D () = default;
```

default is possible only for certain constructors, which are furthermore trivial. Here, for instance, we have not initialized anything. But because that is the meaning of a constructor, let us initialize something. The most natural way is probably the following.

```cpp
class Point2D {
public:
    Point2D () { R=F=0; }
...
};
```

As you saw, this could be already accomplished by default values for data members. Often in C++, different ways will lead to the same result. It is a bit of your taste, which way you finally prefer. So you can also use initialization lists.

```cpp
class Point2D {
public:
    Point2D () : R{0}, F{0} { }
...
};
```

In this case, the body of the constructor stays empty. For some cases, this is much more efficient, and for some cases the only possibility for initialization, as we will see when e.g. for inheritance. There is only one thing you should remember and stick to in habit. **The class data members are initialized in the sequence of their declaration in the class.** So, better keep this order of sequence also in the initialization lists. Keeping this in mind, you certainly would understand why

```cpp
class Point2D {
public:
    Point2D () : R{0}, F{R} { }
...
};
```

works, while

```cpp
class Point2D {
public:
    Point2D () : F{0}, R{F} { }
...
};
```

might cause problems. To catch such errors, it is a good idea to issue the compiler flag `-Wall`, which make the compiler a bit more critical about such things. Calling now in `main()`

```cpp
Point2D p;
```
or, equivalently,

```cpp
Point2D p {};```
calls precisely the default constructor.

---

6Do not confuse this with *initializer lists*, which are introduced later, too.

7If you do not believe, include some `std::cout` output in the constructor's body! To see what's going on, printing
9.2.2 Initializing Constructors

Next, we would like to initialize a 2D point by means of some coordinate pair – and not just setting the coordinates to \((0, 0)\). We can do this as follows. We simply add another constructor – this time one with parameters.

```cpp
class Point2D {
public:
    Point2D () : X{0}, Y{0} {} // ambiguous with default parameters below
    Point2D (double x=0, double y=0) : X{x}, Y{y} {} 
    void setXY(double x, double y) { X = x; Y = y; }
    double getX() { return X; }
    double getY() { return Y; }
    double getR() { return std::sqrt(X*X+Y*Y); }
    double getPhi() { return std::atan2(Y,X); }
private:
    double X=0, Y=0;
};
```

We hope it is obvious that we wanted a constructor, which takes Cartesian coordinates. At this point, we have hardly a way of flexibility to specify the point with different coordinate representation. It is feasible by means of an enum, which we give as third parameter to the constructor. This enum can parameterize, which of the set functions to use. We will illustrate this a bit later.

At this place, we switch back to the internal Cartesian coordinate representation, which makes it easier again to use initialization lists at the constructor. Furthermore, we set default values for all the parameters in the initializing constructor (see `cppreference: Constructors and Member Initializer Lists`). As this would allow for calling it without any parameters, this is ambiguous with the default constructor – so, we removed the less general default constructor.

```cpp
#include <cmath>
#include <iostream>
enum coordRep { cart, polar }
class Point2D {
public:
    // Point2D () : X(0), Y(0) {} // ambiguous with default parameters below
    void setXY(double x, double y) { X = x; Y = y; }
    double getXY() { return X; }
    double getR() { return std::sqrt(X*X+Y*Y); }
    double getPhi() { return std::atan2(Y,X); }
private:
    double X=0, Y=0;
};
```

We also used this constructor in the last two lines of `main()`.

With the enum, specifying the coordinate representation, the whole `Point2D` class looks as follows.

```cpp
#include <cmath>
#include <iostream>
enum coordRep { cart, polar }
class Point2D {
public:
    void setXY(double x, double y) { X = x; Y = y; }
    switch (CR) { 
        case polar: setRPhi(Coord1, Coord2); break;
        default: setXY(Coord1, Coord2);
    }
};
```

(output to screen is always a good thing for testing.)
### 9.2. CONSTRUCTORS/DESTRUCTORS – SPECIAL CLASS MEMBER FUNCTIONS

```cpp
void setRPhi(double r, double phi) {
    X = r*std::cos(phi);
    Y = r*std::sin(phi);
}

double getX() { return X; }
double getY() { return Y; }
double getR() { return std::sqrt(X*X+Y*Y); }
double getPhi() { return std::atan2(Y,X); }

private:
    double X=0, Y=0;
};
```

Although that is a possible solution, it already looks quite complicated.

**Remark:** The specification of an initializing constructor prevents the automatic creation of a default constructor by the compiler!

### 9.2.3 The Copy Constructor

Let us have a look on the following code.

```cpp
#include <cmath>
#include <iostream>
class Point2D {
public:
    Point2D(double x=0, double y=0) : X{x}, Y{y} { }
    void setXY(double x, double y) { X = x; Y = y; }
    double getX() { return X; }
    double getY() { return Y; }
    double getR() { return std::sqrt(X*X+Y*Y); }
    double getPhi() { return std::atan2(Y,X); }
private:
    double X=0, Y=0;
};

int main() {
    Point2D p {2.0,3.0};
    std::cout << "Point p(x=" << p.getX() << ",y=" << p.getY() << ")\n";
    std::cout << "Point p(r=" << p.getR() << ",phi=" << p.getPhi() << ")\n";
    Point2D q {1.0,polar};
    std::cout << "Point q(x=" << q.getX() << ",y=" << q.getY() << ")\n";
    std::cout << "Point q(r=" << q.getR() << ",phi=" << q.getPhi() << ")\n";
}
```

In line 16, we created here point q as a copy of point p. This is accomplished by the so-called *copy constructor* (see [cppreference: Copy Constructor](https://en.cppreference.com/w/cpp/language/copy_constructor)). As proof that it is a copy, we changed p afterwards, and printed out both points.

In the example above, the copy constructor of `Point2D` is again created as a default by the compiler, because we have not specified it explicitly. You can be explicit by writing it as

```cpp
Point2D(const Point2D& g) = default;
```

as we did with the default constructor. **The copy constructor is identified by the const reference argument (of same class type)!**

However, if we want to specify one explicitly, where something is deviating from the default behavior, we need to create a constructor with the following signature.
Point2D(const Point2D& g) : X(g.X), Y(g.Y) { }

It is important that the only parameter is a const reference to an object of the same class. Otherwise, the compiler will not recognize it as a copy constructor.

Semantically, that is clear. A copying process should create an identical copy of the original, without changing the original. Therefore, there is an equivalent operator =, called copy assignment. Usually, copy constructor and copy assignment appear together as partners. We saw already for structures that we could write the following.

Point2D p {2,3};
Point2D q = p;

Semantically, that’s exactly the same as

Point2D p {2,3};
Point2D q = p;

However, operators are treated separately from constructors in C++. Therefore, the assignment is really different in detail from the copy constructor. Amongst others, the assignment involves often temporaries. We will detail this later in this course when we introduce operator overloading.

**Shallow Copy versus Deep Copy**

So, when do we need to specify a copy constructor?

- When your class is semantically copy-able, and
- when the default copying does not work.

The latter point needs maybe some explanation. If your class data members are copy-able – that is if you stick with value semantics – the default copy constructor will copy them.

```cpp
#include <iostream>
#include <string>
#include <vector>

class MyCopyableClass {
public:
    MyCopyableClass(int i, double d, std::string s, std::vector<int> v)
         : I(i), D(d), S(s), V(v) { }
    void printClassMembers() {
        std::cout << std::endl;
        std::cout << "I=" << I << " D=" << D << " S=" << S << " V=[";
        for(auto& p : V)
            std::cout << " p=" << p << " ]";
    } 
    void changeClassMembers() { 
        I=0; D=0.; S="dummy";
        V.clear(); V.push_back(I); 
    }
private:
    int I;
    double D;
    std::string S;
    std::vector<int> V;
};

int main() {
    MyCopyableClass mcc1(1,2.,"Test",{1,2,3,4});
    mcc1.printClassMembers();
    MyCopyableClass mcc2 (mcc1);
    mcc1.changeClassMembers();
    mcc1.printClassMembers();
    mcc2.printClassMembers();
}```
We defined a class with class data members in value semantics (no pointers; also not what is contained in the vector). We print \texttt{mcc1} to check the content. Then, we copy \texttt{mcc1} into \texttt{mcc2}, change \texttt{mcc1} afterwards, and print the content of both, which now differ. That is, \texttt{mcc1} has changed independently from \texttt{mcc2}. The example is made a bit more complex, including string and vector, to show that this copy incorporates also user defined types.

Let us look at another example, now including a pointer.

```cpp
#include <iostream>
#include <string>
#include <vector>

class MyCopyableClass {
public:
  MyCopyableClass(double \* d) : D{d} {}  
  void printClassMembers() {
    std::cout << "-------------------\n";
    std::cout << \" D\" << (*D) << \"\n\";
  }
  void changeClassMembers() { (*D)=0.; }
private:
  double \* D;
};

int main() {
  double dd = 1.3;
  MyCopyableClass mcc1(&dd);
  mcc1.printClassMembers();
  MyCopyableClass mcc2 {mcc1};
  mcc1.changeClassMembers();
  mcc1.printClassMembers();
  mcc2.printClassMembers();
}
```

The most important observation is that it becomes obscure, which of the functions and objects is responsible for the resource (where the pointer points to) – i.e. who is supposed to change the value of the double \texttt{dd}. This is the reason, why pointer semantics is so intricate, and why we would like to avoid it if possible.

Nonetheless, let us investigate further what happens, when we create \texttt{mcc2} as a copy of \texttt{mcc1}. \texttt{D} is copied as is. That is, it is copied as a pointer, and simply the address it points to is copied. So, \texttt{mcc1} and \texttt{mcc2} point to the same address, i.e to the same object, \texttt{dd}. This is called a shallow copy. In this example, it is reasonable to perform a shallow copy, because the class data member \texttt{D} does not point on a resource that the class \texttt{MyCopyableClass} does manage. If it were so, we would have to manually copy also the object, the pointer points to, and copy it. Because dynamic memory management is subject of a later chapter, we just show a simplified example here.

```cpp
#include <iostream>
#include <string>
#include <vector>

class MyCopyableClass {
public:
  MyCopyableClass(double d) : D(d), E{\&D} {} 
  void printClassMembers() {
    std::cout << "-----------------\n";
    std::cout << \" D\" << (*E) << \"\n\";
  }
  void changeClassMembers() {(*E)=0.; }
private:
  double D;
  double \* E;
};

int main() {
  MyCopyableClass mcc1(1.3);
  mcc1.printClassMembers();
  MyCopyableClass mcc2 (mcc1);
  mcc1.changeClassMembers();
  mcc1.printClassMembers();
  mcc2.printClassMembers();
}
```
This time, E points per construction to the address of D in mcc1. The default copy constructor copies D and E from mcc1 into mcc2. The result looks as follows.

\[
\begin{align*}
&D=1.3 \quad *E=1.3 \\
&D=0 \quad *E=0 \\
&D=1.3 \quad *E=0 
\end{align*}
\]

In the first line, D=1.3 of mcc1, and as E points to it, its dereferenced value is also 1.3. Next we copied mcc1, and changed the value of what E is pointing to to zero. Printing mcc1 results in D=0, and *E, too. In the last line, the output of printing mcc2, shows D=1.3 unchanged – as should be, because we copied before changing mcc1. But *E is now zero. This is so, because it points to mcc1’s D – which we already changed.

If we don’t want it like this, but rather that E points to the OWN D, we need to perform a so-called deep copy. More precisely, we cannot rely on the default copy constructor, but write our own one. So we add the following copy constructor to the class definition.

```
MyCopyableClass(const MyCopyableClass& MC) : D(MC.D), E(&D) {}
```

The rest being the same. The result is, as hoped.

\[
\begin{align*}
&D=1.3 \quad *E=1.3 \\
&D=0 \quad *E=0 \\
&D=1.3 \quad *E=1.3 
\end{align*}
\]

E points to the desired location. How important that is will become obvious when we handle dynamic memory management!

### 9.2.4 Constructors and Access Specifiers

Up to now, we have made all constructors public. But could we make them also private?

Well, certainly not all of them (unless we want to render a class useless). And mostly only those with a special meaning such as the copy constructor. If we namely would do so, an object would not be copy-constructible. Can you conceive an object that should not be constructible as a copy?

Or is there a case, where you want to avoid the default construction of an object? You can achieve this by setting the default constructor as private, or by deleting those constructors you don’t want to be compiler generated. For default and copy constructor, this would look like this.

```
Point2D() = delete;      // default c'tor
Point2D(const Point2D& g) = delete; // copy c'tor
```

### 9.2.5 The Destructor

Finally, there is also a destructor, to be called when the object gets destroyed. There are also some rules.

- **There must be at most ONE destructor per class!**
- **Destructors do not have any parameters!**
- **The destructors name has a leading ~, supplemented by the name of the class.**
9.3 Lifetime of Objects in Value Semantics

We would like to investigate, which objects are created and destroyed when? Consider the following code snippet.

```cpp
#include <iostream>
using std::cout;
struct A {
    A(int i) : ai{i} { cout << "A c'tor " << ai << "\n"; }
    A() { cout << "A d'tor " << ai << "\n"; }
    private:
        int ai;
};
struct B {
    B(int i) : a{i}, bi{i} { cout << "B c'tor " << bi << "\n"; }
    B() { cout << "B d'tor " << bi << "\n"; }
    private:
        A a;
        int bi;
};
void doSmth() {
    B b (2);
}
int main() {
    A a (1);
    doSmth();
}
```

Class A has a constructor, which takes an int value – for bookkeeping. This integer value is copied into the integer data member, ai, where we store it (we need it later in the destructor again). Class B contains also an integer value, but also nested an A class member. The latter is initialize in the initializing constructor.

Furthermore, we created a free function doSmth(), which instantiates a B, and which is called from main(), after we created an instance of A. Question is, when is which of the objects created and destroyed – in which sequence?

**Exercise:** Think about what seems logical, and afterwards compile and run the code! In future, when you are unsure about such questions, build yourself a dummy like that above and test it!

We are talking here about the *lifetime* of objects. While in value semantics, that’s quite straightforward once you got to use it, it is far more complicated in pointer semantics. In value semantics, the automatic clean-up is guaranteed! To avoid resource leaks, stick to it, and have an easy life!
9.4 The "this" Pointer

Each instance of a class (struct) contains implicitly a pointer to itself. And this pointer is also the first (implicit) class member function argument. "Implicit" means that it is not explicitly written – unless I hadn’t told you, you maybe never had known about.

What is this this pointer good for? For clarity, sure. When you have a class like the following, it might not immediately clear which function doSmthgElse is meant.

```cpp
void doSmthgElse() { /* whatever 1 */
    struct A {
        void doSmth() { doSmthgElse(); } // <-- ??
        void doSmthgElse() { /* whatever 2 */}
    };
}
```

So, this is about scopes! The global doSmthgElse could have been meant, or the class member function. Of course, you could explicitly resolve the global namespace, if you really want the global function.

```cpp
void doSmthgElse() { /* whatever 1 */
    struct A {
        void doSmth() { ::doSmthgElse(); } // <-- whatever 1 !
        void doSmthgElse() { /* whatever 2 */}
    };
}
```

But what if you mean the class member? Then this is the matter of choice.

```cpp
void doSmthgElse() { /* whatever 1 */
    struct A {
        void doSmth() { this->doSmthgElse(); } // <-- whatever 2 !
        void doSmthgElse() { /* whatever 2 */}
    };
}
```

The same holds for data members!

We will see later that the this pointer plays an even more important role for operator overloading, and for Lambda functions (capture clause – see below), where instances (say, the handle) of an object must be given to other objects out of the instance itself.

9.5 Const Correctness

Most C++ beginners feel overwhelmed by the versatility and flexibility, and are afraid of doing mistakes. For one, C++’s flexibility is intended by design. C++ does not prohibit or constrain any way of programming, because there might be good reasons why those ways are necessary. On the other hand, there is a strong guide-lining effort within the C++ community to answer exactly this issue of good and stable programming. Some of the guidelines were already mentioned – like DRY or SRP.

Another one is the so-called const-correctness (see isocpp: const correctness). It does not state that you must put everything as constant. But you should have a look where you can support the compiler in identifying your errors. For instance, the following should not compile.

```cpp
#include <iostream>
struct Test {
    Test(int i) : I{i} {}
    void print() const {
        std::cout << " I=" << I << "\n";
        I=3; // error
    }
    void change() { I=5; }
private:
    int I;
};
void doSmth(const Test& t) {
    t.print();
    t.change(); // error
}
```
Let us go through this code step-by-step. `main()` creates a non-constant reference object of class `Test`. Then it calls `doSmth()` with that object as parameter. Although `t` is not declared constant, the compiler assumes that `void doSmth(const Test& t)` is the matching function and casts `t` into a `const Test&`. Now, inside `doSmth`, `t.print()` is called. As we have declared this class member function as `const`, the compiler will complain about the `I=3` assignment. The `const` specifier designates namely that when calling this function, the object does not change its state.

Next, `doSmth` calls `t.change()`, which is not declared `const`. So, the compiler must assume that it will change the state of `t`, and complains.

**Exercise:** Remove all the compromising lines (`I=3;` and `t.change();`) and try to compile (with option `-Wall`). If this works without problems, add a free function

```c
void doSmth(Test& t) {
    t.print();
    t.change();
}
```

Will it compile? If so, why?

## 9.6 Some Words about Class Design

For beginners, often the question arises, how to decide, which classes should be created, with which class data members and member functions. A beginning guideline could to look at the text describing what your program shall do. Underline the nouns, which will become the classes. The noun’s adjectives can become the class’s attributes (data members). And the verbs, describing what the objects are doing, become the member functions.

Well, you’ll notice soon that it is rarely so simple! But as a beginning, it is probably okay.
Chapter 10
Dynamic Memory Management

Eventually, you may wish to create objects or to acquire resources dynamically during run-time. As an inheritance from C, C++ admits this. But also as an inheritance, this is related to the wide field of possible pain and memory leaks. So, let us start out slowly.

The C/C++ memory model knows two pieces of memory for data – the stack, and the heap (or free store). The stack is build up step by step with deeper and deeper function call nesting, where several meta data (return jump address, return value, ...) are stored. But also the function local data (value semantics). When a function finishes, and the scope thus left (as we have seen in the last chapter), all local variables are removed from the stack again (automatically).

But sometimes, you need that an object survives the scope. Or you need objects that are transferred between scopes in an efficient way. We have seen that references, and return values, can do the job in most cases. Nonetheless, C++ lets you do also explicit memory handling – via pointers. And we describe by example how it works.

Beware! You are responsible for the resources you acquire!

10.1 Operators new and delete

Consider the following example.

```cpp
#include <iostream>

int main() {
    int *i = new int(2);
    std::cout << *i << "\n";
    delete i;
}
```

We have created an integer pointer i, and we created a new integer with initial value 2. The call to `new` returns the address of that newly allocated memory piece\(^1\), and assigns it to i. We could write this also as follows.

```cpp
#include <iostream>

int main() {
    int *i = nullptr;
    i = new int(2);
    if (i != nullptr)
        std::cout << *i << " " << i << "\n";
    delete i;
}
```

nullptr is a generic object indicating that the pointer does not point anywhere. We afterwards check whether the memory allocation was successful, and that i is not nullptr – because **Never dereference an invalid pointer!**\(^2\). Then namely, we can dereference it to read out its value – and

\(^1\)Or throws an error if no memory allocation was feasible.
\(^2\)... which would be the cause of a memory access violation!
print it to the screen. In the same course, we also print \( i \)’s value, i.e. the address where it points to. It will not surprise you that this address changes from program execution to program execution.

Finally, we delete the allocated memory again. This is necessary, because you might loose the address of that memory (\( i \) is deleted when it goes out of scope, or is overwritten), and that memory, acquired by \texttt{new}, is never released (until the program’s end) – a classical memory leak!

Between the calls of \texttt{new} and \texttt{delete}, a lot of things might happen. So, this kind of (pointer semantic) programming is rather risky, and prone to resource leaks and memory access violations. By using value semantics, you can tacitly avoid these problems. But if you need to create dynamically allocated memory, also go sure to release it again!

### 10.2 Operators new[] and delete[]

Of more practical issues are the array variants of \texttt{new} and \texttt{delete}. Suppose, you want to create an array of dynamic size, i.e. where the size is determined during run-time rather than during compile-time. A possible realization is the following.

```cpp
#include <iostream>
#include <limits>

class Array {
public:
  Array(int i) : sz{i}, arr{new double[i]} {}
  ~Array() { delete[] arr; }
  double& at(int i) { return arr[i]; }
  int size() { return sz; }
private:
  int sz;
  double* arr = nullptr;
};

void fill(Array& ar) {
  for(int i=0; i<ar.size(); ++i) ar.at(i) = i;
}

void print(Array& ar) {
  for(int i=0; i<ar.size(); ++i)
    std::cout << " arr[" << i << "]=" << ar.at(i); 
  std::cout << 
}

int main() {
  int size;
  do {
    std::cout << "Enter a number >0: ";
    std::cin >> size;
    if(std::cin.fail()) {
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
      continue;
    }
  } while(size<=0);
  std::cout << " size is " << size << "\n";

  Array bca {size};
  fill(bca);
  print(bca);
}
```

We use the constructor of \texttt{Array} to allocate the memory (uninitialized), and the destructor to release it. By the way of creating the array bca in value semantics, the call of the destructor – and thus the release of the allocated memory – is guaranteed!

We do also some input sanity check in order to assure that \texttt{size} is really a number and \( > 0 \). Filling and printing is done via some free functions, which employ \texttt{Array}’s \texttt{at(int)} class member function (here without bound check).

**Exercise:**
10.2. OPERATORS NEW[] AND DELETE[]

1) Add bound-checking in the \texttt{at(int index)} class member function!

2) Add a reasonable copy constructor! (Put for the moment \texttt{Array& \_operator=(const Array&){}} in the private section.)

3) Add a \texttt{resize(int newSize)} member function, which deletes the old, and allocates a new memory of a given size.

\textbf{Remark:} Now you may appreciate the STL containers a bit more!
Chapter 11

The Static Keyword

By the means learned so far you have already much of control over how your classes are to be used. We would like to show here how this can be used on purpose. At the same time, we introduce the keyword static, which can be sometimes really helpful.

11.1 Reference Counting

With the addition of a new keyword static, it is possible to create a simple reference-counted class. Here, we simply count how many objects of a class are created.

```cpp
#include <iostream>
struct RefCnt {
    RefCnt() {
        cnt++;
        std::cout << cnt << " RefCnt objects!\n";
    }
    ~RefCnt() {
        cnt--;
        std::cout << cnt << " RefCnt objects left!\n";
    }
    static void print() {
        std::cout << "There are " << cnt << " RefCnt object alive!\n";
    }
private:
    int d = 5;
    static int cnt;
};
int RefCnt::cnt=0; // static variable initialization
void doSmth() {
    RefCnt B {};
    RefCnt *C = new RefCnt{};
    RefCnt::print();
    delete C;
}
int main() {
    RefCnt::print();
    RefCnt A {};
    doSmth();
}
```

The variables and member functions declared static are kind of unique to the class, and can be accessed without the creation of an instance. Therefore, cnt is shared among the different instances of RefCnt. And each instance (here the constructors and destructors) can work on that variable, and thus share messages to the other instances.\(^1\)

\(^1\)WARNING!! This example is not thread-safe, and might experience race conditions if those instances are run in different threads!
The static variable cnt must be initialized separately in the global scope\(^2\), because it exists already before any object can be instantiated. This is true although cnt is private! And you cannot re-assign values later – except through public member functions.

We also created a static member function, which can be called before and without any instance of that class. It is called like a member of a namespace, i.e. using the name of the class, and the scope resolution operator, ::. Because this kind of member functions can work without instances, it can work only on static variables! Other non-static variables are not existent, possibly. So, you cannot access d from print().

On the other hand, cnt can be accessed also from non-static class member functions, as we showed here with the constructor and destructor.

### 11.2 Object only on Heap

Another application of access-specifiers and static is when you need objects that can only be created via new, i.e. on the free store.

```cpp
class On_free_store {
  ~On_free_store() {} // private d'tor
public:
  static void free(On_free_store* p) { delete p; }
}
On_free_store glob1; // error
int main() {
  On_free_store loc; // error
  On_free_store* p = new On_free_store(); // ok
  delete p; // error
  On_free_store::free(p); // fine
}
```

The private destructor prevents the (global or local) reference object creation, because the destructor for such objects needs to be called implicitly – which is not possible, if it is private. Also the explicit call of delete fails therefore. But the public and static member function free can do it, because it can access the private destructor.

**Exercise:** Remove step-by-step (comment out) the compromising lines until it compiles. To see what happens during run-time, you can add std::cout comments in constructor and destructor.

How to achieve a stack-only objects will be discussed in the next part.

\(^2\)static variables are sort of global variables!
Chapter 12

Lambda Expression – Capture Clause; Function Pointer and Function Objects

We’ve already seen that Lambda Functions are really useful little helpers. Even more so as they can be sort of stateful. This means that one can remember some information from one function call to the next one. An example will surely illustrate that.

```cpp
#include <iostream>
int main() {
    int i = 0;
    auto f = [&i](int j) { return i += j; };
    std::cout << f(2) << "\n";
    std::cout << f(2) << "\n";
}
```

f above is a function, which takes an integer argument, and returns an integer value (both by value). Usually, when we consider functions, the return value is mostly identical for the same input. Above it is not! We obtain 2 in the first call, and 4 in the second. Furthermore, as i is given by reference, i in the outer scope of the Lambda function is changed as well. This opens up very interesting possibilities.

Remark: With free functions, you could do already similar things when using global variables. But it can be very difficult to understand such functions when they access a variable that are not defined within the immediate scope, and nowhere to see (global variables could be defined even in headers – out of sight). Global variables should be avoided, as much as possible!

The capture clause has some nomenclature that we simply list.

- `[x]` : x is captured by value
- `[&x]` : x is captured by reference
- `[&x, y]` : x is captured by reference, y by value
- `[&]` : variables in the local scope are captured by reference
- `[=]` : variables in the local scope are captured by value
- `[this]` : within a class, the current instance is captured, and all members accessible

[&] and [=] captures only what is used inside the Lambda expression (see CppCon: O’Dwyer: Lambdas from Scratch).
Function Objects: When you try to do the following,

```cpp
#include <iostream>
using Fptr = int(*)(int);
void smthg(Fptr f) {
    std::cout << f(2) << "\n";
}
int main() {
    int i=0;
    Fptr f = [&i](int j){ return i+=j; };  // or auto f = ...
    smthg(f);
}
```
you will earn an error like this.

```
main.cxx: In function ‘int main()’:
main.cxx:8:40: error: cannot convert ‘main():<lambda(int)>’ to ‘Fptr’
    {aka ‘int (*)(int)’} in initialization
Fptr f = [&i](int j){ return i+=j; };
```

Although the Lambda’s signature is actually correct, the non-empty capture-clause changes the type of this object.

But there is a simple possibility to overcome this – std::function from the functional header. std::function is again a template, which is rather easy to use, but difficult to implement – thank you STL! It is sort of replacement for function pointers. The template parameter is just the function signature. The following now works.

```cpp
#include <functional>
#include <iostream>
using Fptr = std::function<int(int)>;
void smthg(Fptr f) {
    std::cout << f(2) << "\n";
}
int main() {
    int i=0;
    Fptr f = [&i](int j){ return i+=j; };
    smthg(f);
}
```

Some of you will surely ask "Who the hell would need something like this?!" and "That’s too complicated! Why is this subject of a basic C++ course?!". That’s fair, of course! An attempt of an answer is the following. The power of this construct will become obvious very soon! And the complexity is NOT in its usage as you can see! Actually, the STL in general is rather easy to use! The difficulties you experience here is to comprehend the concepts! Concepts are just of higher abstractions, and we usually not much used to them. This makes it difficult to understand them.

Let me try to illustrate what happens here. Because this is NOT related to the programming languages in its own. And you will experience these difficulties in any language, when you try to go to higher abstractions (mostly meaning you will try to simplify things syntactically)!

When you were a child, you may have had a situation similar to the one described now (at least we hope that you can imagine it). On the party of your fifth or sixth birthday, there were some other children celebrating the day with you. Each one got two muffins from your Mom. And when she now asked you how many muffins there are in total, you and your guests probably started to count the muffins – one-by-one. Addition was thus already present in your list of abstractions. Multiplication not – because otherwise you had counted the children, and multiplied that number by two. This had been twice as fast as counting the muffins separately. Furthermore, it is simpler, and therefore less prone to errors.

In a more modern history there is the essay of the young Carl Friedrich Gauss (CFG), sitting in school, and being charged with the task to sum up the numbers from one to hundred – as all the children in his class, too. "Who manages it to do this correctly, could go home!" – said the teacher. When CFG finished after five minutes with the correct result, his teacher was astonished ... and annoyed,
as we can imagine, because he certainly looked forward to a silent and relaxed afternoon (this was
the supposed intent of the teacher when posing this problem to his pupils). CFG used an abstraction,
which is known today as follows.

\[ \sum_{k=1}^{n} k = \frac{n(n + 1)}{2}. \]

(Proof is left to the reader!)

Our advice is: Please stay patient and open-minded, and play around. If you don’t see any
application now, you will do so certainly later in your life! For the moment only important is to
get to use to the concepts and the syntax. And we think you are clever! So, we are sure that you will
recognize the sense step-by-step when you continue patiently.
Chapter 13

CMake Configuration

13.1 CMake Variables and Configuration Adaption

You can configure everything related to the build, compilers, linkers, release type, etc., including all possible flags. CMake does respect this by accepting variables. Once you are in the build folder, you can execute `cmake` with many `-D<variable>=<value>` options. One such variable is `CMAKE_BUILD_TYPE` and can have the values `Release`, `Debug`, `RelWithDebInfo`, `MinSizeRel`, etc. For all, a default set of flags for the compiler is already specified.

Instead, you can specify `CMAKE_CXX_COMPILER` (e.g. `/usr/bin/g++`) for the compiler you want, and `CMAKE_CXX_FLAGS` (general), `CMAKE_CXX_FLAGS_DEBUG`, `CMAKE_CXX_FLAGS_MINSIZEREL`, `CMAKE_CXX_FLAGS_RELEASE`, `CMAKE_CXX_FLAGS_RELWITHDEBINFO`, etc. for the compiler flags (for the respective build type). There are also `CMAKE_LINKER` (e.g. `/usr/bin/ld`) and `CMAKE_EXE_LINKER_FLAGS`, etc. for the linker, and many more for other build related programs.

If you build also libraries, the default is that this library is a static library (also called an archive). The linking of static libraries incorporates a copy of the relevant library part into the executable. This might lead to faster code, if it does not increase in size too much. But in modern computers, dynamic libraries (also called shared libraries or shared objects) are usually considered more performing as they decrease often the memory footprint of a program. And they allow for a rather large run-time flexibility, up to even a run-time linking and plugin mechanism. If you want shared libraries build, use `-DBUILD_SHARED_LIBS=ON`.

A full example of a cmake command then looks as follows.

```bash
$ cmake -DBUILD_SHARED_LIBS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=. \
  -DCMAKE_CXX_COMPILER=$(which g++) \
  -DCMAKE_CXX_FLAGS="-O3" \
..```

Here, `-O3` switches on compiler optimization (level 3).

13.1.1 POSIX Tools to investigate Executables and Libraries (mostly Linux)

- `ls -l`: displays file ownership, access permissions, size, last modification date
- `stat`: displays detailed file access permissions, modification dates
- `file`: displays executable/library file information like type, architecture, SHA1
- `size`: show size of different segments of executables/libraries
- `ldd`: shows executables/libraries dependencies from dynamic libraries
• **nm**: shows symbol table of executables/libraries
• **readelf**: displays information about ELF\(^1\) files
• **objdump**: shows information of object files (disassembler)
• **gdb**: GNU Debugger

### 13.1.2 Libraries (Linux)

When you link libraries dynamically, the executable needs to know where to find this shared library during run-time. CMake is quite clever about this and sets the `RUNPATH` – to be checked via (see [Shared Libraries: Understanding Dynamic Loading](#))

\[\text{$\texttt{readelf -a <executable>} | \texttt{grep -i runpath}$}\]

If the library is not there, or `RUNPATH` not set, you will obtain an error during execution

\[\text{<executable>} : \text{error while loading shared libraries: <missing library>: cannot open shared object file: No such file or directory}\]

In that case, also `ldd` will tell you what is missing.

\[\text{$\texttt{ldd <executable>} \ \texttt{<missing library> => not found}$}\]

There are now two possibilities. Either you set `LD_LIBRARY_PATH` via

\[\text{$\texttt{export LD_LIBRARY_PATH}=$\texttt{LD_LIBRARY_PATH:<path-to-missing-library>}$}\]
\[\text{$\texttt{./<executable>}$}\]

Or, you can also preload the library directly via `LD_PRELOAD`

\[\text{$\texttt{export LD_PRELOAD}=<path-to-missing-library>/<missing-library>}$]\n\[\text{$\texttt{./<executable>}$}\]

### 13.2 GUI Tool: ccmake

If `cmake-curses-gui` is installed, you have `ccmake` at your disposal. Similar as with `cmake`, you can configure your build by entering a out-of-source directory and call

\[\text{$\texttt{ccmake <path to top-level CMakeLists.txt>}$}\]

Instead of running the configuration, a GUI is opened showing you several variables relevant for the project build. Using key `'c'`, you can configure your project, and variables are set, or added to the project. Pressing `'h'` gives you help – if needed (`'e'` exits from sub menus back to the main menu). You can move the cursor to a specific variable and press `Enter` key. Now you can modify the entry. Pressing `'d'` deletes an entry.

Once finished, you press `'g'` to generate the `Makefile`, and quit the GUI. If you face problems, or want to interrupt the configuration, press `'c'` to leave the GUI without configuration.

By default, only few variables are shown in the GUI. Pressing `'t'` toggles to a view to see all (also the implicit) variables. Lot of them may not be set if they are not relevant for the build.

I usually use this tool only for gazing through the (default) variables. This can be but also accomplished by means of Google. The command-line configuration is usually easier to do. And the string of the configuration command can be easier documented.

Once you ran `cmake`, you can afterwards run `ccmake` inside the build directory.

---

\(^1\)Executable and Linkable Format
Chapter 14

Refactoring

Programming is usually driven by needs of solving problems. Insofar, the usage of a programming language is sort of aid to think. In this process, the programmer usually produces highly volatile and complicated code, in order to get the problem solved, and the desired behavior of the program implemented. That is perfectly fine! Programming languages are a kind of design tool in this respect.

But good programmers should keep in mind that more time is spent to read the code than is spent to write it. This means, code maintenance. Of course, you want that your code is easy to read – not necessarily only by you, btw. – and easy to extend for e.g. new functionality. But usually, the code after the initial design phase is not maintainable in this respect.

Therefore, part of a good software project is code refactoring (see Wikipedia: Code Refactoring). In short this means that the code is rewritten again and again in order to simplify it (possibly also to optimize it at the same time), while keeping its behavior invariant. That’s why we introduced the idea of Test Driven Development (TDD) in the first part of this course. Because by using TDD, you can refactor your code, and at the same time test that you did not change its behavior.

So, what exactly could be refactored? First of all, for convenience, many programmers start with very short variable names. For writing, this is more efficient. For maintenance, it is horrible to read everywhere int i or i=..., etc. unless in very specific circumstances (loop variables). So, renaming variables is one of the most important things.

Next, in the fast code design and implementation stage, programmers tend to put too many things into the code structures – functions and classes – and overload them. Furthermore, they also violate the Single Responsibility Principle (SRP) (and other principles, too). In a refactoring stage, functions could be split into smaller functions, which adhere to the DRY and SRP (and other principles, as well). The same holds true for classes.

Once you got your problem solved by a code, you find that it could solve also other problems. This usually requires the generalization of your types and functions. Later in this course, you will learn what means polymorphism – actually, there are several kinds of polymorphism. But in the first step, programmers usually write many if-conditions instead. These are often performance killers, because they mostly act during run-time. However, many of those conditionals can be replaced by statically morphing objects – i.e. during compile-time, avoiding this problem during run-time. Generalization and polymorphism often go hand in hand – but are also abstractions often difficult to realize in early design periods. But during refactoring, the time should be used. In the course of this, not seldom it will be realized that easier solutions to your problem can be found (again, increasing the efficiency and/or performance of your code).

More subtle is that you feel that a function does too much, or uses a lot of data from another object. Those are indications to refactor. Extract parts as independent functions. Or move functions to another object – where the access to the vast class members is easier.

There is certainly more to code refactoring, but please be content with this introduction here, and be referred to the literature [8] in case of interest.
Chapter 15

Hands-On Exercises for Part 2

15.1 C++/Python Coupling – PyBind11 (Tutorial)

Python embraces that packages can be written in C/C++, and bound and used from inside Python. As a mean, we chose here PyBind11. It is a former part from Boost (boost.python), which is but still further developed (in contrast to boost.python). And so, PyBind11 is now in some respects easier to use than boost.python.

The first business is to install PyBind11. From the PyBind11 Git Page, you can download it, and observe that it is a cmake project. Building should therefore be no problem anymore for you.

We outline here the first steps to create the binding between C++ and Python to a point, where the interested programmer can start off on his/her own. In order that this works, the PyBind11 must have been successfully built before! Please, add the pybind11 installation path to CMAKE_PREFIX_PATH.

Now create a C++ file, say example.cxx, with the following content.

```cpp
#include <pybind11/pybind11.h>
#include <iostream>
#include <string>

class Example {
public:
    Example(double a) : ea(a) {
        std::cout << "Example: c'tor\n";
    }
    double getA() const { return ea; }
    Example & operator+=(const Example & other) {
        std::cout << "Example: +=\n";
        ea += other.ea;
        return *this;
    }
private:
    double ea;
};

int add(int i, int j) { return i + j; }

PYBIND11_MODULE(example, m) {
    m.doc() = "Python bindings for an example library";
    namespace py = pybind11;
    py::class_<Example>(m, "Example")
        .def(py::init([](double a) {
            return new Example(a);
        }))
        .def("__iadd__", &Example::operator+)
        .def("getA", &Example::getA);
    m.def("add", &add, "A function which adds two numbers");
}
```

Even a overload of operator +=. Operator overloading will be explained in the next part.
As you can see, the first part is quite normal C++. A class Example with a member and member functions. To bind this to python, the first thing to do is to include pybind11/pybind11.h. The next thing is to declare the Python objects. This is done via a macro called PYBIND11_MODULE. The first parameter of it is the Python module name that we later import. The second parameter is just a handle, to which we attach the docstring a the top. We can later read it like this (e.g. using ipython3).

In [1]: import example as ex
In [2]: print(ex.__doc__)
Python bindings for an example library

Next, we introduce a namespace py, which points to pybind11. We use it immediately in the next line. The nomenclature appears a bit strange. But sticking at this example, it will become clearer with the time. First, we create an instance of type py::class_<Example>, i.e. we call a constructor with parameter m (the handle), and a name "Example" (string). This name is the type name under Python. Hodgepodge is probably not a good idea here. So, we choose the same name as the C++ class name.

The following .def calls attach (or register) functions to the python class. py::init is used not to wrap the constructor, but to execute it. And __iadd__ is a label to overload operators in Python. With it, we wrap the operator overloading. The last member function is getA().

In the last line, we also register the free function add. It also obtains a docstring.

In order to translate this, we use the following CMakeLists.txt file. Before, please add the pybind11 installation path to CMAKE_PREFIX_PATH.

```
cmake_minimum_required (VERSION 3.5)
project(example)
find_package(pybind11 REQUIRED)
pybind11_add_module(example example.cxx)
```

This looks quite simple, doesn’t it? A shared library will be created with the name example (also better be consistent here!). Its name is e.g. example.cpython-37m-x86_64-linux-gnu.so, so it has quite more information in its name than just the package name (notably the used python version). 

After out-of-source building, when this library is present, one can directly use it in Python.

```
In [1]: import example as ex
In [2]: a=ex.Example(3.4)
Example: ctor
In [3]: b=ex.Example(1.7)
Example: ctor
In [4]: b.getA()
Out[4]: 1.7
In [5]: a+=b
Example: +=
In [6]: a.getA()
Out[6]: 5.1
In [7]: ex.add(3,7)
Out[7]: 10
```

**Task:** Acquaint yourself with the user documentation of pybind11 to bring more functionality in this example! For instance, create a function in C++, which you can use in Python to plot via matplotlib.

---

2The practiced eye recognizes here something with templates again, as we had with e.g. std::vector.
3You must stay in the build directory! For including this library into you user or system python repository, please
15.2 Statistics

15.2.1 The CERN ROOT C++ Library (Tutorial)

CERN ROOT is a C++ class hierarchy package for among others data analysis. This section shall give a shallow introduction about the capabilities of this tool. At the same time, it is a good opportunity to practice the handling of objects in C++. A rather complete and nice User Guide is available on the CERN ROOT Web Page.

The Interactive C++ Shell (CINT)

So, ROOT has a C++ interpreted as command line shell interface.

```
$ root -l
# to start the environment
root [0] TF1 f("f","x*exp(-x)",0,10);
root [1] f.Draw();
Info in <TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [2] .q
```

This produces windows as shown in Figure 15.1.

![Figure 15.1: A function drawn using CERN ROOT.](image)

You might be happy to learn that this is a fully interactive window. Clicking View → Editor, you see an editor. You can also switch on a tool bar and a status bar. Moving the mouse of an element (the function graph, or one of the axes), you can click on them – either left, to see the things to change in the editor, or right, to get a pull-down menu with options.

**Task:** Play around with the axes titles, colors, etc. Also do so for the function itself.

**ROOT Scripting**

Everything and more can be done also in scripts, which can be called using the `root` executable. Let’s retrace the former example – now as a script. Create a file named e.g. `test.cxx` (ending actually is not important; the default for ROOT is `.C`).

consult the Python documentation.
CHAPTER 15. HANDS-ON EXERCISES FOR PART 2

```cpp
void test() {
    TF1 f("f","x*exp(-x)",0,10);   // create function f
    f.SetNpx(200);                  // 200 points (100 is default)
    f.SetLineColor(4);              // change f's line color
    f.GetXaxis()->SetTitle("x-axis"); // set x-axis title
    f.GetYaxis()->SetTitle("y-axis"); // set y-axis title
    TCanvas c("c","c",800,600);    // create a canvas
    f.Draw();                       // draw f
    c.SaveAs("test.pdf");         // save the plot as PDF
}
```

It is important to have a function with the same name as the script itself! But else, you can define even more functions inside the script, and call them, if you like!

Run this script as follows!

```bash
$ root -l test.cxx
```

The PDF file will contain the same plot as before. But the axes are labeled. And the function curve is smoother and blue. (To leave the ROOT environment: .q <Enter>.)

Usually, you cannot see the plot in that way immediately. This is because of the lifetime of reference objects. And here, \( f \) and \( c \) are such reference objects, which stop to exist as soon as the \( \text{test()} \) scope is left. To overcome this, if needed, create dynamically allocated objects.

```cpp
void test() {
    TF1 *f = new TF1("f","x*exp(-x)",0,10);
    f->SetNpx(200);
    f->SetLineColor(4);
    f->GetXaxis()->SetTitle("x-axis");
    f->GetYaxis()->SetTitle("y-axis");
    TCanvas *c = new TCanvas("c","c",800,600);
    f->Draw();
    c->SaveAs("test.pdf");
}
```

As we do not delete the objects, they are persistent (as long at least as the ROOT command line shell is still open).

A last thing – parameters. Maybe you wish to execute ROOT scripts several times, but e.g. with different parameters – here illustrated by the color of the function, the plot resolutions, and the y-axis label.

```cpp
void test(int col, const char* s, int div=100) {
    TF1 *f = new TF1("f","x*exp(-x)",0,10);
    f->SetNpx(div);
    f->SetLineColor(col);
    f->GetXaxis()->SetTitle("x-axis");
    f->GetYaxis()->SetTitle(s);
    TCanvas *c = new TCanvas("c","c",800,600);
    f->Draw();
    c->SaveAs("test.pdf");
}
```

Execute this via (parameters are exemplary)

```bash
$ root -l 'test.cxx(3,"f(x)")'
```

So, default parameters are also possible here. The single quotes are necessary to prevent the bash from interpreting the bracket (and largely complaining by throwing errors).

---

4 You can open the PDF with **okular** or **evince** under Linux.

5 Also the formula for the function can be a parameter – const char*!
ROOT Scripts Compiled

There might be good reasons to compile the script somehow, because it is otherwise too slow. You can do this with ROOT scripts as well. Let’s take a new example, to show also more capabilities of ROOT.

```cpp
#include "TCanvas.h"
#include "TH1D.h"
#include "TRandom3.h"

void fitGaus(int num=100000) {  
  TRandom3 rng(1234);  
  TH1D *h = new TH1D("h","Gaus",100,-4.,8.);
  for(int i=0; i<num; ++i)  
    h->Fill(rng.Gaus(2,1.5));
  TCanvas *c = new TCanvas("c","c",800,600);
  h->SetFillColor(41);  // ?
  h->SetLineColor(kBlue); // kBlue == 4
  h->GetXaxis()->SetTitle("x");
  h->GetYaxis()->SetTitle("# of occurrences");
  h->Fit("gaus");
  h->Draw();
  c->SaveAs("hist.pdf");
}
```

We must include headers now. But that’s rather simple, because they are named exactly as the classes are named. To compile it, you run it as follows.

```
$ root -l 'fitGaus.cxx+(100000)'
```

Note the + character!

The first time you run it, it will take a while because of the compilation. A file called `fitGaus.cxx.so` will be there afterwards (and maybe more — Do not care!). But any later call again will execute the code with the speed of compiled programs.

**Task:** Practice a bit the compilation and the scripting. Familiarize yourself with the syntax! Use different random number distributions and fit them with different built-in functions.⁶

**Hint:** Btw. if you forgot some of the class’ member functions; just start the `root` CINT environment, and issue e.g. `TH1D::<tab>` or `TH1D::S<tab>` (meaning: press the tabulator key!). So, tab-completion works inside the ROOT environment of CINT.

ROOT in full-fledged C++ Programs

ROOT contains a lot of useful classes dealing with data. Simple functions and histograms could you see already in action. But there are also graphs, multidimensional functions and histograms, trees, files (ROOT has an own data file format), and more far beyond the scope of this course. But to illustrate how it could be used, we mock a simple experiment producing data on a sample base, which we write into a tree, which is written into a file. Afterwards we analyze it in different ways. We would like to show how ROOT can be used also as normal C++ library.

This is the content of `create_sample.cxx`.

```cpp
#include "TFile.h"
#include "TRandom3.h"
#include "TMath.h"
#include "TTree.h"

int main() {  
  const double sigma = 1.5; // std-deviation  
  const double rho = -0.5; // correlation coefficient  
  const double mux = 1.; // mean x  
  const double muy = 0.5; // mean y
```

⁶Mouse right-click on the histogram → *Fit Panel* opens a GUI to adjust the Fit settings, for interactive analysis.
In detail, after some constant definitions, we create a file, a RNG (similar as for the STL RNG), and then a ROOT tree with two branches \( x \) and \( y \). The values of \( x \) and \( y \) are thrown randomly, and then filled into the tree. Finally, the tree is written to the file.

We show two ways how to compile it. First, the classical way.

```
$ g++ -o create_sample create_sample.cxx $(root-config --cflags --libs)
```

root-config is a tool (hopefully in the PATH), which provides command-line options for includes and libraries (among others). We use here the bash's subshell capability, $\( \ldots \)$, to execute this command. For the compilation, only the resulting string is needed.

Second, via CMake, using the following `CMakeLists.txt`.

```
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
project(event)
find_package(ROOT REQUIRED)
include(${ROOT_USE_FILE})
add_executable(create_sample create_sample.cxx)
target_link_libraries(create_sample ${ROOT_LIBRARIES})
```

If `CMAKE_PREFIX_PATH` is not set, usually the ROOTSYS environment is set. And so one can extend the `CMAKE_PREFIX_PATH` by `list(APPEND CMAKE_PREFIX_PATH $ENV{ROOTSYS})` before `find_package(ROOT REQUIRED)`.

**Analysis 1:** The result of the execution of the above program should result in a file `data.root`. One can open it using the CINT shell.

```
$ root -l data.root
root [0]
Attaching file data.root as _file0...
(TFile *) 0x55d7a0cd4f20
root [1] TBrowser b
```

This opens the data browser. One can click on each leaf (\( x \) and \( y \)), which will automatically create a histogram as shown in Figure 15.2.

**Analysis 2:** Investigating interactively data has its limits. So we show here another, more automated way to analyze the tree data above.

```
#include "TCanvas.h"
#include "TChain.h"
#include "TF2.h"
#include "TH2D.h"
```
15.2. STATISTICS

Figure 15.2: The data browser in ROOT.

```
#include "TMath.h"

double Gaus2D(double* x, double *p) {
    // p[0] = Norm
    // p[1] = mean_x
    // p[2] = mean_y
    // p[3] = sigma_x
    // p[4] = sigma_y
    double val=(x[0]-p[1])*(x[0]-p[1])/(p[3]*p[3]);
    val-=2.*p[5]*(x[0]-p[1])*(x[1]-p[2])/(p[3]*p[4]);
    val+=(x[1]-p[2])*(x[1]-p[2])/(p[4]*p[4]);
    val/=(1.-p[5]*p[5]);
    return p[0]*TMath::Exp(-0.5*val);
}

int main() {
    TChain tch("T");  // name of tree must agree
    tch.Add("data.root");  // one can add several files
    double x, y;
    tch.SetBranchAddress("x", &x);
    tch.SetBranchAddress("y", &y);
    TH2D h("h", "2D Gaus", 100, -5, 7, 100, -6, 7);
    for(int i=0; i<tch.GetEntries(); ++i) {
        tch.GetEntry(i);
        h.Fill(x,y);
    }
    h.GetXaxis()->SetTitle("x-axis");
    h.GetYaxis()->SetTitle("y-axis");
    TF2 f("g2d",Gaus2D,-10,10,-10,10,6);
    f.SetParameters(140., 0., 0., 1., 1., 0.);
    h.Fit("g2d");
    // h.GetFunction("g2d")->GetParameters();  // returns an array
    h.GetFunction("g2d")->SetLineWidth(1);
    TCanvas c("c", "c", 1200, 1000);
    c.Divide(2,2);
    c.cd(1);
    h.SetTitle("Histogram 2D colored");
    h.Draw("colz");
    c.cd(2);
    TH2D *h2 = (TH2D*)h.Clone("h2");
    h2->SetTitle("Histogram Scatter Plot");
    h2->Draw("histscat");
    c.cd(3);
    TH2D *h3 = (TH2D*)h.Clone("h3");
    h3->SetTitle("Histogram 3D Lego Plot");
    h3->Draw("lego");
```
c.cd(4);
f.SetRange(-5,-6,7,7);
f.SetNpx(50);
f.SetNpy(50);
f.SetLineWidth(1);
f.SetLineColor(kGreen);
f.SetTitle("Fitted Gaussian");
f.GetXaxis()->SetTitle("x-axis");
f.GetYaxis()->SetTitle("y-axis");
f.GetZaxis()->SetTitle("f(x)" );
f.Draw("surf3");
c.SaveAs("result.pdf");
}

As can be seen, the tree can be read from the file in rather convenient fashion\textsuperscript{7}. In the \textit{for}-loop, we access the single events from the tree. This is done by first registering the address of a variable via SetBranchAddress. Afterwards, on setting the tree entry via GetEntry(i), the value of the branch variables are set in the outer variables. That’s kind of pointer magic! But probably, that’s hard to comprehend on a first gaze! Also, notice that we used sometimes reference objects, and sometimes pointers. That’s sort of harder business in ROOT to figure out when to use what, meaning that you need to find out, which function return what. Do you now agree that a thorough documentation is inevitable?\textsuperscript{8}

The next remarkable thing is that we define a 2D function with (6) free parameters. This function can now be fitted to the histogram. We only need to set some (reasonable) initial values for the parameters. For those who do not immediately see it: The function is a 2D Gaussian of the following form.

\[ f(x) = Ne^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)} \]

where \( x - \mu = (x - \mu_x, y - \mu_y) \), \( \Sigma = \begin{pmatrix} \sigma_x^2 & \rho \sigma_x \sigma_y \\ \rho \sigma_x \sigma_y & \sigma_y^2 \end{pmatrix} \).

\( \mu_x, \mu_y, \sigma_x, \sigma_y \) are the respective mean values and standard deviations. \( \rho \) is the correlation coefficient. Thus \( \Sigma \) is the so-called covariance matrix.

Finally, we plot the histogram, and the fit function, as shown in Figure 15.3.

The terminal output of the fitting procedure is the following.

\$, ./analyze\$

```
Info in <TCanvas::MakeDefCanvas>: created default TCanvas with name c1
FCN=4033.3 FROM MIGRAD STATUS=CONVERGED 288 CALLS 289 TOTAL
EDM=8.28695e-07 STRATEGY= 1 ERROR MATRIX ACCURATE

EXT PARAMETER STEP FIRST
NO. NAME VALUE ERROR SIZE DERIVATIVE
1 p0 1.27437e+02 6.01266e-01 1.27590e-02 2.57487e-04
2 p1 1.00212e+00 4.84463e-03 1.29843e-04 -1.29914e-01
3 p2 5.05733e-01 4.83328e-03 1.29529e-04 -2.85060e-01
4 p3 1.47712e+00 3.65691e-03 8.61075e-05 -9.91303e-02
5 p4 1.47408e+00 3.61981e-03 8.56353e-05 -7.74903e-04
6 p5 -5.02869e-01 2.57435e-03 6.50082e-05 5.55743e-03
```

Info in <TCanvas::Print>: pdf file result.pdf has been created

From here, you can read of the values of the fitted parameters, and the respective uncertainties. Compare the values with those we entered into the analysis!

**Task:** ROOT knows a lot of other ways to define functions. A very convenient one is via a formula. This goes also with 2D functions, TF2. Play around with these capabilities, and test out draw options!

\textsuperscript{7}For the user convenient! For the programmer probably not so!

\textsuperscript{8}Cloning is necessary in order to have and plot different objects. Otherwise, the title would be shown as in the last change.
In principle, you can configure each little bit – font types and colors, distances of texts and title from the figure. You can plot diagrams inside diagrams, or both within the plot. ROOT has a huge amount in capabilities. Using it will sooner or later teach you how C++ object handling can work.

15.2.2 Full-Text Analysis – Maps

Task: Pick some (larger) text from Internet and store it in an ASCII file (some literature piece from Shakespeare will do)! Write a program which counts the number of occurrence for each word, and give out the ten most frequent words.

Hint: Remember that some of the words may have punctuation characters such as , or . attached! Please consider cppreference: std::map for this task, and think about algorithms! In order that this works, we have to anticipate somewhat template structures. std::map works with std::pair, here e.g. with std::pair<std::string,int> a pair with a string key, and a string value. But otherwise, std::pair is just a simple struct with two data members: pair.first containing to the key (here a string), and pair.second containing the value (an integer).

15.2.3 A Histogram Class

A histogram is an approximation to a continuous distribution density function in statistics. An example of a Gaussian distribution can be found in the following figure 15.4.

So, a histogram is a collection of consecutive, non-overlapping bins, i.e. intervals with boundaries,
a width, and a content. This bin content signifies the amount of "events" falling into this interval — or, more correct, an approximation thereof.

Usually, we obtain random numbers from a Monte-Carlo simulation, or from an experimental setup measuring some physical quantity. Histograms can help us to visualize the random distribution of these random numbers. Event by event, one counts upward (usually by one) the content of that bin, in the interval of which the random value falls. This way, a histogram is filled.

Of course, every bin can have a different bin width. But we are interested in only equal-width bins — otherwise, we do not see the relevant shape of the distribution, but a distortion of it. We would like to have a Histogram class, which can be used like follows.

```cpp
#include "Histogram.h"
#include <random>

int main() {
    std::mt19937 gen(1234);
    std::normal_distribution<> d(1,1.5);
    Histogram h(100,-4.,6.);
    for(int i=0; i<100000; ++i)
        h.Fill(d(gen));
    h.Normalize(1.);
    h.WriteToGnuplotFile("test.txt");
    h.Draw();
}
```

Gaussian random numbers are generated. We create then a histogram with 100 bins in the interval [-4,6]. Afterwards, we fill this histogram in a for-Loop. Next, we normalize the histogram, i.e. we multiply one factor to each bin such that afterwards it holds,

\[
\sum_{i=1}^{\text{# bins}} \text{bin-content}_i \times \text{bin-width}_i = 1.
\]

Finally, we write the two columns of bin-center and bin-content to a file, which my the be plotted using Gnuplot. The plotting routine might look like this.

```cpp
// Histogram.cxx
#include <fstream> // WriteToGnuplotFile -> std::ofstream
#include <string>  // std::string
#include <sstream> // std::ostringstream
[...]
```
That is, we create first a file with the data, and use then the `system` call to execute an external command – here `gnuplot` to plot this file content using Gnuplot.\(^9\)

**Task:** Your task is now to implement the Histogram class. You are free to implement it as you like. But it must provide an interface according to

// part of Histogram.h
class Histogram {
public:
  Histogram(unsigned int NoOfBins, double xmin, double xmax);
  void Fill(double val, double weight=1);
  void WriteToGnuplotFile(std::string filename);
  void Draw();
  void Normalize(double Norm);
private:
  double MinX, MaxX;
  [...]  
};

The [...] means the private part of the class – e.g. some data structure carrying the bin data. Add possibly also a function, bool `HistogramEmpty()`, which checks whether the histogram is empty.

**Hint:** For implementing the Fill member function, you can use whatever mean you conceive reasonable – linear search, binary search, ... But check carefully whether that’s really the most efficient way.

**Hint:** Maybe, the Filling of the histogram could be done more directly, such that out of the for-loop, we can create a statement like the following.

```cpp
h.Fill(d,gen,100000);
```

Or, we might completely put this into a constructor.

### 15.2.4 Physics: MC Toy Model for Energy Straggling

Heavy charged particles loose their energy during a passage through a material due to many small collisions with the electrons inside the material (from time to time, even a collision with a nucleus might happen – but we neglect this). We can imagine this passage a stochastic process. We assume that the heavy particle travels a while without interaction (i.e. at constant energy). And after a random distance, it collides, and looses a part of its energy. Also this energy loss is random between 0 and some maximal energy (that’s due to the collision kinetics).

Without any derivation, the distance of travel is distributed according to an Exponential distribution, with parameter $\Sigma$ (which possible can depend on the particle’s (or usually more correct: the collision

\(^9\)This is surely not necessary. But it brings some fun to see the plot popping up. How Gnuplot can be used is not part of this course, but can easily be found in the internet.
This means, we have  

Let’s assume we have a thin wall (say, a detector). So,  

between the direction of flight before and after collision), or by  

verse  

so that we can create also from here a probability density,  

E, d, T,  

the mean energy loss, given all the parameters  

sake of simplicity. We simply play a game. We are interested in the distribution of energy loss, and energy loss per particle passing through the wall. Although that’s feasible, too, we stop here for the of thickness  

in average about  

Σ = 100 collisions per passage of a particle. More difficult to assess is the average energy loss per particle passing through the wall. Although that’s feasible, too, we stop here for the sake of simplicity. We simply play a game. We are interested in the distribution of energy loss, and the mean energy loss, given all the parameter  

E, d, T, Σ.

We start with a particle of energy  

E_0 = 10^5 \text{ eV}  

at the left-hand side of the wall at  

x_0 = 0, and move it according to the following process (I = 10 \text{ eV}).

1. Randomly create a distance  

Δx by throwing from  

p(Δx) = Σe^{−ΣΔx}. Move to that position, i.e.  

x_{n+1} = x_n + Δx.  

2. Randomly throw from  

π(T) ∝ \frac{1}{T^2}  

in [I, 0.2(E_n − I) + I]. Subtract the Energy accordingly,  

E_{n+1} = E_n − T.  

3. Stop at the right place and count!  

If  

x_{n+1} ≥ d = 10^{-4}m, the particle left the wall. Calculate  

(E_0 − E_n)/E_0  

and fill it into a histogram (from 0 to 1; normalized energy loss).  

Start with a new particle  

E = E_0, x_0 = 0!

Task: Implement this process! And answer the questions: How much energy does a particle loose in average? And plot the energy loss distribution!

Hint: This just illustrates the principle how MC particle transport e.g. for detector systems works. If that’s too simple, include the next dimension (deviation from the straight line). Or, add another interaction that e.g. creates another particle, such as a photon, generated as bremsstrahlung.

---

10Σ = σn, where  

σ is the microscopic interaction cross-section [m^2], and  

n is the collision center density, [m^{-3}].

11That’s rather simplistic.  

I is just a mean ionization energy. And  

T_{max} is a function of the energy  

E of the particle, but, except for equal kind collision partners, not necessarily proportional to the particle’s energy  

E.

12eV = 1.602 · 10^{-19}J is a common unit for energy in particle physics. But this is not essential here in the mock-up!
Bayes’ Theorem in the form

\[ P(A|B) = \frac{P(B|A)P(A)}{P(B)} \]

is a statement about relations between probabilities of two propositions \( A \) and \( B \). Herein, we denote the conditional probability \( P(A|B) \) as the posterior probability, and \( P(B|A) \) the likelihood. \( P(A) \) is the prior probability for \( A \), and \( P(B) \) is a normalization factor – sometime called the evidence. But let’s forget about the terms, and focus on an application.

Imagine, we have some data \( D \), and a model with parameters \( \theta \) (read it as a vector), which shall describe the data. Then we can use Bayes’ Theorem to formulate the following,

\[ P(\theta|D) \propto P(D|\theta)P(\theta) . \]

We can express this statement as follows. Given some prior knowledge (or reasoned assumption; \( P(\theta) \)) about the parameters \( \theta \), we can use the model (likelihood; \( P(D|\theta) \)) to infer more probable parameter value \( \theta \) based on the knowledge supplied by the data. The posterior thus can be interpreted as the conclusion we draw about \( \theta \) after we received more information. Considered like this, Bayes’ Theorem is sort of an update scheme.

To be of some use, we write Bayes’ Theorem in differential form,

\[ \pi(\theta|D) \sim \pi(D|\theta)\pi(\theta) . \]

For illustration, we assume we measured some data as a sample of size \( N; \{x_i\}_{i=1}^N \). We assume that the \( x_i \) might be distributed exponentially (for instance!) with parameter \( \lambda \). Therefore,

\[ \pi(D|\theta) = \prod_{i=1}^N (\lambda e^{-\lambda x_i}) = \lambda^N e^{-\lambda \sum_{i=1}^N x_i} = \lambda^N e^{-\lambda \bar{x}} , \]

where \( \bar{x} \) is again the arithmetic mean of the sample values. Depending on our prior knowledge for \( \lambda \), \( \pi(\theta) \equiv \pi(\lambda) \), we can now update out knowledge about \( \lambda \) using the sample data.

Priors is a bit tricky stuff. The minimum knowledge we have is that \( \lambda > 0 \). This would cause the least informative prior (look for Maximum Entropy Principle (MAXENT)), \( \pi(\lambda) = \lambda^{-1} \). The posterior for this case would be

\[ \pi(\lambda|\{x_i\}) \sim \lambda^{N-1} e^{-\lambda \bar{x}} . \]

Here we are actually already finished. But people don’t like a distribution. They want a number (and maybe an uncertainty). Depending on the utility function, you can now easily find a number – e.g. \( \hat{\lambda} \) which maximizes the posterior (maximum posterior estimator); or the mean value (and variance) of the posterior, \( \langle \lambda \rangle \pm \sigma_\lambda \).

Well, the maximum value has the big advantage that the normalization of the posterior (which we currently don’t have) is not needed. Simply

\[ 0 = \frac{d\pi(\lambda|\{x_i\})}{d\lambda} \Rightarrow 0 = \frac{d}{d\lambda} \lambda^{N-1} e^{-\lambda \bar{x}} = ((N-1)\lambda^{N-2} - N \bar{x} \lambda^{N-1}) e^{-\lambda \bar{x}} , \]

which obviously results in

\[ \hat{\lambda} = \frac{N-1}{N \bar{x}} . \]

We leave it as an exercise to the reader what we would obtain if we had as prior knowledge a formerly measured value in the form \( \hat{\lambda}_p \pm \sigma_{\lambda_p} \), which results according to the Maximum Entropy Principle in Gaussian prior with mean \( \hat{\lambda}_p \) and variance \( \sigma_{\lambda_p}^2 \).

But as mentioned above, instead of searching for the maximum, we could also look for the posterior mean (and variance). One issue to do this analytically starts already with that we have no normalization. So, we would first have to determine \( \int_0^\infty \pi(\lambda|\{x_i\}) d\lambda = 1 \).
But wait! We are looking for
\[ \langle \lambda \rangle = \int_0^\infty \lambda \pi(\lambda | \{x_i\}) \, d\lambda \, ? \]

Didn’t we noticed in the previous part of the course that we could approximate this by
\[ \langle \lambda \rangle \approx \frac{1}{N} \sum_{j=1}^{M} \lambda_j , \]
when the \( \lambda_j \)s are distributed according to \( \pi(\lambda | \{x_i\}) \)? And we already know the Markov Chain Monte Carlo (MCMC) method to generate such \( \lambda_j \)s from even a non-normalized distribution – here, \( \lambda^{N-1}e^{-\lambda N \bar{x}} \), which is not normalized.

**Task:** Implement a MCMC process with \( \lambda^{N-1}e^{-\lambda N \bar{x}} \) as distribution, and determine from the resulting \( \lambda_j \)s the mean value (arithmetic mean), and the standard deviation! In order to test this, create \( x_i \)s before using an exponential RNG. Compare the approximation of \( \langle \lambda \rangle \) with the analytical maximum posterior result \( \hat{\lambda} \).

**Task:** (Curve fitting) Assume you have some data \( \{x_i, y_i, e_i\} \), where the \( e_i \) are the error values (standard deviations) of the \( y_i \)s. We may assume that \( (x_i, y_i) \) are value pairs that belong to a functional relationship \( y = f(x, \theta) \), where \( \theta \) is some set of parameters. These could for instance be the coefficients of a polynomial, if \( f \) is assumed to be a polynomial. For the likelihood, we can assume (MAXENT) a Gaussian
\[ \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi e_i}} e^{-\frac{(y_i-f(x_i))}{2e_i^2}}. \]

The prior of the \( \theta \) shall be assumed uniform, i.e. \( \pi(\theta) = const \). So, the posterior is
\[ \pi(\theta | D) \sim \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi e_i}} e^{-\frac{(y_i-f(x_i, \theta))}{2e_i^2}}. \]

Implement a MCMC process to sample the coefficients \( \theta \)! Determine the mean values (estimates) and the covariance matrix (if you know what that is).

**Hint:** Create a sample of \( (x_i, y_i, e_i) \) first. \( i = 1, ..., N \). \( e_i \) could all the same, \( e_i = e \), if you want. Now write a function to calculate the posterior given the data for a set \( \theta \). Choose \( y = f(x, \theta) \) to be a polynomial – say a parabola (2nd order) \( ax^2 + bx + c \). Now use this to generate MCMC samples for the \( a, b, c \). In order to do this, just sample one-dimensional into the \( a, b, \) and \( c \) direction separately via a Gaussian distribution (see previous part’s Hands-on tasks)!

This is called Gibbs-sampling. It might be necessary to have different orders of sizes for the test steps for \( a, b, \) and \( c \). Check the rejection rates in these directions. Also important: Choose some reasonable start values for \( a, b, \) and \( c \), such that the distribution is sampled in a representative fashion. Once you have these samples, calculate the arithmetic mean (and covariance).

**Hint:** Before starting to implement, first try to visualize the procedure – each single step. Take a sheet of paper and sketch the process. Try to decompose the procedure into subtasks that can be implemented and tested separately.

### 15.3 Algebra

#### 15.3.1 Square Matrix Class

**Task:** In the previous part, we use `std::array` in order to realize vectors and matrices. Now, let’s realize a square matrix using classes! Add a class member function for adding another matrix to the current matrix. Create a global function, which adds two matrices, and returns the result. Add matrix-matrix multiplication. Should this be done as class member? Or as free function? Add also a transpose function, and a scalar multiplication!
15.4. GEOMETRY – GEOMETRIC OBJECTS AND OPERATIONS

**Hint:** Add more class members and functions if needed!

**Task:** If you know how to realize it, add a function which calculates the inverse of the matrix! How would you test it?

15.3.2 Operations with Matrices and Vectors

**Task:** Create now also a vector class, and functions to multiply matrix and vector.

15.4 Geometry – Geometric Objects and Operations

**Task:** Create geometric 2D objects like a square, a triangle, a circle. Add class member functions for printing the objects position and orientation, and functions which change (move(distX,distY)) the position and orientation (rotate(angle)).

15.5 Refactoring

**Task:** Take one of your former exercise results and try to refactor! Look whether you have duplicated code and put this into a new function. Look for names you could/should change for better readability! Check whether STL containers or algorithms can simplify your code! Discuss with your partner what seems reasonable.

Take the chance and practice this (and also other techniques when the language features become available) in future always when you want to extend your code. This also include generalizations (e.g. templates, inheritance) as well as design patterns, when appropriate.

15.6 STL Containers and Algorithms

**Task:** Fill a std::vector<Person> with data for a structure Person, which has a first name (std::string), last name (std::string), age (int), gender (bool, e.g. male == true), profession (std::string). Fill the vector with about 20 persons, where in part repetitions of names or professions make the business more interesting.

Write a function, which prints the vector’s content! Next, write a loop with a input for a letter (char), and a switch case statement to dispatch the different following tasks in each loop passing.

a) Random shuffle the persons!

b) Sort according to the persons’ last name!

c) Sort according to the persons’ first name!

d) Sort according to the persons’ age!

e) Sort according to the persons’ profession, where else the order for persons of the same profession are preserved!

f) Print all male persons (do not remove persons)!

g) Find the oldest person (after random shuffling)!

h) Find the third youngest person (after random shuffling)!

i) Partition into a group with last names starting with A to M and the rest (after random shuffling)!

j) Partition into a group with last names starting with A to M with the rest preserving the order
within the group (after random shuffling)!
k) Check whether there is at least one carpenter!
l) Shift all persons a place to the right, the last becoming the first one!
m) Print all persons with an age between 30 and 40!
n) Sum up the ages of all persons!

Use free functions or Lambda expressions as you consider reasonable! Otherwise, realize the above tasks with STL algorithms!
Part III

Operator Overloading, Advanced I/O, and more STL
Chapter 16

Operator Overloading

There is a very powerful feature of C++, which makes code potentially much more readable – Operator Overloading (see cppreference: operator overloading). In essence, this means that you can define the behavior of operators such as =, +, >, etc. for your classes. Correct! These very operators you already used for the built-in types!

But some words of warning. Overuse or wrong usage of such a feature can bring more pain and frustration as you may want. Keep the semantics (the meaning of an operator) close to its natural sense. A + sign just means to add or connect (concatenate) something. Code, where 3+4 results in 12 would not be easily understood by anyone.

Furthermore, the operator precedence is built-in and fix (cppreference: C++ Operator Precedence). You cannot change it. For those who like operator overloading, this can be sometimes frustrating. Keep in mind that you can more or less achieve everything also by immediate member functions!

Last but not least, you cannot redefine operators that are built-in (especially those for the built-in and STL types)! This feature was made to allow extension of the language – not to change it!

16.1 Operator Overloading in Action

Operators can be unary (taking just a single operand), like ! (logical NOT) and ++ (post-/pre-increment), or binary (having two operands), such as +, -, etc. Some operators can be both (-, +, *, ...). Accordingly, you will have to think about whether operators shall be class members (for unary operators this makes often sense), or free functions (mostly for binary operators).

Some operators such as =, (), [] (index), -> (pointer member access) can also be overloaded with very interesting effects.

The only ternary operator of C++ ((cond) ? branch1 : branch2;) cannot be overloaded.

All operators return something.\(^1\) In any case, the overloading happens always in the same way.

\[
\text{ret-type operator} \ <\text{optype|new|delete...}> \ (<\text{param list}>) \ {\text{body}}
\]

ret-type is the type of the returned object. operator is a keyword (not negotiable). Then, the operator to be overloaded is specified.\(^2\) The parameter list can be more or less flexible in number of arguments, depending on the operator, which becomes overloaded.

Like all member functions, also member operators overloaded or defined within a class, have an implicit first parameter – this.

---

\(^1\)Otherwise, their sense would be largely obscured.

\(^2\)This can be also a type for conversion operations, or new,delete.
16.2 Operator Overloading by Example

There are many more operators such as bit manipulation operators, or referencing and member access operators, etc., which we do not deal with here. Some of them may become naturally introduced when we need them later in the course.

16.2.1 Assignment Operator, =

One of the most important operators is the assignment operator, =. Always when you design a class with a copy constructor, you probably also want explicitly a copy assignment operator ([cppreference: Copy assignment operator]) — meaning an assignment, which copies from the right-hand side (which stays unchanged) to the left-hand side.

In the point example of the previous part, we had such a copy constructor. And re-assigning a point makes certainly sense. So, let us include also a copy assignment.

```cpp
#include <cmath>
#include <iostream>

class Point2D {
public:
    Point2D(double x=0, double y=0) : X{x}, Y{y} { } // Default/Init c'tor
    Point2D(const Point2D& g) : X{g.X}, Y{g.Y} { } // Copy c'tor
    Point2D& operator=(const Point2D& rhs) {
        X = rhs.X; Y = rhs.Y;
        return *this; // return own reference
    }
    void setXY(double x, double y) { X = x; Y = y; }
    double getX() { return X; }
    double getY() { return Y; }
    double getR() { return std::sqrt(X*X+Y*Y); }
    double getPhi() { return std::atan2(Y,X); }

private:
    double X=0, Y=0;
};

int main() {
    Point2D p {2,3}; // init c'tor
    Point2D q {0,0}; // init c'tor
    q = p; // re-assignment
    p.setXY(4,5);
    std::cout << "Point p(" << p.getX() << "," << p.getY() << ")\n"
    std::cout << "Point q(" << q.getX() << "," << q.getY() << ")\n"
};
```

The parameter list contains a constant reference to the right-hand side (const is clear, because the right-hand side does not change — see: const correctness). Copy assignment must be a class member function — logically and semantically. And so it can be also used as follows.

```cpp
q.operator=(p);
```

Writing out operators in this explicit fashion (mentally, at least) often clarifies better if you get unexpected behavior by inadvertently screwing up the operator precedence. This might occur, when you chain several operation into sequence, as we will see immediately.

Some may wonder why we return a reference (non-const is hopefully obvious) to the own class, and then return *this. C++ admits for built-in types the following.

```cpp
int a,b,c;
a=b=c=4;
```

That is, a is afterwards (by value) equal to b, and equal to c. So, why shouldn’t be this also possible for points, or for (mathematical) vectors, or for matrices, or other objects? For points in the example above,

---

3White-spaces between operator, =, and the opening parenthesis ( don’t matter.
4Normal at the beginning!
5Remember Bjarne Stroustrup saying about operator overloading: “Do as the ints do!”.
Point2D p, q, r;
q = p = r = {4,5};

would create a unnamed temporary point (4,5), and would assign it to p, which returns its own reference as input to the assignment of q, etc.

Point2D p, q, r;
r.operator=(q.operator=(p.operator={4,5}));

But the former is much more comprehensible, won’t you agree?

Btw. from now on, we don’t need void setXY(double x, double y) anymore – at least not for the Cartesian coordinate representation. Simply

q = {4,5};

will do it. You can also use variables instead of number literals.

**Exercise:** Create a lattice of 10-by-10 points on a Cartesian grid, with distance 1 between neighboring points.

**Exercise:** Add std::cout statements to the init and copy constructors, and to the copy assignment, to figure out, which operation is used when!

So, be aware that those operations might have performance implications!

**Shallow- and Deep Copy, and Self-Assignment**

From the previous part 2, you may remember the copy construction, and our discussion about deep and shallow copies. This topic reappears here. In principle, everything from the copy construction holds verbatim for assignment. Take care to correctly assign the ownership as you intend, and clean resources!⁶

But in contrast to copy construction, we have a possible problem with assignment – namely self-assignment. Nobody prevents you from writing

```cpp
Point2D p {4,5};
p = p;
```

Although that’s certainly not really meaningful, mathematically and logically it’s not prohibited. Even worse, you cannot prevent this (on this level) unless you want to prohibit assignment at all.

For shallow copies, that’s usually not big of an issue. But once you perform memory management using new and delete, you might come into trouble and leak resources if you do not care. The usually way to prevent this disaster is to check for self-assignment.

```cpp
Point2D& operator=(const Point2D& rhs) {
    // check if this == rhs
    // if not equal, create new resources and copy
    return *this;
}
```

This may not be trivial. But checking e.g. for equality of pointers, pointing to the managed resource, is a possibility.

**16.2.2 Arithmetic Operations: +=,*=,-=,/=,%=**

Now might it be a nice feature, if our point could be added in place, i.e. we want to do something like

```cpp
Point2D p {2,3};
Point2D q {4,5};
q += p;
```

⁶Otherwise they would leak!
such that \( q = (6, 8) \) afterwards. We simply overload `+=`.

```cpp
class Point2D {
public:
    ...
    Point2D& operator += (const Point2D& rhs) {
        X += rhs.X; Y += rhs.Y;
        return *this;
    }
    ...
};
```

Interesting enough, we have used the same parameter list and return value as for assignment. When you think sharply, this makes sense, as you may also do the following.

```cpp
Point2D p {2,3}, q {4,5}, r;
r = q += p;
```

The compiler will translate this into

```cpp
r.operator=(q.operator+=(p));
```

So, chaining is not only possible, but also desirable in this case, though certainly not possible for all kinds of conceivable classes.

`-=` is verbatim copy-able from `+=`. In that way, we have introduced some kind of vector addition and subtraction. That is, as a refactoring step, should we change the name of the class from `Point2D` to e.g. `Vector2D`, because our objects now behave more like a vector. But mathematically, points can also be position vectors (being different from vectors as elements of an affine space) – and can be added and subtracted like vectors. A way around this might be using aliases!

```cpp
using Point2D = Vector2D;
```

Now, we can assign certain variables to be `Point2Ds`, and others to be `Vector2Ds` – internally, both are the same. And we can still use all the nice ways to add and assign them together. Optically, we can distinguish them in a program – that’s clean and comprehensible enough.

`%=` would be used for modulo operations. It makes no sense for points or vectors. But if it would make, you can redefine it.

But `*=` and `/=` could be useful, if we mean by that kind of scaling (usually `*=` would suffice to serve this purpose). In that case, we would like to interpret

```cpp
Vector2D p {1,3};
p *= 10;
```

as coordinate-wise multiplication by a scalar. We simply overload `*=`.

```cpp
Vector2D& operator *= (double val) {
    X *= val; Y *= val;
    return *this;
}
```

Again, it makes sense to return `*this` for chaining of operations. Also, `*=` must return a `Vector2D`, because the scaling produces a vector, and not a scalar.

### 16.2.3 Arithmetic Operations: `+,-,\ast,/,\%`

Binary operators are usually not class members (although they can be). Often, in combination with conversion constructor and operators, it is really convenient, to make binary operators free, i.e. non-members.
The friend Keyword

Let us start with the simple scalar (inner, or dot) product.

```cpp
double operator*(const Vector2D& v1, const Vector2D& v2) {
  return v1.getX() * v2.getX() + v1.getY() * v2.getY();
}
```

```cpp
... Vector2D p{2, 3}, q{3, 5};
std::cout << " p*q = " << p*q << "n;"
```

This will fail to compile until `getX` and `getY` are marked as `const`.

```cpp
double getX() const { return X; }
double getY() const { return Y; }
```

If you want access to private class members, make operators a friend of a class.

```cpp
class Point2D {
  public:
    ...
    friend double operator*(const Vector2D& v1, const Vector2D& v2);
    ...
  }

  double operator*(const Vector2D& v1, const Vector2D& v2) {
    return v1.X * v2.X + v1.Y * v2.Y; // no getX() etc. here
  }
```

These friends must be declared in the public section of a class. Don’t overuse this feature!

For the scalar product, it should not play a role whether to compute `p*q` or `q*p` (scalar multiplication is commutative). Obviously, that’s fulfilled here.

Again, you can read the product as a function call. So, `p*q` then will be resolved by the compiler as

```cpp
::operator*(p, q);
```

The scope resolution operator takes care that you take the operator from the global space.

**Exercise:** We can add and subtract two vectors. This is also easy to realize. Realize it! Does it make sense to make those operators friends? Please, consider that the member operators `+=` are already defined. Does it help?

```cpp
Vector2D operator+(Vector2D v1, const Vector2D& v2) {
  return v1+=v2;
}
```

Some explanation. Here, we return a `Vector2D`, and not a reference. Furthermore, the first argument is also `Vector2D`, and not const `Vector2D&`. The effect is that we create `v1` as a copy of the first operand. We can (mis-)use it for our own good, because it is a temporary object. We simply add the right-hand side to it and return it.

Division and Modulo make no sense here – semantically. But if you find a reasonable application for it, you can use it.

**Type Conversion Constructor and the explicit Keyword**

Next, we would like to have also sort of a free multiplication of a scalar and a vector, `q=p*10`. or `q=10.*p`. The signature of the free operator would be

---

7 The return value optimization (RVO) might optimize this temporary copy, which is returned, away.
CHAPTER 16. OPERATOR OVERLOADING

Vector2D operator*(Vector2D v, double s) {
    return v*=s;
}

Having this, the following would compile, and give the resulting vector (20, 30).

Vector2D p {2,3};
Vector2D r = p*10.;

But what would happen if we had written the following?

Vector2D p {2,3};
Vector2D r = 10.*p;

The * operator above would require the vector first, and the scalar last. Nonetheless, it compiles, and runs. But what you obtain is (20, 0) – not exactly what you expected, right? So, what happened here? And how can we cure it?

The C++ compiler is quite clever here. 10 is converted to (10, 0) by the initializing constructor – thus becoming a conversion constructor. Next, we have the multiplication of two vectors (scalar product), resulting in 20 – a scalar! But the left-hand side, r, is a vector. So, the compiler employs again the initializing constructor to create the vector (20, 0) from the scalar 20. Logical! Right?

For beginners of C++, that is often very strange! C++ does a lot of things implicitly. But you will learn to appreciate it, because most of the time this is what you want.

In this case, the simple solution is to provide

Vector2D operator*(double s, Vector2D v) {
    return v*=s;
}

This matches much more directly. And so the compiler will take this one instead of the complicated procedure above.

You can also prevent the compiler from doing the nonsense above by making the initialization constructor explicit.

explicit Vector2D(double x=0, double y=0) : X{x}, Y{y} { }

Exercise: Comment out or delete Vector2D operator*(double, Vector2D) with keeping the initialization constructor explicit. Try to compile it!

Again, we returned here a plain Vector2D, and no reference. And also the vector parameter is transferred by value, thus creating a temporary Vector2D object, which can be modified and returned.

16.2.4 Type Conversion Operators aka User-Defined Conversion

As we have seen in the last section, constructors can be used to convert one type into another – that of the class the constructor belongs to. There is also opposite way by means of user-defined conversion (see cppreference: user-defined conversion). Frankly speaking, this immediately opens up the big field of type casting, which we postpone to a bit later – part 4. But we want to refer very interested readers to cplusplus: Type Conversion, and outline here only the principle idea an syntax of conversion.

Imagine you have two types that can be expressed by each other. This means, both have a lot of behavior in common, and you can somehow make one into another type, and vice versa.

#include <iostream>
struct A {
    void print() { std::cout << "I'm A\n"; }
};
struct B {
    B (const A&) {} // conversion c'tor from A

---

8Each constructor creating an object from an object of any arbitrary other type is a conversion constructor.
9Copy Elision aka Return Value Optimization will avoid an extra copy.
16.2. OPERATOR OVERLOADING BY EXAMPLE

```cpp
B& operator= (const A& x) { return *this; } // conversion assign from A
operator A() { return A{}; } // conversion to A (type-cast operator)
void print() { std::cout << "I'm A\n"; }
int main() {
  A foo;
  foo.print();
  B bar {foo}; // calls conversion c'tor
  bar.print();
  bar = foo; // calls conversion assignment
  bar.print();
  foo = bar; // calls type-cast operator
  foo.print();
}
```

You may think that this is a design flaw, and that C++ spoils the strong type safety by this. Actually there are other candidates of C++ language features that could be accused for that crime. But user-defined conversion does not belong to it. The opposite is actually the case! The programmer is supposed to write conversions ONLY for types that can be converted into each other in some meaningful way.

Let us consider as example a DateTime class. Time can be a point of time with some format, expressed by a character string such as "Sa 13. Jul 12:43:39 CEST 2019". But time can also be a time period – the difference of two points. Internally, in a computer, the (Unix) time is an integer counting the seconds since 1st January 1970, 00:00:00. Differences of integers are straightforward. But maybe you want to get the period of time in terms of Hours, Days, Months, Years, ...

Here are plenty of possibilities for conversion from one type to another.

**Exercise:** Write a DateTime class with all the conversions you consider as reasonable!

### 16.2.5 Comparison Operators: ==, !=, >, <, >=, <=

Which objects are comparable – mathematically spoken, which objects have kind of an order? Quantifiable things for sure: integers, doubles. Strings, as we have seen: by lexicographical order. But we can be even worse. We can (surely not objectively) order books or movies according to how much we like them. Singers, Web pages, dishes, ... In most cases, this can be somehow represented by an integer point system. For instance,

```cpp
struct Movies {
  std::string title;
  Year publishingYear;
  int likes;
};
```

We have already seen that we can sort a vector (container) of Movies by means of lambda expressions. Within the lambda expression, we could locally define the sorting order by comparing respective class members.

By operator overloading, we can do this but also globally.

```cpp
bool operator==(const Movie& m1, const Movie& m2) {
  bool equal = (m1.title == m2.title);
  return (equal && (m1.publishingYear == m2.publishingYear));
}
```

This would compare for equality by title and name, and could be used for checking whether a new DB entry designates an item already being present in the DB.

For bringing movies in a container into a sequence, you still need to overload <, or >. We can even change the semantics, what we mean by "less" or "less than".

```cpp
bool operator<(const Movie& m1, const Movie& m2) {
  return (m1.likes < m2.likes);
}
```

It then can be used as follows
Movie m1 {"2001: A Space Odyssey",1968,100};
Movie m2 {"Alien",1979,100};
if(m1 == m2)
  std::cout << "Both movies are the same!\n";
if(m1 < m2)
  std::cout << "Movie 2 is the better one!\n";

And since sorting goes with <, we could sort our movie data base for the their popularity.

This might be but also a bit unexpected, as another programmer would like to sort movies in a lexicographic way for their title. Or, for the director/producer (if we extend the Movie class in that direction).

After lambda expressions, that's the second way to get access to the STL algorithms for sorting, partitioning, extracting, searching, ... global overloading of comparison operators.

**Exercise:** Pick up the exercise at the end of the struct section in the previous part – DB_Item. Add comparison operators, and sort the DB (std::vector<DB_Item>). Also search (with user input) for an item in the database!

### 16.2.6 Pre-/Post-In-/De-crement

For built-in basic types like integers, pre-increment and pre-decrement are designated by ++i and --i, respectively. Semantically, it means that the object under consideration is first changed, and then returned. For a user-defined class, this can be accomplished by

```cpp
class A {
  public:
    ...;
    A& operator++() { // pre-increment
      // change the inner state
      return *this;
    }
    ...;
};
```

And accordingly with pre-decrement, when we overload them inside classes.\(^{10}\)

In contrast, post-increment and post-decrement, indicated by i++ and i--, respectively, have the meaning of "change the object AFTERWARDS, and return the former initial state". As operator++ already exists, and redefining would be ambiguous, the C++ designers created the convention, that for post-in/de-crement, the operator takes an int argument, although this is never used. Therefore,

```cpp
class A {
  public:
    ...;
    A operator++(int) { // post-increment
      A tmp {*this}; // make a copy
      // change the inner state
      return tmp;
    }
    ...;
};
```

Similarly with post-increment. As you can see, there is a temporary object involved in post-increment. This might have performance implications. Therefore, a lot of style guides vote for pre-in/decrement in for-loops.

**Beware!!** For post-in/decrement (and any function), don’t (never ever) return a reference! Beginners, and also advanced C++ programmers, often commit this mistake, to return a locally defined object as reference from a function.

\(^{10}\)You can do this also outside a class (see cppreference: Increment/decrement operators).
Exercise: What might be the problem here? Discuss, or try and figure out (with some chance)!

16.2.7 Index-Operator

In the previous part of this course, we created an Array class.

```cpp
class Array {
public:
  Array (int size) : sz {size}, arr{new double[size]} {}  
  Array() { delete [] arr; }
  double& at(int pos) { return arr[pos]; }
  int size() { return sz; }
private:
  int sz ;
  double * arr = nullptr;
};
```

In those days, we introduced the at(int) member function to have access to the array elements. In the std::vector and std::string classes, this member function indeed exists – and performs even bound checking. Something you hopefully have already implemented as an exercise.

But of course, as for normal arrays, we would like to use our array also with [int] indexing.

```cpp
Array myArr {10};
for(int i=0, end=myArr.size(); i<end; ++i) myArr[i]=i;
```

With overloading of the operator[] (index operator), that's possible.

```cpp
class Array {
public:
  ... 
  double& operator[](int pos) { return arr[pos]; } // return a reference!
  ... 
};
```

Some clever C++ pupils might ask " Couldn't I create a matrix indexing by this?". The answer is "Well, it depends!". The following does not work,

```cpp
class Matrix {
public:
  ... 
  double& operator[](int row, int col) { return arr[row*size+col]; }
  ...
private:
  double *arr;
};
```

as C++ requires that operator[] has exactly one argument. So, M[i,j] will not be possible!

Exercise: We had already shown in the hands-on exercises of part 1 how one can obtain 2D indexing, M[i][j]. This required but nesting of std::array or std::vector. You can try to reproduce this behavior as an exercise – rewrite the Matrix class above such that M[i][j] correctly accesses a 1D array via arr[M*i+j]. Hint: Remember how we achieved this with std::arrays (part 1).

---

11Defining end in the initialization part of the for loop is a performance trick to avoid the call of size member
16.2.8 Iterators

Before we describe iterators, we want to extend our Array example from the previous part 2.

```cpp
#include <algorithm>
#include <initializer_list>
#include <iostream>
#include <iterator>

class Array {
public:
    Array(size_t i) : sz{i}, arr{new double[i]} {}  
    Array(std::initializer_list<double> l)
        : sz{l.size()}, arr{new double[sz]}  
            {  
            std::copy(l.begin(),l.end(),arr);  
    }  
    ~Array() { delete [] arr; }  
    double& at (int i) { return arr[i]; }  
    double& operator[](int i) { return arr[i]; }  
    int size() { return sz; }  
    double* begin() { return &arr[0]; }  
    double* end() { return &arr[(int)sz]; }  
private:
    size_t sz;  
    double* arr = nullptr;  
};

int main () {
    Array ar {1,2,3,4,5};  
    std::copy(ar.begin(),ar.end(),  
                std::ostream_iterator<double>(std::cout, " ");  
    std::cout << "\n" ;  
    std::fill(ar.begin(),ar.end(),5.);  
    std::copy(ar.begin(),ar.end(),  
                std::ostream_iterator<double>(std::cout, " ");  
    std::cout << "\n" ;
}
```

First of all, we extended the Array class by an initializer list constructor in line 8, such that we can initialize an array according to line 24. In the course of this, we changed the type of the class member sz into size_t (the largest possible unsigned integer type; see cppreference: std::size_t). A std::initializer_list has similar properties as a std::vector (see cppreference: std::initializer_list).

Second, we would like to use also the STL algorithms. In order to do this, we only need to define begin() and end() in lines 17 and 18, respectively. The reason why this works is because STL algorithms follow the pointer arithmetic semantics. (Look into the implementation of e.g. std::transform!)

**Short Intermezzo: Pointer Arithmetic**

We can define an array and pass through it (reading or writing) by

```cpp
double a[5] = {1,2,3,4,5};  
for(int i=0; i<5; ++i) a[i]=6;
```

This can also be accomplished by

```cpp
double a[5] = {1,2,3,4,5};  
std::fill(std::begin(a), std::end(a), 6);
```

which is supposedly more expressive in that it is written what it does. So, how does it work internally? Let us translate this into pointer arithmetic. a is the pointer name (address) pointing to the beginning of the array. This is equivalent to &a[0] – the address of the first element of the array a. If we want to access the second element, we must write &a[1].

12 This is but also equivalent to a+1. The 1 is function repeatedly in each loop round. For small loops as in this example it is of course exaggerated! But we wanted to illustrate, that the comma operator has many applications.

12Remember that addresses are those things, which pointers can hold and point to.
interpreted by the compiler to be times the size of double (because \(a\) is an array of double), such that \(a+1\) now points to the address of the second element.

Now, we want to read or write into the array fields, i.e. \(a[0]=5\) or \(a[1]=5\). In pointer semantics, that’s done via dereferencing, i.e. by the unary star operator \(*a=5\) and \(*a+1=5\) (remember the operator precedence? — therefore parenthesis).

Having this, we can rewrite the first approach above as follows.

```c
double a[5] = {1,2,3,4,5};
for(double* p=a; p!=(a+5); ++p) *p = 6;
```

or, in more C++ manner

```c
double * begin(double * a) {return a;}
double * end(double * a) {return a+5;}
void fill(double * begin, double * end, double value) {
    for(double* p=begin; p!=end; ++p) *p = value;
}
```

```c
int main() {
    double a[5] = {1,2,3,4,5};
    fill(begin(a), end(a), 6);
}
```

Of course, this here works only for built-in double arrays of size 5. We would like to have this more generic (later!). But we only outline the notion here. It would be cool, if we could get the size from the array itself, in order to define `end()`. But in native C, this cannot be accomplished. And also in C++, there is no way to figure out how large the memory is that a pointer is pointing to.\(^{13}\) This was the reason to introduce user-defined types and containers (like `std::array`), which keep this information in their record.

But the scheme above is appealing as it is less prone to errors, specifically out-of-range errors (and subsequent memory violation errors). Furthermore, it is more generic in the sense that algorithms can work in the same syntactic way on many types of containers. For integer arrays, we would have to write another such functions. But C++ can do this for us with templates – as will be shown in part 5.

Back to Iterators

What the pointer did above, can also be done by an iterator. The semantics is (mostly) the same. But iterators are user-defined types, and as such more general. They can be stateful, for instance, meaning that their inner state can evolve over time if necessary.

So, what does an iterator need? A copy constructor, assignment operator, dereference operator \(*\), comparison operators, and pointer member access \(->\) (because the iterator shall not only point to built-in types, but also to user-defined types, which possess member functions that shall be called). The last operation is skipped here, as there are difficulties involved to realize it with the means we currently have in hands.

\(^{13}\)The reason that `std::fill` works here in spite of this is that a built-in array is still different from a pure pointer.
bool operator==(const Iter& a, const Iter& b) { return a.p==b.p; }
bool operator!=(const Iter& a, const Iter& b) { return !(a==b); }

class Array {
public:
    Array(size_t i) : sz{i}, arr{new double[(int)i]} {} 
    Array(std::initializer_list<double> l) : sz{l.size()}, arr{new double[(int)sz]} {
        std::copy(l.begin(), l.end(), arr);
    } 
    ~Array() { delete [] arr; }

    double& at(int i) { return arr[i]; }
    double& operator[](int i) { return arr[i]; }
    int size() { return sz; }
    Iter begin() { return Iter{arr}; }
    Iter end() { return Iter{arr+sz}; }
private:
    size_t sz;
    double* arr = nullptr;
};

void print(const Iter& begin, const Iter& end, std::ostream& ofs) {
    for(auto p = begin; p!=end; ++p)
        ofs << *p << " ";
    ofs << "\n" ;
}

void print(Array& a, std::ostream& ofs) {
    for(const auto& p : a)
        ofs << p << " ";
    ofs << "\n" ;
}

int main () {
    Array ar {1,2,3,4,5};
    print(ar.begin(), ar.end(), std::cout);
    std::fill(ar.begin(), ar.end(), 5.);
    print(ar, std::cout);
}

So, we added an iterator (not full-fledged) for our double array – Array – with all necessary constructors and operators. As the iterator does not possess any resource – it just points to a resource – we can skip the destructor. The == operator needs to be a friend such that we can access the pointers of the Iters. But otherwise, the operators are binary and need to be non-members. Attempts to avoid friends completely is a bit more cumbersome, but possible. But at least by expressing != by ==, we could prevent that it also needs to be friend.\footnote{C++ is strange in that having less friends is better.}

For the Array class, only the begin() and end() members have changed to return now Iter (type). The rest stays the same.

The STL is more complicated, and thus std::copy could not be used unless we add something more to our iterator (interested reader: look for iterator traits). Other algorithms might exhibit the same issue. But we easily resemble the wished behavior by writing our own "copy" algorithm, which we but call print (as this is done here).

The abstraction, and even more the implementation are certainly overwhelming at the beginning. But as you have seen with the STL, well designed and implemented, it is really easy to use (and difficult to misuse), and it performs usually quite well – even on that higher level of abstraction.

Final Remark – Implementation

Not always will you need the full power of a fully-fledged iterator. And pointers fulfill already the semantics! All we may want is some sort of interface independence. This can be achieved as follows.
#include <algorithm>
#include <initializer_list>
#include <iostream>

class Array {
public:
    Array(size_t i) : sz{i}, arr{new double[(int)i]} {}
    Array(std::initializer_list<double> l) : sz{l.size()}, arr{new double[(int)sz]}
    {
        std::copy(l.begin(),l.end(),arr);
    }

    ~Array() {
        delete [] arr;
    }

double& at (int i) { return arr[i]; }

double& operator[](int i) { return arr[i]; }

int size() {
    return sz;
}

using iterator = double*;
using const_iterator = const double*;

iterator begin() { return arr; }
iterator end() { return arr+sz; }
const_iterator begin() const { return arr; }
const_iterator end() const { return arr+sz; }

private:
    size_t sz;
    double* arr = nullptr;
};

using Iter = Array::const_iterator;

void print(Iter begin, Iter end, std::ostream& ofs) {
    for(auto p = begin; p!=end; ++p)
        ofs << *p << " ";
    ofs << "\n" ;
}

void print(Array& a, std::ostream& ofs) {
    for(const auto& p : a)
        ofs << p << " ";
    ofs << "\n" ;
}

int main () {
    Array ar {1,2,3,4,5};
    print(ar.begin(),ar.end(),std::cout);
    std::fill(ar.begin(),ar.end(),5.);
    print(ar,std::cout);
}

Using the using aliasing introduced already earlier provides a convenient way. In lines 18 and 19, we define these iterator types, which are then used in lines 21 through 24. Actually, the rest stays roughly the same, except for the one print function. We gain by this that we can use this mechanism also for general containers. And e.g. std::vector and the other STL containers use it, too. Thus, the print function could be made a template function (see later), and used generally for any sort of container. Required is only that the iterators can be dereferenced, and that the contained objects can be printed, in this example.

16.2.9 Function Objects aka Functors

General Layout of a Functor

We can overload also the "function parameter" operator operator(). Let us consider this in action.

#include <iostream>

class Parabola {
public:
As the class `Parabola` contains a state – the current values of \( a \), \( b \), and \( c \) – instances of that class represent stateful functions. And we can use function objects as arguments for other functions – instead of the native C function pointers – because they are types. And now the clue: Lambda expressions ARE function objects!

Of course, a functor class can contain more member functions. One can write `operator()` with parameter lists of rather arbitrary types and argument numbers, and can even overload it several times for different argument lists and return values.

To comprehend the flexibility and the utilization of function objects you will probably need patience. We show more examples in the Hands-on exercises.

### Usage as a stateful Predicate for STL Algorithms

Function objects are the third (well, almost) way to describe predicates in STL algorithms. Imagine you have a vector of integers, and you want to find one even number.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

struct even {
    bool operator()(int i) {
        ++a;
        if(i%2==0) state();
        return (i%2)==0;
    }
    void state() {std::cout << "Needed " << a << " attempts!\n"; }
    private:
        int a = 0;
    }

int main() {
    std::vector<int> v {23,1,32,44,15,17};
    even e();
    auto res = std::find_if(v.begin(),v.end(),e);
    std::cout << *res << " is even\n";
    e.state();
}
```

It should be noted that \( e \) is copied into `std::find_if`. Therefore, the final `e.state()` results in 0.

We could construct an even object also in-place (unnamed). This is done via

```cpp
[...]
auto res = std::find_if(v.begin(),v.end(),even());
[...]
```

The `even` object is used by the `find_if` algorithm to evaluate the elements of the vector by means of `even's operator()(int)`, until the latter returns `true`. `find_if` returns an iterator, which we evaluate for the output.

**Exercise:** Use `std::generate` and a functor to fill a `std::vector<double>` with the exponential values of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

(See also Function Objects)
16.2. OPERATOR OVERLOADING BY EXAMPLE

16.2.10 Streaming Operators

We return once more to the DB_Item – introduced in the previous part 2. There, we struggled with handling them, i.e. filling them into our vector (database), and printing them. Furthermore, it was quite tedious to handle possibly many of them, sort them according to different criteria, or do more complex analyses using STL algorithms. This must be easier!

First of all, we want to read-in the items from a file! Hard-coding data inside a program is seriously out of fashion (and question)! Second, we want to read and print the items in a simple manner – something like input >> item;. We can solve both requirements at the same time. But we start with separate steps.

Let us agree on the following record format for our text file, which contains our database.

```
year (int)
title (string with white-spaces)
author (string with white-spaces)
item_type (string)
```

Let us agree on that the author is written lastname, firstname(s), for simplicity of illustration. Furthermore, the item type (CD, DVD, book, …) is written as human readable string, which must be converted. So, each record must contain four lines. If that’s not the case, we should capture this as error (at least writing out a warning that the database file is corrupted). This is the business of exceptions, one topic of the next part of the course.

Step one: We add streaming operator overloading. Because we defined DB_Item as struct, it is rather easy to access the class data members. We don’t need to make the operators << and >> friends.

Exercise: As an optional exercise, make DB_Item a class, and use getters and setters to access the respective data members. Then adapt the streaming operators!

The stream extraction and insertion operators have a certain signature – see cppreference: Stream extraction and insertion

```cpp
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

enum item_type {book, booklet, CD, DVD, unknown};

struct DB_Item {
    int year;
    std::string title;
    std::string author;
    item_type type;
};

// stream insertion
std::ostream& operator<<(std::ostream& os, const DB_Item& obj) {
    os << obj.year << "\n";
    os << obj.title << "\n";
    os << obj.author << "\n";
    switch(obj.type) {
        case book : os << "book\n"; break;
        case booklet : os << "booklet\n"; break;
        case CD : os << "CD\n"; break;
        case DVD : os << "DVD\n"; break;
        default: os << "unknown\n";
    }
    return os;
}
```

\[15\]If you want, you can separate them.
CHAPTER 16. OPERATOR OVERLOADING

```cpp
// stream extraction
std::istream& operator>>(std::istream& is, DB_Item& obj) {
    std::string year;
    std::getline(is, year);
    if(year.length() == 0) return is; // end of file
    obj.year = std::stoi(year);
    std::getline(is, obj.title);
    std::getline(is, obj.author);
    std::string It_type;
    std::getline(is, It_type);
    if(It_type == "book")
        obj.type = book;
    else if(It_type == "booklet")
        obj.type = booklet;
    else if(It_type == "CD")
        obj.type = CD;
    else if(It_type == "DVD")
        obj.type = DVD;
    else
        obj.type = unknown;
    return is;
}
using DB = std::vector<DB_Item>;
int main() {
    DB my_db;
    std::ifstream ifs("db.txt");
    DB_Item inputItem;
    while(ifs >> inputItem) {
        my_db.push_back(inputItem);
    }
    for(auto& p : my_db)
        std::cout << p ;
}
```

Please have once more a look on the signature of the stream operators. The first parameter must be a reference to a stream (istream or ostream). The second parameter is a (possibly const, for output) reference to the corresponding user-defined type. And for chaining, the return value is also a reference to the stream. Putting the object into the stream, or taking it from the stream, is then just using the stream operators as for the built-in types.

For the input, the end of file condition is a bit tricky. We chose to return is (with the state EOF), once the read year string is empty. There are certainly more possibilities to achieve the same thing.

Well, yes! There is a bit work to do for correct reading in a single DB_Item. However, the object orientation makes us focus only on reading such a single record – and not all records at a time. This latter namely is business and responsibility of another object (or function), when you remember the Single Responsibility Principle.

But we can do even better once we’ve accomplished this work, using the range-constructor of std::vector, iterators, and algorithms. The main() is now changed to the following.

```cpp
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
...
int main() {
    std::ifstream file("db.txt");
    DB my_db ((std::istream_iterator<DB_Item>(file)),
              std::istream_iterator<DB_Item>());
    std::copy(my_db.begin(), my_db.end(),
              std::ostream_iterator<DB_Item>(std::cout));
}
```

Everything in just three lines. Maybe it is interesting to hear that this is considered a very good
solution! In general, we agree. But our solution above is also comprehensible.

We have now reached a state, where it is easily imaginable to handle the problem to read-in a database, manipulating it (sorting, searching, filtering, ...) using the STL algorithms, and printing the results to the screen, or into a file, or actually into any stream we could imagine. Therefore, once more the hint: Try to stick with a high level of abstraction as much as possible!

**Exercise:** Use the Array class from the previous and this part, and

1) Add an assignment operator,

2) Add an iterator to it, a range-initializing constructor, and `operator<<` and `operator>>`. The goal is to read the array content from a file.

### 16.2.11 Overloading Operator `new`, and Advanced Memory Management

One can also overload operators `new` and `delete` for user defined classes. And accordingly the array variants. There are serious applications of this. One can placement construct objects ([cppreference: new expression](https://en.cppreference.com/w/cpp/language/new) and [Geeks for Geeks: placement new operator](https://www.geeksforgeeks.org/)), and thereby be much more memory efficient. This really gives you control over the memory allocation and usage. For cases, where you have to create and remove objects frequently (we may think of applications in a web server, where regularly user requests for an interactive web page appear), this can really be a performance boost, because allocating memory from the system is rather expensive – takes time. Practically, this means that you usually allocate much memory once, and distribute it via the operator `new`.

So, `new` is user-customizable. This business is advanced memory management, and is out of scope of this course.\(^\text{16}\)

However, a slight misuse of overloading operator `new` can create another interesting effect. In the last part, we’ve shown how an object can be forced to be placed into the free store (heap). Here is an example how to force it onto the stack. That is, this class cannot be instantiated via `new`.

```cpp
#include <cstddef>
class No_free_store {
    class Dummy;
    void *operator new(size_t, Dummy) {};
};
No_free_store glob2; // OK
int main() {
    No_free_store loc; // OK
    No_free_store* p = new No_free_store(); // error
}
```

The error occurs because there is no `No_free_store::operator new(size_t)`! We hid it by overloading it. And our operator `new` requires a second parameter, the type of which is but class internal – outside of the class, you cannot create a `No_free_store::Dummy` object, because it is private.

---

\(^{16}\)C++ knows another concept serving this purpose – Allocators (see e.g. [Geeks for Geeks: std::allocator](https://www.geeksforgeeks.org/)). But that becomes understandable only later after dealing with templates.
Chapter 17

Advanced I/O

By "advanced", we still mean rather basic in terms of simplicity. But applications are probably less frequent as they are more special to certain problems. For instance, binary I/O will become interesting only when you start to write large amount of data into a file.

17.1 std::cout, std::cerr and std::clog

The POSIX shell (the terminal or console) knows two output channels for a program – stderr and stdout. The latter is used for normal output to the terminal, for which we used std::cout. stderr is usually used for error and log messages. While stdout is buffered, which is usually better performing, because the buffer is only written when it is full (flushing), stderr is unbuffered. For error output the delay of buffered output is but not desirable. Therefore, writing into stderr is unbuffered. Accordingly, std::cout writes buffered, and writes to stdout, while std::cerr is unbuffered, and writes to stderr. There is a third such static object, std::clog, which writes to stderr, but in a buffered fashion (!).

So, for you to remember: normal output to std::cout. Error messages to std::cerr. And log information to std::clog.

17.2 Stringstreams

From a higher level of abstraction point of view, we have already noticed that it does not play much of a role for a C++ programmer whether she/he writes into an output stream to screen, or to a file. The same is true for reading. And this was the intent of the stream class design. The user (meaning the programmer who uses this library) should not care where a stream comes from, or where it goes to. Under this abstraction also fall all kind of network streams.

There is a special class called stringstream, where the back end is not a physical device, but just memory. You can write into or read from a stringstream in the same way as you write to or read from a file stream. A simple example (from cppreference: std::stringstream) may illustrate the usage.

```cpp
#include <iostream>
#include <sstream>
#include <string>

int main(){
    std::stringstream buf1;
    buf1 << 7;
    int n = 0;
    buf1 >> n;
    std::cout << "buf1 = \n" << buf1.str() << " n = " << n << 'n';

    // input stream
    std::istringstream inbuf("-10");
    inbuf >> n;
}
```
std::cout << "n = " << n << '\n';

// output stream in append mode (C++11)
std::ostringstream buf2("test", std::ios_base::ate);
buf2 << '1';
std::cout << buf2.str() << '\n';
}

std::stringstream can be used in a bi-directional way. Via << and >> you write to and read out from the stream, respectively. The str() returns the buffer string, which can be used for any other purpose.

If you have already a string, and you only need to read out from it in a formatted way, you can use std::istringstream. Via clear, you can clear the stream. And via str(string), you can reassign a new source.

#include <iostream>
#include <string>
#include <sstream>

int main() {
    int n;
    std::istringstream inbuf("-10");
inbuf >> n;
    std::cout << "n = " << n << '\n';
inbuf.clear();
inbuf.str("20");
inbuf >> n;
    std::cout << "n = " << n << '\n';
}

This can be used for easy conversion between strings and numbers, or other types.

There is also a dedicated output stringstream, std::ostringstream, where you can write into. The resulting string (buffer) can be obtained via str() member.

Of course, all stream manipulators and flags can be used also for stringstream.

### 17.3 Binary I/O

Currently, we wrote out and read in ASCII. For larger amounts of data that is highly inefficient. A number of 1.234560e+00 is of 12 bytes size – one byte per character. A normal double is but only 8 bytes (64 bits). So, writing this number in binary comprises not only a significant compression rate (substantially reducing also read and write times). It is also much more precise, because the full number is stored in the 8 bytes. The ASCII number literal above but has only 7 significant digits. So, for many applications, it makes sense to think about writing and reading in binary representation.

Binary operations have of course the disadvantages, e.g. that of hardly human-readability of binary files. Therefore, there is a decision to make.

An example illustrates the way binary read and write works – here as an implementation of copy (to copy files; no directories, access permissions are not taken care for).

#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
    if(argc!=3){
        std::cerr << "Syntax: " << argv[0] << " infile outfile\n";
        return 1;
    }
    std::ifstream ifs(argv[1],std::ifstream::binary);
    if(!ifs){
        std::cerr << "Specified input file does not exist!\n";
        return 2;
    }
    std::ofstream ofs(argv[2],std::ofstream::binary);

---

1In a program, you can obtain the size of a type by a macro, e.g. sizeof(double).
The char buffer is here used only as a temporary. So, the full relevance why it is a char array will not become clear, yet.

The most important thing here is the specification `std::ifstream::binary` or `std::ofstream::binary`. `seekg` sets the input position indicator (kind of a pointer) to a wished position – here end and beg. `tellg` just returns the input position indicator. Doing this at the end of the file stream, it returns the number of bytes – the file size.

Finally, you can read and write chunks (or all of the file at once) as char array – binary I/O performs only via bytes. In the example, we also explicitly closed both file streams again.

It is even possible to write instances (i.e. objects) of a class in a binary format, as long as a shallow copy suffices.

```cpp
#include <initializer_list>
#include <iostream>
#include <fstream>

class T {
public:
  T(int i=0, double d=0.0) : i{i}, d{d} {}  // Constructor
  int getI() const { return i; }  // Get the integer member
  double getD() const { return d; }  // Get the floating-point member
private:
  int i;  // Integer member
  double d;  // Floating-point member
};

int main() {
  std::ofstream ofs("test.bin", std::ios::binary);  // Create output stream
  double d = 3.1415;  // Define double variable
  std::cout << d << "\n" << std::endl;  // Output the double variable
  ofs.write(reinterpret_cast<char*>(&d), sizeof(d));  // Write double to file
  T t1{i3, 2.71};  // Create T object
  std::cout << t1.getI() << " " << t1.getD() << "\n" << std::endl;  // Output the T object
  ofs.write(reinterpret_cast<char*>(&t1), sizeof(t1));  // Write T object to file
  ofs.flush();  // Flush write buffer
  // or ofs.close();

  std::ifstream ifs("test.bin", std::ios::binary);  // Create input stream
  double dd;  // Define double variable
  ifs.read(reinterpret_cast<char*>(&dd), sizeof(dd));  // Read double from file
  std::cout << dd << "\n" << std::endl;  // Output the double variable
  T t2;  // Create T object
  ifs.read(reinterpret_cast<char*>(&t2), sizeof(t2));  // Read T object from file
  std::cout << t2.getI() << " " << t2.getD() << "\n" << std::endl;  // Output the T object
}
```

We declare a class `T` with built-in types as data members. You can try also to include user-defined types as members as an exercise. But those classes should not contain resources that are dynamically handled – so, `std::string` and `std::vector` will not work!

Reading and writing binary are low level operations. That is the reason why you see `reinterpret_cast<char*` – this casts the address of the objects into a character (byte) pointer, `char*` – the type `read` and `write` expect. Also, `read` and `write` require a number of bytes to be read or written, respectively. `sizeof` is exactly providing this.

In the example above, we needed to execute `flush()` (or, alternatively, `close()`) to flush the write buffer (in order that something is written to the file). Otherwise, the input stream immediately afterwards won’t find anything to read.
17.4 User-Defined Stream Objects

Using almost all of the experience so far, it should be easy to write an own stream object (at least a wrapper). That is a class, into which one can feed data. There is only a little thing left, which simplifies the live – templates. Just take note of it, otherwise ignore it, for now. So, the code review first.

```cpp
#include <iostream>

enum LogLevel {INFO=5, WARNING=3, ERROR=1};

class Logger {
public:
  Logger(std::ostream& ofs = std::cout) : ofs{&ofs} {}
  std::ostream* getStream() { return ofs; }
  Logger& operator()(LogLevel ll) {
    llevel=ll;
    return *this;
  }
  LogLevel getLogLevel() const { return llevel; }
  void setGlobalLogLevel(LogLevel globLL) {globalLogLevel=globLL; }
  LogLevel getGlobalLogLevel() const { return globalLogLevel; }
private:
  std::ostream* ofs;
  LogLevel llevel=ERROR;
  LogLevel globalLogLevel=ERROR;
};
template<
  typename T>
Logger& operator<<(Logger& logger, const T& t) {
  if (logger.getGlobalLogLevel() >= logger.getLogLevel())
    (*(logger.getStream())) << t;
  return logger;
}
static Logger log {std::cerr};

int main(int argc, char* argv[]) {
  log.setGlobalLogLevel(WARNING);
  log << 1 << "\n";
  log(ERROR) << 1 << "\n";
  log << 2 << "\n";
  log(WARNING) << 3.42 << "\n";
  log << 3 << "\n";
  log(INFO) << "Text\n";
  log << 4 << "\n";
}
```

The enumeration at the beginning defines our log levels. Goal is it that only those log data are written to the output stream, which are below the wished global log-level. We overloaded operator() to set the local log-level of the messages that follow. So, we can easily define whether a message is important (ERROR), less important (WARNING), or just information (least important). The user of the program can set the global log-level.

We had to overload the stream insertion operator, operator>>. But this time for our Logger. The semantics stays the same as for the normal stream objects (so also a reference is returned for chaining). We used here a template, in order to avoid writing dozens of such overloaded operators that differ only in the second parameter type. Templates will but be explained in a later part of the course.

Finally, we defined a static Logger (global) to which every function and class member can write (one needs to care with the names).

---

2So you may see at least one advantage of templates now.
**Exercise:** The customer comes and requests that before each message, we wants automatically also a label [ERR], [WRN], or [INF] to distinguish the different log messages. Furthermore, also a time stamp shall be included. Your job! What needs to be changed? Does it make sense to chain the streaming in this case?
Chapter 18

Debugging

Debugging (Wikipedia: Debugger) is – in the field of software engineering – the process of removing errors (bugs) from the code base, and increase the reliability and stability of the programs. That’s the goal!

The root cause for bugs is ALWAYS the programmer! (You will learn what this means!)

Now, there is a small problem. Because errors are not all equal. Errors can occur during compile time, link time, and run-time. Errors at run-time are most difficult to analyze. So, one should try to push errors to occur as early in the build and run process as possible.

Some errors are also intrinsically run-time bound. User input is eminent in this respect. Some errors are recoverable – others not. So, some errors can be anticipated by the programmer. This will be dealt with in the next part, when we talk about exceptions.

Not recoverable run-time errors usually do not simply crash the program. This namely would be the best thing to happen – a clear location in code that fails. This includes cases where your program does not even start. But Murphy won’t do us this favor. And so, more often than not, the crash happens erratically – once yes, another time not – or at different points of time after the program start. Or, the program does not crash at once, but continues to work – but showing unexpected behavior, or degradation of e.g. performance or responsiveness. This is often very hard to fix. Root causes might be memory leaks. But that is only one of many many other possibilities.

18.1 Compile and Link Time Errors

Syntax errors can be caught be the compiler. These are errors in the language use (missing semi-colons, opening or closing braces/parentheses, etc.). A compiler is very effective in that. Using -Wall and -pedantic-errors, you can make your (GCC) compiler very piggy even about "slight errors". -Wall makes the compiler complain about everything, although most things are just warnings. It is a good idea to remove as many as possible of these warnings. -pedantic-errors goes a step further, and transforms all warnings to errors. So, you could not compile until all warnings are handled and cured.

The compiler is clever in telling you if you do something unexpected for him. Prominent example is when the compiler finds declared but unused variables. Also, when adhering to const-correctness, you can help the compiler to identify problematic or critical code locations. At least, he can throw warnings.

It is also a good idea to tell the compiler which C++ standard to check for. If you want strict compliance, you should use the ANSI standard. But this may mean that you cannot use all the nice coding features the compiler might allow. That is the price you pay for producing code that is better portable, and easier to error check.

The linker usually complains mostly about undefined references. This means that symbols could not be resolved. In most cases, you probably only forgot to link-in a library. But in rare cases, you might have created variables or functions with external linkage, which never have be instantiated correctly.
Except for the symbol name, you have no other hint. That’s why it is so important to choose reasonable
names for variables and functions. Possibly, during debugging, it makes sense to NOT strip symbols
from the executable or library. Therefore, use the compiler option -g!

The opposite is also possible. Ambiguous symbols. This means, you have defined a function or
variable etc. twice (or, maybe not you, but it is there twice). And the linker is reluctant to decide
which to use. In such cases, you have to figure out in which translation units (possibly also in a library)
those symbols are defined. The decision about what to do – removing one of the translation units or
libraries, and other possibilities – is now upon you.

18.2 Run-time Time Errors

"Hello" Debugging

If the code system is not too large, and source code is available at all, then putting std::cout or
std::cerr statements in the right place can be very efficient. You can use this to print out variables to
screen for visual inspection. If you require to stop right after those statement, you can use std::exit()
from <cstdlib> (cppreference: std::exit), or std::abort() (cppreference: abort).

Also viable, as we have seen before in part 1, you can use assert from <cassert> (cppreference: assert), to check the correctness of variables. It has the advantage that when you compile your source
code with the option -DNDEBUG, the assert statements will be completely removed as if they had
never been programmed into the code. CMake sets this compile option for all but the Debug
build type.

Very helpful for "Hello" debugging are the three macro variables, __FILE__, __LINE__, and
__func__. They specify the file name, line number, and the function, where these variables are
located, and can be used for an orientation in the output. Example follows.

```cpp
#include <iostream>
void func() {
    std::cerr << __FILE__ << ":" << __LINE__ << ":" << __func__ << "\n";
}

int main() {
    std::cerr << __FILE__ << ":" << __LINE__ << ":" << __func__ << "\n";
    func();
}
```

Test Driven Development

Once your program compiles, you have passed the very first hurdle. Next is run-time error checking.
One thing you can do is to run the tests with your compiled executable. Sure, it does not mean that
everything is fine when the tests all pass. Depending on your test suite, this might but make it less
probable.

If you caused an error, or unexpected behavior, which is not caught by your tests, you need to
improve your tests. But as usual, when you create the tests before writing the code implementation to
make the code compile, this should not happen.

To be sure that your test suite covers and tests all you code, you can use e.g. gcov to test the
coverage. It should be larger than about 75%.

1 Advanced capabilities lie in static_assert – cppreference: static_assert – which checks assertions during compile time,
and interrupts the compilation on errors.
2 Upper and lower case writing must be correct! And __func__ is available starting with C++ 11.
18.2. **RUN-TIME TIME ERRORS**

**Other Analysis Tools for Run-time Checks**

If the above does not help, the last resort is usually only hard work with the following or similar tools.

- **`nm`**: list symbols from object files – good for checking static symbol table *(nm (Linux man page))*
- **`ldd`**: print shared library dependencies – good for checking dynamic/shared library dependencies *(ldd (Linux man page))*
- **`size`**: list section sizes and total size – good to check the size of the different sections in executable *(size (Linux man page))*
- **`readelf`**: displays information about ELF files – amongst others check RPATH settings of an executable/shared library *(readelf (Linux man page))*
- **`objdump`**: disassembler – good for assembly code analysis *(objdump (Linux man page))*

```bash
$ objdump -d <binary executable|object file> | less
```

- **`strace`**: shows verbose stack and other information – good for crashes *(strace (Linux man page))*
- **`gdb`**: GNU Debugger: lets you run your program interactively – good for all kinds of error analyses *(GDB: The GNU Project Debugger)*
- **`valgrind`**: good for memory leak checking, and many other things *(Valgrind Homepage)*

This list is surely not comprehensive. And an introduction into the more elaborate tools like `gdb` and `valgrind` would vastly exceed the scope of basic C++ course. But it might suffice as an entry point once you face problems with erroneous programs. 

---

3If you want to test these tools, the VM might contain them, or can be installed.
Chapter 19

Hands-On Exercises for Part 3

19.1 Visualization

19.1.1 Image Class – PPM File Format

For diverse applications, it is desirable to have a library for creation of images – both as data representation and files. One of the simplest such formats is the PPM file format (Wiki: Netpbm format or ppm). We are interested in the binary form of PPM. Its format is given as follows.

```
P6
x-dimension y-dimension
255
```

P6 is just a magic number, marking the format as color image. The x- and y-dimensions are just the pixels in those dimensions. 255 is just the maximum color value (< $65536$). The rest of the file is then filled with binary data of the colors of the pixels, encoded in red-green-blue (RGB) sequence (unsigned char == 256 values = 8 bit). The image is rendered from top left to bottom right line-wise.

**Task:** Create your own Pixel Graphics Class!

**Hint:** First construct a class for the data structure of colors – e.g. Color. Think about the design of this class! Which class member needs it? Could we use some operators to overload?

Next, design the image class – e.g. XYimage –, one constructor of it takes the width and height of the image in pixels! Add class member functions to set single pixel’s colors, and to print the image to a PPM-file! Which data structure should carry the data?

Test your implementation by creating a simple gradient image! At success, you should be able to display the image via

```
$ display file.ppm
```

19.1.2 Image Class – Color Map, Mandelbrot Set

**Task:** Test the image class above with a more elaborate example! For this purpose consider the so-called Mandelbrot set. It is created as follows. Start with a complex number $z_0 = 0$, and another arbitrary complex number $c$. Perform the following iteration $z_{n+1} = z_n^2 + c$. Depending on $c$, this sequence will converge or diverge. As we cannot iterate until infinity, there is a simpler abortion criterion: Try at maximum $N$ times. If $|z_{n+1}| > 2$ return $(n + 1)/N$, otherwise $-1$.

One now can colorize the pixel around the position $c$ in the complex plane according to some color
map, which maps the return value of -1, 0, ..., N to RGB (so to speak, a scalar to a three-component vector).

**Hint:** Use the header `complex` to include complex numbers. They are declared by e.g. `std::complex<double>` as double-valued complex numbers. For them, the arithmetic operators are overloaded. So, two complex numbers `z1` and `z2` can be added, multiplied, etc. Furthermore, functions like `std::abs` for the absolute value of a complex number, and those to obtain real and imaginary parts are defined (see `cppreference: complex`).

### 19.1.3 Paraview File I/O (Tutorial)

Paraview is a free and open-source GUI-based scientific visualization data analysis tool, based on the Visualization Toolkit (VTK), which itself is open-source C++ class framework for scientific visualization. The Links before refer to extended documentation. And we can only recommend you to take the time to learn how to use this tool. Paraview can be downloaded as pre-compiled binaries from the Paraview webpage for the three most dominant operating systems. For this tutorial we recommend version 5.6.0.

Paraview can handle many file formats to read. And further so-called *Readers* can be added. Although this would be a nice exercise in C++, too, we would like to restrict ourselves here to VTK’s own (legacy) file format for simple cases – structured point grids. Such a grid is essentially a Cartesian regularly spaced grid of point values. An example `.vtk` file of that type looks as follows.

```plaintext
# vtk DataFile Version 2.0
Some Comment here, max 256 chars
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 3 4 6
SPACING 1 1 1
ORIGIN 0 0 0
POINT_DATA 72
SCALARS volume_scalars double 1
LOOKUP_TABLE default
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 5 10 15 20 25 30 35 40 45 50 55 0 0 0 0 0 0
0 10 20 30 40 50 60 70 80 90 100 110 0 0 0 0 0 0
0 10 20 30 40 50 60 70 80 90 100 110 0 0 0 0 0 0
0 5 10 15 20 25 30 35 40 45 50 55 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
```

The first line is mandatory – the version could be changed (when you know what you do). The second line is a comment (max. 256 characters). The third line says that we store the data in ASCII format. BINARY is also feasible, as shown below.

Our data sets are STRUCTURED_POINTS, meaning essentially a regularly spaced Cartesian point distribution.

**DIMENSIONS** specify the number of points in x-, y-, and z-direction. Their product must be equal to the value of **POINT_DATA**. If one of the **DIMENSIONS** is 1, we have 2D data. Accordingly, two values of 1 imitate 1D data.

**SPACING** tells which distance the points in x-, y-, and z-direction have. The **ORIGIN** is the coordinate of the very first point. **SCALARS** has three parameters. The name as is represented by Paraview, the data type (here double, meaning 64-bit floating point), and the number of components. We use the simplest default, 1.

The data are written in the order x-direction first, y-direction second, and z-direction last. So, each of the six lines above corresponds to a different z-coordinate.\(^1\) Within each line, the first three values

\(^1\)There doesn’t need to be line breaks, but! It is for illustration only!
are values of our field for the three x-coordinates and the first y-coordinate. The next three values are those of the three x-coordinates and the second y-coordinate, etc.

To enter numbers is tedious. So we write a program, doing this for us.

```cpp
# include <fstream>
# include <cmath>
#define N 41
#define dx 0.1

double f(int i, int j, int k) {
    double x = (i/(double)(N-1)-0.5)*dx;
    double y = (j/(double)(N-1)-0.5)*dx;
    double z = (k/(double)(N-1)-0.5)*dx;
    return x*x*std::sin(50.*y)*std::exp(10*z);
}

int main() {
    std::ofstream ofs("functiontest.vtk");
    ofs << "# vtk DataFile Version 2.0\n" << "Some Test File\nASCII\n" << "DATASET STRUCTURED_POINTS\n" << "DIMENSIONS " << N << " " << N << " " << N << "\n" << "SPACING " << dx << " " << dx << " " << dx << "\n" << "ORIGIN " << -(N-1)/2.)*dx << " " << -(N-1)/2.)*dx << "\n" << "POINT_DATA " << N*N*N << "\n" << "SCALARS some_vol_data double 1\n" << "LOOKUP_TABLE default\n";
    for(int z=0; z<N; ++z)
        for(int y=0; y<N; ++y)
            for(int x=0; x<N; ++x)
                ofs << f(x,y,z) << " ";
    ofs << "\n";
    ofs.close();
}
```

This solely creates a file `functiontest.vtk`, which can be opened using `paraview`. The result (with some analysis, and another field added to the data for the contour coloring) is shown in Figure 19.1. For

![Figure 19.1: Paraview GUI to analyze the resulting data file.](image)
another field to bring in, just enter another section starting with SCALARS.

Using BINARY data (which results in much smaller files usually), we need to apply some trick, because of the Endianness business in Paraview.

```cpp
void SwapEndianness(double& var) {
    char* varArray = reinterpret_cast<char*>(&var);
    for(long i = 0; i < static_cast<long>(sizeof(var)/2); i++)
        std::swap(varArray[sizeof(var)-1-i],varArray[i]);
}
int main() {
    std::ofstream ofs("functionbintest.vtk");
    ofs << "# vtk DataFile Version 2.0\n" << "Some Test File\nBINARY\n" [...]
    ofs << "LOOKUP_TABLE default\n";
    for(int z=0; z<N; ++z)
        for(int y=0; y<N; ++y)
            for(int x=0; x<N; ++x) {
                double val = f(x,y,z);
                SwapEndianness(val);
                ofs.write(reinterpret_cast<char*>(&val),sizeof(double));
            }
    ofs << "\n";
    ofs.close();
}
```

SwapEndianness does the right job for us. If you do not understand it, don’t worry. Just copy!

**Task:** You can familiarize with that program to produce data, and analyze them with Paraview. Try also 2D data!

**Task:** Create a data-class (say, sort of an array class), with a class member function, which writes out the data into the VTK format!

### 19.2 Algebra

#### 19.2.1 Operator Overloading for Matrices and Vectors

**Task:** In the previous part’s Hands-on, we introduced square matrices as classes, together with member functions, which perform operations like adding or multiplying two matrices, or applying a matrix to a vector.

Realize this now using operator overloading. A code like the following should compile!

```cpp
#include "Matrix.h"
#include <iostream>
int main() {
    Matrix A {...}; // fill reasonably!
    Matrix B = transport(A);
    Matrix C = A + B;
    Matrix D = A * C;
    std::cout << "A = " << A << "\n";
    std::cout << "B = " << B << "\n";
    std::cout << "C = " << C << "\n";
    std::cout << "D = " << D << "\n";
    Vector v = {...}; // fill reasonably!
    std::cout << "A*v = " << (A*v) << "\n";
    std::cout << "v^T*A*v = " << (v*(A*v)) << "\n";
}
```

The latter bilinear form is certainly useful when writing more complex functions like e.g. a multidimensional Gaussian.
19.2. ALGEBRA

Hint: Think whether the overloaded operators should be class members or free functions! Also think about the parameter lists and the return values! Let you guide by intuition about the semantics of the operations (and their mathematical properties like commutativity).

What do you think happens to expressions like these?

\[ D = A + B \ast C; \]
\[ D = (A + B) \ast C; \]

Will they be handled correctly?

19.2.2 CERN ROOT Linear Equation Solver (Tutorial)

Let’s consider a simple 1D boundary differential equation problem.

\[
\frac{d^2 \phi(x)}{dx^2} = -\sin(\pi x) \quad \text{on} \quad x \in [0, 1], \quad \phi(0) = \phi(1) = 0.
\]

In order to solve this, we discretize this equation (in later parts more on that) – we use finite differences. First, we select \( N + 1 \) discrete \( x \)-values,

\[ x_i = i \cdot \Delta x, \quad i = 0, \ldots, N + 1, \quad \Delta x = \frac{1}{N+1}. \]

We approximate the second derivative above by

\[
\frac{d^2 \phi(x)}{dx^2} \approx \frac{\phi(x_{i+1}) - 2\phi(x_i) + \phi(x_{i-1})}{(\Delta x)^2} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{(\Delta x)^2}.
\]

So, the equation to solve is now a linear matrix equation.

\[ \phi_{i+1} - 2\phi_i + \phi_{i-1} = -(\Delta x)^2 \sin(\pi x_i) \equiv b_i. \]

The essential code in ROOT is the following.

```cpp
#include "TArrayD.h"
#include "TDecompLU.h"
#include "TMath.h"
#include "TMatrixD.h"
#include "TVectorD.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TAxis.h"
#include <iostream>
#include <vector>
#define N 50
int main() {
    using namespace std;
    TArrayD d(N*N);
    d.Reset(0.0);
    for(int i=0; i<N; ++i) d[i+N*i] = -2.;
    for(int i=1; i<N-1; ++i) d[i+1+N*i] = d[i-1+N*i] = 1.;
    TMatrixD Am(N,N);
    Am.SetMatrixArray(d.GetArray());
    const double dx = 1./(N+1.);
    TVectorD rhs(N);
    for(int i=0; i<N; ++i) // Init RHS
        rhs[i] = -dx*dx*TMath::Sin(TMath::Pi()*(i+1.)*dx);
    TVectorD fi(N);
    fi.Zero(); // Init solution
    TDecompLU lu(N);
    lu.SetMatrix(Am);
    std::cout << " Start LU Decomposition\n";
    lu.Decompose(); // <-- LU Decom
    cout << " Solve Equation\n";
}
The only complication is that because of the boundary conditions, the matrix equation is of dimension \( N - 1 \). So, we have to do a bit mapping. Finally, we draw the solution (is easier than to print out the numbers). The result is shown in Figure 19.2. Btw., it’s indeed true: \( \frac{d^2}{dx^2} \sin(\pi x) = -\pi^2 \sin(\pi x) \), i.e.

\[
\phi = \frac{\sin(\pi x)}{\pi^2}.
\]

**Task:** Use another differential equation, and solve it using the method above.

**Task:** Of course, you can use this also if you need to solve just a linear matrix equation (e.g. least squares fitting). ROOT offers but more. Investigate the singular value decomposition and the matrix inversion!

### 19.2.3 Function Object Algebra

In mathematics, we use to define composition of functions of the same type (meaning to map from the same space into the same space) point-wise. That is,

\[
f(x) = g(x) + h(x)
\]
is defined for each valid value \( x \). But functions can also be entities of their own, being objects (vectors) of a function space. That is,

\[
f = g + h
\]

based on e.g. the above point-wise definition has its own right of existence.\(^2\)

But also in C++ (and other programming languages), functions can have these two aspects. Defining a function and evaluating it happens in different places.

**Task:** Using your knowledge about Lambda functions, `std::function`, and now also function objects (i.e. classes with `operator()` overloaded), realize a function algebra. That is, if \( F_1 \) defines such a function object type, we want that code like the following compiles and works.

```cpp
int main() {
    F1 f1 {[[](double x){return x*x;}];
    F1 f2 {[[](double x){return x*x*x;}];
    F1 f3 = f1 * f2;
    F1 f4 = f1(f2);
    std::cout << " f1(2) = " << f1(2) << \n;
    std::cout << " f2(2) = " << f2(2) << \n;
    std::cout << " f3(2) = " << f3(2) << \n;
    std::cout << " f4(2) = " << f4(2) << \n;
}
```

That is, the function objects shall be added, subtracted, multiplied, and divided (by numbers or other functions). And we would like also to have a possibility to compose functions by evaluating them in a nested way like \( f4 \) above.

### 19.3 STL Algorithms – Functors instead of Lambda Functions

**Task:** In the previous part’s hands-on session was an exercise for using STL algorithms to manipulate a `std::vector<Person>`. Realize the same procedures with function objects (aka functors)! What do you notice?

### 19.4 Numeric

#### 19.4.1 Boost.odeint – ODE Solver

At the [Boost odeint web page](http://www.boost.org/doc/libs/1_64_0/libs/odeint/doc/html/index.html), you can find a short example of the boost ODE integrator (odeint). It solves the equations of the damped harmonic oscillator,

\[
\ddot{x} - \gamma \dot{x} - x^2 = 0,
\]

aka

\[
\begin{align*}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= -x_1 - \gamma x_2,
\end{align*}
\]

with \( \gamma = 0.5 \), and with initial conditions \( x_1 = 1, x_2 = 0 \) from time 0 to time 10, with initial time step 0.1 (integrate is based on and adaptive time step Runge-Kutta method).

---

\(^2\)This is the field of functional analysis.
CHAPTER 19. HANDS-ON EXERCISES FOR PART 3

**Task:** Get this example compiled and running! Try to understand what every line of code is doing!
If you managed to do so, try to implement the Lorenz Attractor:

\[
\begin{align*}
\dot{x}_1 &= 10(x_2 - x_1) \\
\dot{x}_2 &= x_1(28 - x_3) - x_2 \\
\dot{x}_3 &= x_1x_2 - \frac{8}{3}x_3
\end{align*}
\]

with the initial condition at \( t = 0 \): \( x_1 = x_2 = x_3 = 1 \). Written to a file (e.g. \textit{lorenz.txt}) with 4 columns (time, \( x_1 \), \( x_2 \), \( x_3 \)), \textit{Gnuplot} can be used also here to visualize the result:

```
gnuplot> splot 'lorenz.txt' u 2:3:4 w l
```

**Hint:** ODEINT solves general systems of the form \( \dot{x} = f(x) \), where \( x \) is a vector, and \( f(x) \) a rather arbitrary (Wiki: Picard-Lindelöf Theoreme) vector valued function. So any differential equation with higher time-derivatives must be transformed into such a first-order ODE.

An interesting variant for linear systems (e.g. radioactive decay systems) is to use your own vector class (see above), and perform the right-hand side as matrix-vector product.

Another interesting ODE use cases are \textit{Lodka-Volterra models} for predator-prey relationship, or Newtonian systems, like e.g. a point charge released in a stationary \((1/r)\) electrostatic potential (by motion, a magnetic force will be exerted onto the point charge additionally).

**Hint:** For compilation with CMake, look into the Appendix of Part 1, where a \textit{CMakeLists.txt} file is shown!

### 19.4.2 Aerodynamics and Complex Numbers (Tutorial)

**Background – CFD and Theory of complex Functions**

For stationary, laminar, (effective) 2D flows (Wiki: Potential Flow), two equations can be derived for the velocity, \( \vec{v} \), out of the \textit{Navier-Stokes Equations}.

\[
\begin{align*}
\text{div} \vec{v} &= \nabla \cdot \vec{v} = 0 \\
\text{rot} \vec{v} &= \nabla \times \vec{v} = 0
\end{align*}
\]

This allows for the introduction of a potential \( \phi \), such that (from the irrotationality) the velocity can be obtained as gradient of that potential.

\[
\vec{v} = \nabla \phi.
\]

Using also the divergence equation, we obtain the equation for \( \Phi \),

\[
\Delta \phi = 0.
\]

We put the scenario (2D – you remember?) into the \((x, y)\)-plane, i.e.

\[
\vec{v} = \begin{pmatrix} v_x(x, y) \\ v_y(x, y) \\ 0 \end{pmatrix}.
\]

Then the divergence equation can be written as

\[
\Delta \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0.
\]

So, \( \phi \) is supposed to be a harmonic function.

---

\(^3\)If this is completely unknown to you, please stop here, and go to another exercise!
As is maybe known, the 2D space (plane) can be uniquely mapped on to the plane of complex numbers, \( \mathbb{R}^2 \rightarrow \mathbb{C} \) via \((x,y) \mapsto x + iy\). Complex numbers can be used in C++, too (see \texttt{cppreference: complex}). For example.

```cpp
#include <complex>
#include <iostream>

std::complex<double> f(std::complex<double> z) {
    return sin(z);
}

int main() {
    std::complex<double> z1 {1.,5.};
    std::cout << " z1 = " << z1 << "\n";
    std::cout << " Re(z1) = " << std::real(z1) << "\n";
    std::cout << " Im(z1) = " << std::imag(z1) << "\n";
    std::cout << " |z1| = " << std::abs(z1) << "\n";
    z1+=4.;
    std::cout << " z1 = " << z1 << "\n";
    std::complex<double> z2 {4.,2.};
    std::cout << " z2 = " << z2 << "\n";
    std::cout << " z1+z2 = " << z1+z2 << "\n";
    std::cout << " z1*z2 = " << z1*z2 << "\n";
    std::cout << " z1/z2 = " << z1/z2 << "\n";
    std::cout << " exp(z2) = " << exp(z2) << "\n";
    std::cout << " f(z2) = " << f(z2) << "\n";
}
```

Now, without derivation, we introduce a complex potential,

\[
\Phi = \phi + i\psi .
\]

where \( \phi \) is the velocity potential introduced above. \( i = \sqrt{-1} \) is the imaginary unit. Furthermore, we rewrite the velocity as complex velocity

\[
V = v_x + iv_y .
\]

The gradient relation above, \( \vec{v} = \nabla \phi \), is then expressed as

\[
V = \left( \frac{\partial \Phi}{\partial z} \right) .
\]

In order that this works, \( \phi \) and \( \psi \) must fulfill the so-called Cauchy-Riemann Equations,

\[
v_y = \frac{\partial \phi}{\partial x} = \frac{\partial \psi}{\partial y} , \quad v_x = \frac{\partial \phi}{\partial y} = -\frac{\partial \psi}{\partial x} .
\]

We assigned here also the velocity components. Entering one of the equations into the other, we immediately see that \( \phi \) and \( \psi \) are harmonic, i.e. they fulfill also \( \Delta \phi = 0 = \Delta \psi \). And now the good thing: Any analytic (holomorphic) function preserves this property. We will use this to construct flow patterns by this. We only need to determine before what is \( \psi \). For a good reason, it is called the Stream Function, because lines of constant values of that function are stream lines. Some examples will illustrate this.\(^4\)

**Uniform Flow:** \( \Phi = v_0 e^{i\psi_0} z \). \( v_0 \) is the velocity magnitude, and \( \psi_0 \) the angle of attack of the flow. This potential represents a uniform laminar flow profile. Figure 19.3 shows in the left-most plot.

**Source/Sink:** \( \Phi = k \ln(z - z_0) \). \( k \) is the strength, \( z_0 \) is the location of the source/sink.

**Circulation:** \( \Phi = ik \ln(z - z_0) \). \( k \) is the strength, \( z_0 \) is the location of the center of the circulation.

\(^4\)We use here ROOT for visualization.
Dipole Flow: \( \Phi = \frac{1}{2} z \). Can be used for electrodynamics. The ROOT script to create the single flow patterns is shown in Appendix.

More complex flow patterns can be achieved by superimposition. For instance,

\[
\Phi = \nu_0 \left( z + \frac{r_0^2}{z} \right) + ik \log z ,
\]

\( r_0 \) is the radius, \( \nu_0 \) the speed at infinity (\( \nu_\infty \)), \( k \) the circulation (usually written using the letter \( \Gamma \)), which is related to a lift-force (per unit length) via the Kutta-Joukowski Theorem (aka Magnus Effect),

\[
F_L \sim -\rho_0 \nu_0 k ,
\]

perpendicular to the uniform velocity direction. Figure 19.4 shows examples of this flow pattern. \( \rho_0 \) is the density at infinity. The ROOT script is shown in Appendix.

Joukowski Transform

There is still a last step to do. For a wing of an airplane, we need other shapes. A cylinder won’t be helpful. Joukowski investigated for this certain transforms (see Wiki: Joukowski Transform), which looks as follows.

\[
z = \frac{1}{2} \left( \zeta + \frac{1}{\zeta} \right) .
\]

Just to show its effect in Figure 19.5, we map circles in the \( \zeta \)-plane to objects in the \( z \)-plane. The code can be found in the Appendix. So, obviously, we need circles with an center point \( M = (-\delta, \delta) \) and
a radius \( r_0 = 1 + \delta \) such that the lower and right boundaries of the circle are at \( y = -1 \) and \( x = 1 \), respectively.

**Putting it all together**

A flow around a cylinder (aka circle), we know already, also with a circulation (which we will need to set the correct stagnation points). We map this flow then via the Joukowski Transform, which is *conformal*. That is, the properties of of the mapped flows will also be preserved (especially the circulation, and thus the lift force).

We start with the potential.

\[
\Phi = v_0 \left( z e^{i\varphi_0} + e^{-i\varphi_0} \frac{r_0^2}{z - z_0} \right) + i k \log(z - z_0),
\]

here, \( z_0 \) being the center point of the circle, \( z_0 = -\delta + i\delta \). \( r_0 = 1 + \delta \). \( v_0, \varphi_0 \) and \( k \) are still free parameters. The procedure is as follows. First look at one \( \zeta \). Invert the Joukowski mapping to obtain

\[
\Phi(z) = \Phi(\zeta),
\]

and take the imaginary part from the result, which gives the streamlines. Adapt \( k \) so long until the rear

...
stagnation point is at the position of the airfoil’s rear cusp, aka Kutta Condition. Figure 19.6 shows the result, and the source code for this analysis can be found in the Appendix.

**Task:** One could now determine the lift force of the wing as a function of the angle of attack.

**Task:** The solution in the Appendix was written using normal functions. Realize them using function objects (and/or Lambda expressions)!

## 19.5 Debugging Session

We have prepared a very buggy small program, which is supposed to create compile-, link-, and run-time errors. The simple task of this program is to create a file data.dat with two columns – one the abscissa values, the other the function values. Goal might be to plot this afterwards using gnuplot.

**Task:** Use all your skills to get it compiled and running! You can also use GDB for the run-time debugging.

```cpp
// main.cxx
#include "my_sqrt.h"
#include <algorithm>
#include <cmath>
#include <fstream>
#include <vector>
int main() {
    using namespace std;
    vector<double> v1(200), v2;
    generate(begin(v1), end(v1), [d = -10.] () mutable { return d+=0.1; });
    transform(begin(v1),end(v1),begin(v2),
                [](double x){ return sin(x)/x+my_sqrt(x);});
    std::ofstream ofs("data.dat");
    for(int i=0; i<v1.size(); ++i)
        ofs << v1[i] << " \t " << v2[i] << "\n";
}

// my_sqrt.h
#ifndef MY_SQRT_H
#define MY_SQRT_H
double my_sqrt(double x);
#endif

// my_sqrt.cxx
#include "my_sqrt.h"
#include <cmath>
double my_sqrt(double x) {
    return std::sqrt(std::fabs(x));
}

# CMakeLists.txt
cmake_minimum_required (VERSION 3.5)
project (Debugging)
add_executable (data_gen main.cxx )
```

**Hint:** If you get a memory access violation error or similar (also called segmentation fault), you can use

```
$ ulimit -c unlimited
```
to increase the size of the coredump file. This core file can be investigated via

$ gdb ./data_gen core

As GDB usage reference, you can use the GDB Cheat Sheet. A small tour through GDB is attached to the solution in the Appendix.
Part IV

Inheritance, Exceptions, Performance
Chapter 20

Inheritance

20.1 Introduction

Without much fuss: Inheritance is THE capability to express one class (the derived class) in terms of another class (the base class). Inheritance is often considered as THE object orientation feature. IMHO, that is not necessarily the case. And an overuse of this feature can become a torture for maintainers and users of libraries with deep inheritance hierarchies.

True is though that inheritance is another way to extend the capabilities of persisting class hierarchies, with all the advantages of code reuse and other software engineering principles in mind. Furthermore, inheritance can be used to describe hierarchical relations in the sense that more specialized classes are derived from more general or abstract ones. For instance, a car, a ship, and an airplane are all devices for transport. All can move. And all have a shape and a color, for instance. In certain moments, the function may be the only interesting fact. That is, it may only be interesting that they can move and e.g. transport persons or cargoes. Nonetheless, cars, ships and planes are different in other details. Cars move in another way than ships and planes do – the propulsion principles are different. In detail, cars are also build differently with four wheels and no airfoils and so on.

However, “Car” itself can again be an abstract concept, in turn. We know a lot of different types of cars, which differ in details of construction and intended functionality. So, the following tree of vehicle concepts represents a hierarchy tree, where the depth represents the levels of abstraction/concretization.

![Inheritance Hierarchy](image)

Figure 20.1: A simple inheritance hierarchy.

Related to inheritance are but also some conceptual and technical difficulties – not only for beginners. C++ considers inheritance as a mean of polymorphism – one object can behave like another one. So, an object of a derived class can behave like an object of its base class. We have learned already

---

1 See Sean Parent’s Talk on “inheritance the base class of evil”, which is entertaining, but meant seriously – but it is in part quite advanced.

2 If transport vehicles are too far from your business, you may think about representation of computer devices such as memory (abstract), and hard-disk, RAM, caches, etc. (special). Or if you are biologist, you may consider birds, mammals
a polymorphism before – function and class member function overloading: One function (name) can
behave differently depending on how it is called (parameter list). This kind of polymorphism happens
during compile time, i.e. the compiler decides already during the compilation, which function to use.
In contrast, the member function overriding of inheritance is a run-time polymorphism. This means
that a derived class can re-implement a member function with the same name and parameter list that
is already existing in the base class. And which function is called is decided during run-time(!) via the
so-called dynamic virtual function dispatching mechanism. The decision is based on the dynamic (in
oppose to the static) type of an object. This can be very hard to comprehend at the beginning, and
can result in some surprising and unintended effects. Be patient!

When does it make sense to consider inheritance at all? A general answer is hardly possible. But
one can consider more or less two perspectives – the class designer’s and a class user’s one. Both cases
are more or less examples of the so-called public inheritance.

1) As a user, you may want to extend another class. This means that certain functionality is already
there in a class, which you want to reuse. But your new class is sort of specialization or extension
of the class from which yours is inherited. Or, you want to share certain data through the base
class. This is sometimes called implementation inheritance, and aims largely to code-reuse.
Nowadays, implementation inheritance is not considered a good design. Nonetheless, for intro-
ducing the syntax and rules, it is a good starting point.

2) A library writer may want to offer a defined interface for users, where either some part of
functionality is already implemented, or where the user of the library shall add functionality
specific to his/her needs – but still in special compliance necessary for the library to work. This
is sometimes called interface inheritance. The more pronounced goal is design guidance for the
user. In extreme cases, no implementation or data are available at all in the base class, and only
the interface is inherited.

There are so-called abstract classes, which have pure virtual functions. No instances of such a
class can be created. As a classical example, we use geometry. A rendering library supplies a
class Shape, which may have the functionality that it can be drawn (i.e. has a class member
function draw()). But Shape is just abstract – so shall not be instantiated. The users of that
library shall extend the library’s functionality by creating special shapes like e.g. in 2D a circle,
a triangle, a square, etc. But for each derivative (each concrete geometrical shape), concrete
class members must be defined appropriate for the concrete shape, and the inherited draw() function
must be overridden (such that the library can call draw() for each of the concrete shape
objects). That is, it must be implemented accordingly based on the concrete class data members
(a circle is drawn differently than a square, etc.).

In both cases, the essential driver to use inheritance is the already mentioned run-time polymor-
phism, including the usage of the virtual class member function overriding. One should carefully check
whether that’s really necessary, because of possible expense in terms of complexity and performance.
But where it is applicable and necessary, inheritance can indeed be a superior feature, which should
not carelessly neglected either.

Note! While objects are instances (realizations) of a class/struct (as pattern or scheme), inheritance
plays fully in the field of the patterns (schemes) themselves. Depending on the implementation, you
may have objects/instances either of base type, or of derived type, or of both. In this sense, inheritance is
orthogonal with respect to instantiation.

and fish as animals, in general, while sparrows and swallows are special kinds of birds.

3We will learn a third type of polymorphism in the last part of this course: function and class templates.

4Access permission specification is another degree of freedom in inheritance. We will later note on the other inheritance
modes, of which private inheritance is still sometimes occurring. Public inheritance is by far the most often used mode.
20.2 Simple Public Inheritance – IS-A Relation

Let us start considering how it works. We introduce a class named `Shape`, which shall be an abstract class with a 2D render library. (We use output to `std::cout` to mock the result of actions.)

```cpp
#include <iostream>
class Shape {
public:
    void draw() const {std::cout << "draw shape\n";};
    void move(double x, double y) {std::cout << "move shape\n";};
    void rotate(double angle) {std::cout << "rotate shape\n";};
};
```

We have three class member functions – `draw()`, `move(x,y)`, and `rotate(angle)` – by which the library can draw a shape, rotate or move it around. If we are blunt, we could now create a simply derived class – say a circle. A `Circle` class might have been implemented already before, with the following interface.

```cpp
class Circle {
public:
    Circle(double midX, double midY, double radius) :
        Mx_{midX}, My_{midY}, R_{radius} {} 
private:
    double Mx_,My_; // center
    double R_; // radius
};
```

This `Circle` is but completely independent of `Shape`. So, let’s do inheritance.

![Shape hierarchy diagram](image)

```cpp
class Circle : public Shape {
    // <-- here it happens 
public:
    Circle(double midX, double midY, double radius) :
        Mx_{midX}, My_{midY}, R_{radius} {} 
private:
    double Mx_,My_; // center
    double R_; // radius
};
```

Colon, public, and then the base class name – that’s all you need for inheritance. Now, the compiler knows that `Circle` is a `Shape`, and you can use a circle as a shape.

```cpp
void doSomething(Shape & s) {
    s.move(10,10);
    s.rotate(3.14);
    s.draw();
}
```

`doSomething` actually expected a reference object of type `Shape`. But `Circle` was okay, too. If `Circle` was not derived from `Shape`, the compiler would have complaint. And right he is!

**Exercise:** Add a `Square` class! Derive it from `Shape`, too. Create an instance of `Square`, and hand it over to `doSomething`! Try to compile with and without: public `Shape`. 
20.2.1 Overriding and Virtual Function Call Mechanism

But we are not finished now. Square is still drawn as Shape; and so is Circle. So, let us override the functions of Circle.

```cpp
class Circle : public Shape {
public:
    Circle(double midX, double midY, double radius)
        : Mx_(midX), My_(midY), R_(radius) {} 
    void draw() const {std::cout << "draw circle\n";};
    void move(double x, double y) {std::cout << "move circle\n";};
    void rotate(double angle) {std::cout << "rotate circle\n";};
private:
    double Mx_,My_;    // center
    double R_;         // radius
};
```

**Exercise:** Do this accordingly also for Square, compile and run!

You should have noticed that still nothing has changed! A circle is still drawn as a shape! So, what are we doing wrong?! Answer: We have to tell the compiler that he should implement the dynamic dispatching via the virtual keyword.

```cpp
class Shape {
public:
    virtual void draw() const {std::cout << "draw shape\n";};
    virtual void move(double x, double y) {std::cout << "move shape\n";};
    virtual void rotate(double angle) {std::cout << "rotate shape\n";};
};
```

```cpp
class Circle : public Shape {
public:
    Circle(double midX, double midY, double radius)
        : Mx_(midX), My_(midY), R_(radius) {} 
    void draw() const override {std::cout << "draw circle\n";};
    void move(double x, double y) override {std::cout << "move circle\n";};
    void rotate(double angle) override {std::cout << "rotate circle\n";};
private:
    double Mx_,My_;    // center
    double R_;         // radius
};
```

Now it works! The circle is draw, moved and rotated as a circle. Every function in a base class must be marked by virtual, if it is supposed to become overridden! In the derived class, only override suffices.

**Exercise:** Test this also for Square!

We tacitly introduced here the keyword override – which is actually not strictly required. But it is a good practice to show other programmers and the compiler that these member functions are overriding virtual functions of the base class. This will prove really valuable once you made a mistake like writing carelessly e.g. `void rotate(float angle) override {...}` (note the different argument type!). The compiler will notice that you actually did not override a (the intended) base class member function. instead you create a new function. And the real evil thing is that when you omit the override, the code will compile and execute in spite of this. But the rotate (in this example) will be done for the Shape, not for a Circle. The reason is that – as before – `void rotate(double angle)` from Shape is still visible in the Circle (public inheritance!). And when called with argument 3.14 (a double, by default), the compiler chooses the rotate member function from Shape, because the overload resolution deems it a better match. Even more pernicious is that `rotate(float)` is hidden! The only way to get the float function would be to cast Shape& s back to type Circle& explicitly (casting will be explained a bit later in this part).
So, you see that there are some intricacies one can easily stumble over. That's why it is a good habit to write clean and explicit code directly from the outset. virtual to the base class member functions (those that shall be overridable), and override to the overridden functions in the derived class.

Of course, the overridden functions in the derived class can be also be made virtual, if you intend that someone can also derive from it. This would be multi-level inheritance.

### 20.2.2 Slicing and Private/Virtual Base Class Destructors

The issue with the missing virtual keyword in Shape shows up also at another place – the destructor of Shape. This is usually only a problem if you want to delete (destroy) an object of a derived class by a pointer of its base class to it. What happens is that only the base class object is deleted – the rest not. This ill behavior is called **slicing** and should be avoided, as it is a resource leak. To obviate this situation you should accustom yourself to always make base class destructors virtual!

```cpp
class Shape {
public:
  virtual ~Shape() {} // <-- !!!!
  [...]
};
```

This is also a good habit insofar you can strictly indicate that from this class can be inherited – even if no other virtual functions are declared.

**Nota Bene:** Never ever derive from STL classes like `std::string`, `std::vector`, etc. (with only rare exceptions)!! Their destructors are usually not virtual – so, they are not meant for inheritance!

You can also make the base class destructor private, with all its consequences. An explicit call to this destructor is not possible anymore then. For base classes that shall be instantiable, this is of course no solution!

### 20.2.3 Pure Virtual Functions – Abstract Base Classes

As we have it written so far, some programmer could have the idea to instantiate Shape.

```cpp
[...]
int main() {
  Shape s;
  doSomething(s);
}
```

But does it really make sense to instantiate an abstraction? You as the designer of the render library, and creator of Shape probably would rather not like that users can instantiate it. Nonetheless, they shall derive from it.

So, how could we prevent that users instantiate Shape? There is, of course, the possibility to delete the default and copy constructors, or to make them protected/private. The first idea is probably bad, because the constructor of a base class is needed (even if it is just the default constructor). To make them private has the same effect. Making them protected but would solve the problem. A class which publicly derives from the base class is allowed to execute protected member functions.\(^5\) So also is a protected constructor.

```cpp
class Shape {
public:
  virtual "Shape() {};
  virtual void draw() const {std::cout << "draw shape\n";};
  virtual void move(double x, double y) {std::cout << "move shape\n";};
  virtual void rotate(double angle) {std::cout << "rotate shape\n";};

\(^5\)Now you know, why protected is there!
```
protected:
  Shape() {}; // default c'tor
  Shape(const Shape&) {}; // copy c'tor

The copy constructor also must be handled. Otherwise, Shape s {c}; would work – and you had again a Shape instantiated.

But that is all quite complicated, and actually not necessary. Because the C++ language features allow a more direct way, which furthermore gives you control over functions that MUST be overridden by users. Consider the following.

```cpp
class Shape {
  public:
    virtual ~Shape() {};
    virtual void draw() const = 0; // <-- !!!
    virtual void move(double x, double y) {std::cout << "move shape\n";};
    virtual void rotate(double angle) {std::cout << "rotate shape\n";};

  private:
    double Mx_,My_; // center
};
```

The =0 at the draw() function signals the compiler that this function has no implementation (no function body) inside Shape. Together with the fact that it is virtual, this is a pure virtual member function and the class Shape thus a abstract base class. Just a single pure virtual function suffices to make a class non-instantiable – and so also non-copy-able.

At the same time, you as class designer signal users of your class that they MUST override this pure virtual functions. If they don’t do so, the compiler refuses to compile. This is a very fine control over access and instantiation permissions you have as class designer!

### 20.2.4 Multi-Level Inheritance and Final

Assume, we would like to let someone also derive from Circle (it does not make much sense – but just for illustration!). We would make all our member functions virtual; also the destructor. And then we inherit – call it MyCircle (in lack of creativity).

![Inheritance diagram](image)

**Figure 20.3: Shape ← Circle ← MyCircle inheritance hierarchy.**

```cpp
class Circle : public Shape {
  public:
    Circle(double midX, double midY, double radius)
      : Mx_(midX), My_(midY), R_(radius) {};
    virtual ~Circle() {};
    void draw() const override {
      std::cout << "draw circle\n";
    }
    void move(double x, double y) override {
      std::cout << "move circle\n";
    }
    void rotate(double angle) override {
      std::cout << "rotate circle\n";
    }
  private:
    double Mx_,My_; // center
};
```
Here is actually only one thing new. Because `Circle` has no default constructor (we removed it when specifying the non-default constructor), we have to explicitly call the constructor of the base class `Circle`. It is also important to know that the base class is always constructed first — before any initialization of the derived class itself. The base class part is also deleted last when the derived object is destroyed.

Creating a reference object of type `MyCircle`, and handing it to `doSomething()` would result in that it is drawn, moved and rotated as `MyCircle` — as we wanted.

This small multi-level inheritance tree `Shape <- Circle <- MyCircle` is the almost smallest representation of an inheritance hierarchy. Usually, this is already too deep, and you should take fright to go deeper unless there are really good reasons!

Last but not least, imagine that you as designer of `Circle` would not wish that a descendant of your class (here, `MyCircle`) overrides, for instance, `draw()`. Say, it might not make sense, and the implementation of `draw()` in `Circle` would suffice for any derived class. In this case, you can use the keyword `final`.

```cpp
class Circle : public Shape {
    public:
        Circle(double mx, double my, double radius)
            : Mx_{mx}, My_{my}, R_{radius} {}
        virtual ~Circle() {} // <-- compile error
        void draw() const final {
            std::cout << "draw circle\n";
        }
        void move(double x, double y) override {
            std::cout << "move circle\n";
        }
        void rotate(double angle) override {
            std::cout << "rotate circle\n";
        }
    private:
        double Mx_, My_; // center
        double R_; // radius
    };
    class MyCircle : public Circle {
        public:
            MyCircle(double mx, double my, double radius)
                : Circle(mx,my,radius) {} // <-- compile error
            void draw() const override {
                std::cout << "draw mycircle\n";
            }
            void move(double x, double y) override {
                std::cout << "move mycircle\n";
            }
            void rotate(double angle) override {
                std::cout << "rotate mycircle\n";
            }
        };
```
20.2.5 Data Members in Base Classes

As you have seen before, base classes can also contain data members (here, Circle still has the center point coordinates and the radius). As those class members are private, you cannot access them from MyCircle. This is the result of consequent data encapsulation (which is really a good thing!!). If you need to access those data members, there are generally two ways: 1) declare them as protected, 2) make them accessible through protected or public member functions of the base class (getters, setters). From a certain point of view, the latter solution is preferred.

20.2.6 Multiple Public Inheritance

Formally and practically is it possible that a class derives also from several base classes, i.e. from more than one. Obviously, certain objects are related to more than one abstract notion – and so can behave also like the two or more abstract notions. In real life, you certainly noticed already something like this. For instance, you can use a hammer as a tool, but also as a weapon. "Tool" and "weapon" are rather abstract notions, like "fruits" or "furniture". And for each of these we expect that an object as representative of the one or the other notion shows a certain behavior – guide-lined and important for that notion.

It is usually not so easy to find examples of multiple inheritance that are simple and reasonable at the same time. More often than not, you can and should avoid multiple inheritance. But there are patterns, which are more complicated and abstract – such as e.g. policy-based design (see Modern C++ Design), which cannot dispense with multiple inheritance.

Disregarding the mental complications when thinking about such constructs, the implementation is rather straightforward. One essentially needs a comma separated list of access specifier–base class name pairs after the colon (behind the class’s name). The sequence of these base classes determines the order in which the base class parts are created (and destroyed).

```cpp
class A : public B, public C {
    [...] 
};
```

Some complications appear when two base classes have member functions with the same name (and possibly same parameter list). Or, you want to derive from two classes which are themselves derived from one common base class. Once you face such a problem, you are not a novice in C++ anymore, and should consult the internet and literature on that topic (see e.g. [2], and look for virtual inheritance).

The internet offers some rules for class inheritance hierarchy designs (see here, for instance). And the design of such hierarchies is a science of its own. For the beginning, keep it small, simple and modest.

20.3 Private Inheritance vs. Class Data Members – HAS-A Relations

Up to now, we considered relations of the type "IS-A", meaning that public inheritance should be employed only if the derived type IS A base type. So, Circle IS A Shape. The vast majority of cases, this is what you will use – if inheritance at all.

But C++ would not be C++, if it would prohibit other sorts of inheritance, such as protected and private inheritance. The effect of these access specifications is with respect to the accessibility of the base class members (data and functions) from outside through the derived type classes. In public inheritance, the public base class members are accessible also from outside. From within the derived type class, public and protected class members can be accessed.

From the table 20.1 it becomes obvious that protected inheritance is not different from private inheritance. And there is indeed some rule of thumb that you should avoid protected inheritance.

There are cases, where private inheritance can be useful. As an example (from the internet).
20.3. PRIVATE INHERITANCE VS. CLASS DATA MEMBERS – HAS-A RELATIONS

<table>
<thead>
<tr>
<th>permission in base class</th>
<th>access from outside to base class</th>
<th>derived</th>
<th>access in derived class</th>
<th>access from outside through derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✔</td>
<td>public</td>
<td>✔ (public)</td>
<td>✔</td>
</tr>
<tr>
<td>protected</td>
<td>❌</td>
<td>public</td>
<td>✔ (protected)</td>
<td>❌</td>
</tr>
<tr>
<td>private</td>
<td>❌</td>
<td>public</td>
<td>❌ (private)</td>
<td>❌</td>
</tr>
<tr>
<td></td>
<td></td>
<td>protected</td>
<td>✔ (protected)</td>
<td>❌</td>
</tr>
<tr>
<td></td>
<td></td>
<td>private</td>
<td>❌ (private)</td>
<td>❌</td>
</tr>
<tr>
<td></td>
<td></td>
<td>private</td>
<td>❌ (private)</td>
<td>❌</td>
</tr>
<tr>
<td></td>
<td></td>
<td>private</td>
<td>❌ (private)</td>
<td>❌</td>
</tr>
</tbody>
</table>

Table 20.1: Access permission matrix: E.g. if a base class member is declared public, and the derived class inherits private, this base class member is private in the derived class, and can be accessed from within the derived class, but not from outside through the derived class.

```cpp
#include <iostream>
class Engine {
public:
  Engine(int nc) { cylinder = nc; }
  void start() {
    std::cout << getCylinder() << " cylinder engine started\n";
  }
  int getCylinder() { return cylinder; }
private:
  int cylinder;
};
class Car : private Engine { // Car has-a(n) Engine
public:
  Car(int nc = 4) : Engine(nc) { }
  void start() {
    std::cout << "car with " << Engine::getCylinder() 
              << " cylinder engine started\n";
  }
  Engine::start();
};
int main() {
  Car c{8};
c.start();
}
```

But to write it as follows without inheritance would do the same, and is more comprehensible.

```cpp
#include <iostream>
class Engine {
public:
  Engine(int nc) { cylinder = nc; }
  void start() {
    std::cout << getCylinder() << " cylinder engine started\n";
  }
  int getCylinder() { return cylinder; }
private:
  int cylinder;
};
class Car {
public:
  Car(int nc = 4) : engine_{nc} { }
  void start() {
    std::cout << "car with " << engine_.getCylinder() 
              << " cylinder engine started\n";
  }
  engine_.start();
};
Engine engine_; // Car has-a(n) Engine
```
Private inheritance implements a kind of HAS-A relation. This means, that the derived class somehow HAS the base class. One should therefore carefully check, whether to make the base class just a derived class' data member is possibly the better and simpler solution. Only if you need the interface of the base class, and the virtual mechanism, this mode of inheritance makes some sense. Otherwise, we recommend to use the class member variant (no inheritance).
Chapter 21

Exceptions

21.1 Motivation

We talked already about errors in the previous part, and noticed that there might be recoverable and non-recoverable errors. What is meant by that? And in this respect: What is meant by that an error can be anticipated? Let’s look at some examples. If for some reason a segmentation fault (memory access violation) is generated, making the program crash, then this is certainly not intended. And as you do not deliberately write such programs (usually), you, as the programmer, can also not anticipate that this error occurs. The same is true for cases where some eminent part of your hardware dies (CPU, cache memories, ...). In this category, I would also fit any kind of errors in the program logic – all kinds of errors you actually do not intent.

But there are also many cases, where you, as the programmer, can anticipate things that can go wrong. For instance, there might be input required from the user of your program, where you know that the user can enter wrong information – e.g. a number is required, but the user enters characters; or, more subtle, an integer is required, but the user supplies a floating point number. Or, there are network connection or memory allocation (RAM, file system) requirements in your program, where each of the required services might not be available (temporarily, or not at all). This needs to be considered for your program, unless you want to let your program simply crash in such cases. At least, users of your code usually do not like when the code simply fails ... possibly even without any message at all.

So, in these cases, you can do error handling. And you probably should! Initially, you would not worry about programs that simply terminate if something goes wrong. But with the time, your program become so elaborate that it will be used in production – later even by other users. And a crash then might be very expensive. You may try to capture some of the errors that occur during run-time, and next recover the program to a state that it can continue to work (or at least rescue some of the available data after month of simulation work). So, you see that error handling is also a question of the costs.

21.2 Implementation

21.2.1 Classical Error Handling

Error handling can be done in several ways (see this C++ Con Contribution: What could possibly go wrong for a general introduction and discussion). The easiest for beginners, is usually return values – as this is the way C++ inherited from native C. This is accomplished by setting certain return values. For instance,

```cpp
#include <cmath>
#include <iostream>

double Square_Root(double x) {
    if (x < 0)
```

1OS programmers may be exceptions here.
The anticipated error – here that the user enters a negative number – is handled, and a corresponding error message is printed.

Not always appears it reasonable to feed the error into the same format as the return value. Or, in other words, a misuse of return values, is anyway a bad idea. So, we could do a bit better by introducing a function parameter that takes an error flag.

```
#include <cmath>
#include <iostream>

double Square_Root(double x, int & errflag) {
    errflag = 0;
    if (x < 0) {
        errflag = -1;
        return 0.;
    }
    return std::sqrt(x);
}

int main() {
    int errflag;
    double x;
    std::cout << "Please enter a number: ";
    std::cin >> x;
    double val = Square_Root(x, errflag);
    if (errflag == 0) {
        // happy path
        std::cout << " \n\n";
    } else {
        // error path
        std::cout << " Error: sqrt(<0) not possible!\n";
    }
}
```

But I guess, you also think that is not much better – although in this way, you could differentiate different error conditions. You could even use enums to make the code more readable.

```
#include <cmath>
#include <iostream>

enum ErrVal { ok, error };

double Square_Root(double x, ErrVal & errflag) {
    errflag = ok;
    if (x < 0) {
        errflag = error;
        return 0.;
    }
    return std::sqrt(x);
}

int main() {
    ErrVal errflag;
    double x;
    std::cout << "Please enter a number: ";
    std::cin >> x;
    double val = Square_Root(x, errflag);
    if (errflag == ok) {
        // happy path
        std::cout << " \n\n";
    } else {
        // error path
        std::cout << " Error: sqrt(<0) not possible!\n";
    }
}
```
21.2. Exceptions

In any case, there is much potential for improvements, as there is now still much clutter in the code for just error handling. The actual useful code could be much shorter and comprehensible. Return value misuse, or extra function parameters are not really desirable in terms of code maintenance. C++ thus went a different way – Exceptions (see this nice survey C++ Con Contribution: The Unexceptional Exceptions). This comprises the introduction of three essential key words: throw, try and catch.

In its simplest, the above code could be rewritten as follows.

```cpp
#include <cmath>
#include <iostream>

double Square_Root(double x) {
    if (x < 0)
        throw -1;
    return std::sqrt(x);
}

int main() {
    double x;
    std::cout << "Please enter a number: ";
    std::cin >> x;
    try {
        // happy path
        double val = Square_Root(x);
        std::cout << " sqrt(" << x << ") = " << val << "\n";
    } catch(...) {
        // error path
        std::cout << " Error: sqrt(<0) not possible!\n";
    }
}
```

The idea is simple. At the place where the problem occurs, an exception is thrown. That is inside the function Square_Root. Inside `main()`, there is a try block, which represents the normal execution path (“happy path”). And then there is the catch block, where the exception is caught, and handled. The catch also has a parameter list, which contains three dots (...). This is a so-called variadic argument, which is used to indicate that this function can take any type, and also any number of parameters (of different types). Usually, we won’t use it very often. And here, we only want to catch just any thrown exception (kind of default).

The big advantage of exceptions is that their execution is completely disentangled from the happy path. So, we can also throw whatever we like to through.

```cpp
#include <cmath>
#include <iostream>
#include <string>

double Square_Root(double x) {
    if (x < 0)
        throw std::string("Error: sqrt(<0) not possible!\n");
    return std::sqrt(x);
}

int main() {
    double x;
    std::cout << "Please enter a number: ";
    std::cin >> x;
    try {
        // happy path
        double val = Square_Root(x);
        std::cout << " sqrt(" << x << ") = " << val << "\n";
    } catch(std::string& s) {
        // error path
        std::cout << s;
    } catch(...) {
        // error path
        std::cout << " Error: Something else happened -- don’t know what!\n";
    }
}
```
Here, we've thrown a string, and caught it by type. Furthermore, we have introduced a catch hierarchy. If the exception thrown is a string, it will be caught by the first catch block. If it is something else, it will be caught by the second catch block.

What might look quite complicated on a first gaze, is actually rather simple, and will reveal its full power in situations, where there are many different errors possible (more or less at the same time), and can/must be handled differently. In order to understand this, we just need to introduce a last small piece of the puzzle. Because we can obviously throw everything we like, we can also throw class objects. And even better, we can create exception class hierarchies, in order to logically sort them (as the STL does, [cppreference: std::exception]). Thereby, one can capture whole groups of exceptions, on the one hand, but one can also handle selectively each single type of exception, if needed.

Consider the following code.

```cpp
#include <bitset>
#include <exception>
#include <iostream>
#include <string>
#include <vector>

int main() {
  try {
    std::vector<double> v {1,2,3,4};
    double d = v.at(5);              // throws out_of_range
    d += 1.;
    std::bitset<5> mybitset(std::string("01234")); // throws invalid_argument
    throw 5;                        // throws int
  } catch (const std::out_of_range& e) {
    std::cerr << "OOR error: " << e.what() << "\n";
  } catch (const std::logic_error& e) {
    std::cerr << "LOG error: " << e.what() << "\n";
  } catch (const std::exception& e) {
    std::cerr << "STD error: " << e.what() << "\n";
  } catch (...) {
    std::cerr << "Error: Something else happened -- don't know what!\n";
  }
}
```

Line no. 9 will throw an std::out_of_range error, which is a std::logic_error (meaning, it is publicly derived from). But as the first catch handler in line 13 can handle std::out_of_range, this handler will be used. Line 11 throws std::invalid_argument, which is also a std::logic_error. But as there is no catch for std::invalid_argument, the handler of std::logic_error is used (line 15). Finally, line 12 throws an int, which is only caught by line 19 – because there is no catch handler for int, and ... catches everything.

Although a std::logic_error is also a (derived from) std::exception, the handler for std::exception is never called. It would be called, if there would be thrown some other error derived from std::exception except from std::logic_error (there are still some when you look at [cppreference: std::exception]).

We used here the STL exception hierarchy. You can (as exception) either inherit from it.
Implementation

By now, you should understand inheritance enough to anticipate, which of the catch handlers will be called. If not, try it out ... and then explain the outcome!

Of course, you don’t need to derive from std::exception or one of its descendants. You could create a class completely of its own (and so also your own hierarchy if needed).

```cpp
#include <iostream>
#include <string>

class MyException {
public:
    explicit MyException(const char* message): msg_(message) {}  
    explicit MyException(const std::string& message): msg_(message) {}  
    ~MyException() throw () {}  
    const char* what() const throw () { 
        return msg_.c_str(); 
    }  
protected:
    std::string msg_;  
};

int main() { 
    try {  
        throw MyException("Yelp! An error!");  
    } catch (const MyException& e) {  
        std::cerr << " My error: " << e.what() << "\n";  
    } catch(...) {  
        std::cerr << " Error: Something else happened -- don’t know what!\n";  
    }  
}
```

But if you want to catch it, you must implement a corresponding catch handler (i.e. a catch statement with the correct type).2

It has prevailed to write exceptions in such a way that they carry a message, which can be retrieved via what(). But that’s rather more of a convention than of a necessity. You are free to write any kind of class to throw as an exception.

### 21.2.3 Some Final Remarks

If none of the catch handlers captures a thrown exception, then the program terminates without any other message. In this way, exceptions cannot be ignored (as is the case for return values and error flags).

Another advantage of exceptions is that they can be thrown from everywhere – even from constructors and destructors. As these both have no return value to submit, or – in the case of destructors – can have no parameters, exceptions would be the only way to leave some error message to the outside world. However, before you start making all your constructors and destructors throw, think twice! (That’s always a good advice!!) A destructor should possibly never throw. There should never go something wrong when you release resources (in oppose to acquiring them). And also not every constructor should throw. And with hindsight of the next section, in most cases it is better if it does not throw. If you can give this so-called no-throw guaranty, you should mark this by noexcept (cpp-preference: noexcept spec). So, please use exceptions carefully, and only when they really can help. Senseless or very expensive recovery is probably worse than letting the program crash, and subsequently restart.

---

2If you want to derive from MyException, make at least the destructor virtual (as explained before)!
Btw. a caught exception can be re-throw-n. Inside the catch handler just write `throw`. The currently handled exception will then be re-thrown. This can be useful if the problem cannot be fully handled inside the block where the exception was caught, or when another problem also occurred at the same time, which must be handled elsewhere. Of course, this would mean that the whole `try-catch` construct is wrapped in another, an outer `try` block. This nesting of `try-catch` blocks is feasible.

### 21.3 Exception Safety

Of course, exceptions would be a rather useless feature if you could only document the occurrence of an error by writing a message. The catch blocks should be used instead to recover the program to a reasonable and definite state. For instance, although that's not a good programming style, you might have allocated memory and assigned a pointer to it. Afterwards, some exception occurs, which makes this allocation obsolete. The catch block, which handles this exception, should now try to release this memory again (unless it is still in use). In that way, the catch block is kind of a clean-up part. Recovery is but really a tough business. One really needs to think hard, where it makes sense to recover ... and what.

There is another business, which is related to resource management, but also becomes important when you use exceptions. Imagine, you have a function (could be also a class member function like constructors), which already allocates memory, open files, locks some resources (multi-threading), etc. And now happens an exception, and the function is left. What happens to the resources which were already reserved? Who takes care for cleaning or freeing these resources?

In part, we answered this question fortunately already for you. **Value Semantics!** When the scope of the throwing function is left, the destructors of all reference objects (not being created via `new` or `new[]`) are called. When the destructors are written carefully, all resources are released automatically. When you can at least accomplish this guaranty that no resources are blocked or leaked, you can give the so-called **basic exception safety guaranty**. (See Wiki: Exception Safety.)

This is sometimes difficult enough. Even harder to accomplish is the so-called **strong exception safety guaranty**. This means that your throwing function even does not leave the program in a changed (possibly undefined) state once an exception occurred. Many of the STL functions provide such a strong guaranty (e.g. `push_back()` of the containers).
Chapter 22
Type Casting

Already in the good old time, programmers sometimes wanted to cast the value of a variable of one type into one of another type. For instance, a double into an int, or vice versa. In part, the compiler does this already implicitly, if it is not dangerous. For instance, 1 (int) to 1.0 (double) is no problem. But from 1.5 (double) to ?? (int) is a bit tricky. Currently, the procedure is to cut off the post-decimal point part. But a compiler may complain and refuse to do this narrowing implicitly. The programmer has to do it explicitly.

As a legacy of its C-origin, C++ allows the old C-way to do the cast.

```cpp
double d = 1.5;
int i = (int)d;
```

Trying to compile this with GCC, and the options -Wall -pedantic-errors will most probably work. And even worse, there will not even be any warning or complaint at all. That is the reason why you should avoid this.

One correct C++ way is the conversion constructor (see “operator overloading” of the previous part).

```cpp
double d = 1.5;
int i = int(d);
```

This will result in a warning (or an error with -pedantic-errors).

With the advent of classes and user-defined types and inheritance, the need for conversion became even larger. And so, C++ invented the explicit type conversion functions (see cppreference: Explicit type conversion). They are template functions. Their usage is so important that we decided to include it.

Before going on here, a side-note: When you have to do explicit type conversion in that way, it hints maybe on that something is not well designed. Before circumventing the strong type system of C++, please, think carefully whether there is another solution! Finally, programmers can build conversion constructors and operators to accomplish conversion between types. If they did not do so, there is maybe a reason!

There are four such functions, and we list them in the order of safety. Not all conversions are reasonable.

- `const_cast<new type>(expression)` adds or removes const-ness (cppreference: const_cast conversion)
- `static_cast<new type>(expression)` tries to change the static type (cppreference: static_cast conversion)
- `dynamic_cast<new type>(expression)` tries to change the dynamic type (cppreference: dynamic_cast conversion)
- `reinterpret_cast<new type>(expression)` tries to interpret one type as another (cppreference: reinterpret_cast conversion)
The last one, reinterpret_cast<new type>(expression), should be used only as a very last resort.
const-casting is also not treated here, as you should use it only rarely.
static_cast and dynamic_cast are remaining. Before we can start to explain how they work,
we first have to say what they do. And in order to do this, we have first to explain what is a static
type, and what is a dynamic type.
Static type is actually quite straightforward. According to cppreference: Type, a static type is one "that results from a compile-time analysis [...] The static type does not change while the program is running." That is essentially almost all we had before.
So, static_cast-ing is rather straightforward in the previous example. Because i and d have well
defined static types (int and double, respectively), we can simply convert.
```cpp
double d = 1.5;
int i = static_cast<int>(d);
```
The compiler will not complain, and the program will work. By static_cast, we told him that we
know what we are doing, and accept any consequences of our doing.
Of course, you can convert also user-defined types if necessary. static_cast does not perform
any run-time checks. So, it usually also does not throw an exception (except your own conversion
operators and constructors throw).
A dynamic type is accordingly one that can change during run-time. cppreference: Type gives the
following example.
```cpp
struct B {
    virtual ~B() {}  // polymorphic type
};
struct D : B {} ;  // polymorphic type
D d ;  // most-derived object
B* ptr = &d;
```
The static type of (*ptr) is B, and its dynamic type D. cppreference: dynamic_cast conversion gives
the following example for casting.
```cpp
struct V {
    virtual void f() {};  // must be polymorphic to use
    // run-time-checked dynamic_cast
};
struct A : virtual V {};  
struct B : virtual V {
    B(V* v, A* a) {
        // casts during construction
        // (see the call in the constructor of D below)
        dynamic_cast<B*>(v);  // well-defined: v of type V*,
        // V base of B, results in B*
        dynamic_cast<B*>(a);  // undefined behavior: a has type A*,
        // A not a base of B
    }
};
struct D : A, B {
    D() : B(static_cast<A*>(this), this) {}
};
```
The example also explains the terms upcast, downcast, and sidecast. (For the complete example, see
cppreference: dynamic_cast conversion.)
Such complexity should possibly be avoided at the beginning when learning C++. The essential
point to make here is that if you must cast user-defined types in class hierarchies, it is usually a very
good idea (if not even necessary – compiler will tell you) to use dynamic_cast. This has a small
run-time overhead as it determines the dynamic type during run-time. But it offers some safety in that
respect that is throws a std::bad_cast exception (and returns a null pointer) if the cast failed. So,
you are – when using this correctly – rather protected from doing nonsense (unless you find something
else to crash the system).
Chapter 23
Optimization

When people talk about optimization, in most cases, they talk about accelerating the program execution (speed). But there are communities like those for embedded systems, where optimization for size in memory is in part more important. Another optimization goal can be speed-up of compilation, which is surely relevant for large programs and libraries during development. Accordingly, you should choose your methods for optimization. We will focus here on optimization/speeding up the program execution, and give more general guidelines.

The most important points for optimization are:

- **Correctness first!** If your program produces incorrect results, no optimization will cure this! And if optimization spoils correctness, the optimization has no value!

- **Premature optimization is often bad!** Of course, you should design your software with optimization in mind whenever you can. But first, the program needs to run correctly. And early optimization during development might lead to a result that is not optimal at all. And even worse, it might be so complicated by the “optimizations”, such that later optimization efforts might be impossibly difficult.

- **Complexity is an enemy of Performance!** Complexity may cause possibly many problematic performance inhibitors, which all to handle might be impossible. Optimization on a single dominant bottleneck is usually easier, and more profitable!

- **Abstraction is a friend of Performance!** STL shows that this is the case. Abstraction also helps to handle and mitigate complexity.

- **What is done beforehand, does not need to be done at run-time!** In its widest scope, this means that you should do e.g. analytic calculations if and as far as possible, when dealing with numeric, for instance. Something, where our grandfathers were proud on. But already C++ admits several possibilities for compile-time code/program execution (template meta programming, const and constexpr).

23.1 Levels of Optimization

There are several possibilities for a programmer to optimize/accelerate the program’s execution. A surely not comprehensive list could be the following.

- **Algorithmic:** It is always a good idea to think about what you want to achieve by your program, and with which means to do it. This includes also to look for algorithms, which are expected to perform better (see e.g. for complexity classes) than others. Accordingly, all your data structures might have to be arranged accordingly to meet the needs for the chosen algorithm. In our eyes, that is the only early optimization (during the planning) that is reasonable, unless you find a way
to flexibly change this algorithm and the corresponding data structures later during the project advancement or even run-time. But this might incur an increase of complexity. And complexity is an enemy of ... (you remember?)

On modern hardware, most of program acceleration is accomplished mostly by parallelization – on different levels (instruction execution parallelism, vector register parallelism, multi- and many-core (threaded) share memory communication parallelism (TBB, OpenMP, ...), multi-processor network communication parallelism (MPI)). It should immediately be clear that the mathematical structure of algorithms might resist the one or the other parallelization efforts. But that’s far beyond the scope of this basic course.

- **Programmatic**: To some extend, the choice of a programming language, or features of a chosen programming language, fits here as well. But years ago it was paramount to write optimal code yourself. Nowadays, it usually suffices not to stop the compiler from doing a good optimization job. But on modern vector hardware, this can become sort of a challenge again (and also deeply affects the algorithmic level). A very profound survey of C++ performance optimization can be found in [7]. We outline here some basic optimization principles, and introduce also some tools and their basic usage for simple profiling.

A compiler can perform optimization on the different stages of the program building process. C++ unfortunately cannot perform run-time optimization (as, for instance, Python or Java). But it also does not incur much run-time environment overhead (some may come into the game when using RTTI, the virtual dispatching mechanism, and such things ... which but can be avoided). But during compilation, and linking, and even afterwards, compilers can perform large amounts of code restructuring according to certain heuristics in order to optimize and accelerate the code. How the compiler does this is often a kind of black magic. We will not deal much with that. The compiler shall do this!

### 23.2 Optimization – Basic Procedures

#### 23.2.1 Performance Measurement

Let us assume, your program is so far ready – you decided on the programming language and features, the algorithms (even parallelization). What to do next?

The starting point is to fix a performance metric, some number such as e.g. the execution time of the program. This can be done by the OS’s timer capabilities such as that of `time` (either `bash`'s `time`\(^1\), or `/usr/bin/time`\(^2\)).

```bash
$ time ./main
0.00user 0.01system 0:00.02elapsed 95%CPU (0avgtext+0avgdata 3492maxresident)k
0inputs+8outputs (0major+141minor)pagefaults 0swaps
```

Or, by using `std::chrono`.

```cpp
#include <iostream>
#include <chrono>

int main() {
    using namespace std::chrono;
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    for (int i=0; i<10000; ++i)
        std::cerr << *; 
    std::cerr << \n;
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double> time_span = duration_cast<duration<double>>(t2 - t1);
    std::cout << "It took me " << time_span.count() << " seconds.\n";
}
```

\(^1\)check `help time`
\(^2\)man 1 time or `/usr/bin/time --help`. Has more configuration options, and also gives more information. Please
23.2. **OPTIMIZATION – BASIC PROCEDURES**

`high_resolution_clock::now()` acquires the current (in the moment of the call) time stamp. Creating two such stamps – one before, one after the work – and forming the difference (duration), gives you the measured run-time.

`std::chrono` has certainly the advantage to time individual code regions, as needed.

**Exercise:** Exchange `std::cerr` by `std::cout` in the code above! Does something change?

If you need more detailed information about which function needed the most of the program’s total run-time, but are too lazy to instrument it by hand (i.e. using `std::chrono`), then you can use one of the many performance profiling or/and tracing tools. GNU (Linux/Posix), for instance, provides `gprof`. In order to use it, you have to do the following.

- Compile and link your program with the `-pg` flag under GCC. Other compilers possibly require different flags. You can also switch on `-g` in order to have readable names for your functions.

  ```
  $ g++ -std=c++11 -pg -g -Wall -pedantic-errors -o main *.cxx -I.
  ```

- Run the program once normally – for the measurement.

  ```
  $ ./main
  ```

  A binary data file `gmon.out` will be created.

- Analyze the results!

  ```
  $ gprof ./main > results.txt
  ```

  You can now investigate the `results.txt` file with a normal pager or text editor.

What you see is once a **flat profile**, i.e. the function name, the number of total calls, the total/cumulative time within this function, and the time per call. Self is the self-time, while the other is the total time. Self-time is really just the time within the function – not in a function called possibly from this function (see Interpreting gprof’s Output).

If you use the Intel Compiler Suite, VTune Amplifier is comparable to `gprof` – but has a much nicer GUI with graphical result presentations.

Now, you have a more or less finely grained overview of where does your program spend its time. Those parts or functions, where the most of the program’s time is spend, is the most interesting one for optimization, because already smaller improvements here can increase the total program’s performance significantly – much more than in functions, which are almost never executed.

**Remark:** A rather comprehensive list of performance tools can be found [here](#).

### 23.2.2 Compiler-Optimizations

Once you’ve identified the hot loops or functions, i.e. those dominant time-consuming parts, you need to investigate, whether and (if so,) what you can optimize. This is actually the hardest part. On the language level, we would like to refer to [7]. On the compiler level, you can introduce compiler flags. GCC knows quite a lot of optimization flags (see Options That Control Optimization), the most often used ones are `-01`, `-02`, and `-03`. `-03` represents the most aggressive generic optimization level, which still does not include hardware-specific optimization. But larger scale code- restructuring for optimization is possible, making it possibly harder to debug, while increasing the chance that the optimization might spoil the correctness. So, starting with `-02` is usually recommended.
For most purposes, this gives already a nice performance boost. Please remember the 20-80-Rule! Simple performance boosts might be accomplished in short time. But to squeeze out the last drops of speed-up will most probably take much more time and effort.

This short overview over and introduction to optimization as an aspect that programming is always affected with, shall suffice.
Chapter 24

Hands-On Exercises for Part 4

24.1 Cryptography – RSA Encryption

There are at least two shortcomings of the above encryption algorithm. First, there is character-wise
encryption. A block-wise encryption, meaning when several characters would be encrypted at the
same time, would enhance obscurity. There are modern symmetric algorithms like AES that do that.
Second, symmetric encryption (one key for both encryption/decryption) have the problem on how to
safely transfer a key (Do you trust the mail services?!)? A solution are asymmetric cryptographic
algorithms like Wiki: RSA\(^1\). One key is here for encryption, and the other one for decryption, where
both keys are but somehow related. The logic is so that you create this key pair. One key is the public
key you send to the world. When someone out there now wants to send you a secret message, he/she
uses your public key to encrypt it. The clue of the algorithm is that only you can decrypt it via the
other, the private key!

Let us outline (without derivation) the steps in that algorithm.

1. **Generation of Keys:** (these being later passwords; needs to be done ones, only)
   a) Select two large prime numbers, \( p, q \), and calculate their product, \( n = p \cdot q \).\(^2\) Also, calculate
      \( \phi(n) = (p - 1)(q - 1) \).
   b) Determine two numbers \( k, l \) such way that \( k \cdot l \equiv 1 \pmod{\phi(n)} \).\(^3\)
   c) **private key:** \( p, q, l \) (although \( n = p \cdot q \) and \( l \) are actually needed for decryption)
      **public key:** \( n, k \) (that’s the one you can send into the world!)

2. **Encryption:** (using \( n, k \)) Imagine, there is a message represented by a number \( m < n \). Calculate
   the cipher text
   \[
   c = m^k \pmod{n} .
   \]
   Send \( c \) to the recipient with the corresponding private key!

3. **Decryption:** (using \( n, l \)) Having obtained \( c \), calculate the original message by
   \[
   m = c^l \pmod{n} .
   \]

As essentially \( k \) and \( l \) are exchanged for encryption and decryption, similar to the XOR encoder, we
can use the same algorithm for encryption and decryption. Only the keys need to be exchanged.\(^4\)

---

\(^1\)Rivest, Shamir, Adleman – the creators.
\(^2\)If you don’t know what a prime number is, please look later in the Hands-On Exercises. But you also don’t need
them here, as we supply some for you for the sake of exercise.
\(^3\) \( \pmod{X} \) is *modulo* division, meaning division by \( X \), and keeping the rest. In C/C++, the operator \( \% \) realizes this
on integral data types.
\(^4\)
So, one rarely has to send pure numbers in messages. Text would be nice too! This can be done as follows. An short int is a four byte = 16 bit integer number. Two chars fit into it. We can thus exploit that the character ASCII bit sequences can be interpreted as numbers. Two chars are thus bundled, and cast into one short int.

**Task:** Realize such a RSA en-/de-coder! User $n = 176711573$, $k = 72097$, and $l = 45393253$;

**Hint:** For the modular exponentiation, which will overflow if not done carefully, you should write routine that does it more carefully (Wiki: Modular exponentiation).\(^5\)

```c
// for x^y mod n
long power(long x, unsigned long y, long n) {
    long long res = 1;
    x %= n;  // x = x%n
    while (y > 0) {
        if (y & 1) res = (res*x) % n;  // y bitwise AND 1 (y odd?)
        y >>= 1;  // div by 2
        x = (x*x) % n;
    }
    return static_cast<long>(res);
}
```

**Hint:** As it crystallizes, the largest difficulties of this exercise are to correctly read in and write out the short int binary bit patterns, and the type castings.

### 24.2 Visualization

#### 24.2.1 Visualization – Simple 2D Drawing (Tutorial)

In the previous part (3), we introduced a XYimage class with examples. And in the part 2, we introduced some geometric objects, to which we could executed some operations. In this hands-on session, we would like to combine them to one drawing program. The background is that we would like to create a geometric class (Shape), which extensible in a certain way (new geometric objects). This shall illustrate how a class hierarchy design process could work. We show how to accomplish this in a tutorial style. Keen programmers may try their own way.

So, we collect all the following files from the former hands-on sessions: Color.h, Point.h, Triangle.h, XYimage.h into include, and XYimage.cxx into src. We will first refactor a bit, and separate more consequently declaration and implementation.

```c
// Color.h
#ifndef COLOR_
#define COLOR_
#define COLOR_
#include <cassert>
class Color {
    using uchar = unsigned char;
public:
    Color(uchar red=0, uchar green=0, uchar blue=0);
    void setColor(uchar red, uchar green, uchar blue);
    uchar getRed() const;
    uchar getGreen() const;
    uchar getBlue() const;
private:
    uchar r,g,b;
};
#endif
```

\(^5\)The example keys were chosen in order that the cipher text should be maximum int (32-bit). This makes the cipher data twice as large as the plain text data. Larger keys (specifically larger $n$) might require to even use long as cipher type. In that case, the chance is but high to get an overflow in the modular exponentiation!
// Color.hxx
#include "Color.h"

using uchar = unsigned char;
Color::Color(uchar red, uchar green, uchar blue)
    : r(red), g(green), b(blue) {}

void Color::setColor(uchar red, uchar green, uchar blue) {
    r = red, g = green, b = blue;
}
uchar Color::getRed() const { return r; }
uchar Color::getGreen() const { return g; }
uchar Color::getBlue() const { return b; }

// XYimage.hxx
#ifndef XYIMAGE_
#define XYIMAGE_
#include <memory>
#include <string>
class Color;
class XYimage {
    using uint = unsigned int;
public:
    XYimage(uint DimensionX, uint DimensionY);
    void setPixelRGB(uint ix, uint iy, const Color& c);
    void writeToFile(std::string filename);
private:
    uint dimX, dimY;
    std::unique_ptr<unsigned char[]> color;
};
#endif

// XYimage.hxx
#include "XYimage.h"
#include "Color.h"
#include <iostream>
#include <fstream>
XYimage::XYimage(uint DimensionX, uint DimensionY)
    : dimX{DimensionX}, dimY{DimensionY},
        color{std::unique_ptr<unsigned char[^3*dimX*dimY]>(
            new unsigned char[^3*dimX*dimY])}
{}
void XYimage::setPixelRGB(uint ix, uint iy, const Color& c) {
    color[(ix+dimX*iy)*3] = c.getRed();
    color[(ix+dimX*iy)*3+1] = c.getGreen();
    color[(ix+dimX*iy)*3+2] = c.getBlue();
}
void XYimage::writeToFile(std::string filename) {
    std::ofstream file(filename);
    if (!file) {
        std::cerr << "Could not open file \"" << filename << "\"!\n";
    return;
    }
    file << "P6\n" << dimX << " " << dimY << " \n255\n";
    file.write(reinterpret_cast<const char*>(color.get()),3*dimX*dimX);
}

// Point.h
#ifndef POINT_
#define POINT_
#include <ostream>
class Point {
public:
    Point(double x, double y);
    Point(const Point& p);
    double getX() const ;
    double getY() const ;
    void setX(double x) ;
    void setY(double y) ;
    Point& operator=(Point const & p) ;
};
Point& operator=(Point const& p);
Point& operator=(double v);
Point& operator/=(double v);
void rotate_deg(double angle, Point const& RotCenter);
private:
  double x, y;
};
Point operator+(Point p1, Point const& p2);
Point operator-(Point p1, Point const& p2);
Point operator*(Point p1, double v);
Point operator/=(Point p1, double v);
double operator+(Point const& p1, Point const& p2);
std::ostream& operator<<(std::ostream& os, Point const& p);
double distance(Point const& p1, Point const& p2);
#endif
// Point.cxx
#include "Point.h"
#include <cmath>
Point::Point(double x, double y) : x{x}, y{y} {}
Point::Point(const Point& p) : x{p.x}, y{p.y} {}
double Point::getX() const { return this->x; }
double Point::getY() const { return this->y; }
void Point::setX(double x) { this->x=x; }
void Point::setY(double y) { this->y=y; }
Point& Point::operator+=(Point const& p) {
  this->x += p.x, this->y += p.y;
  return *this;
}
Point& Point::operator-=(Point const& p) {
  this->x -= p.x, this->y -= p.y;
  return *this;
}
Point& Point::operator*=(double v) {
  this->x *= v, this->y *= v;
  return *this;
}
Point& Point::operator/=(double v) {
  this->x /= v, this->y /= v;
  return *this;
}
void Point::rotate_deg(double angle, Point const& RotCenter) {
  const double pi = 3.14159265359;
  angle*=pi/180.;
  double ca = std::cos(angle), sa = std::sin(angle);
  Point pt {(*this) -= RotCenter};
  x = ca*pt.x-sa*pt.y;
  y = sa*pt.x+ca*pt.y;
  (*this) += RotCenter;
}
Point operator+(Point p1, Point const& p2) {
  return p1+=p2;
}
Point operator-(Point p1, Point const& p2) {
  return p1-=p2;
}
Point operator*(Point p1, double v) {
  return p1*=v;
}
Point operator/=(Point p1, double v) {
  return p1/=v;
}
double operator*(Point const& p1, Point const& p2) {
  return p1.getX()*p2.getX()+p1.getY()*p2.getY();
}
std::ostream& operator<<(std::ostream& os, Point const& p) {
  os << "(" << p.getX() << "," << p.getY() << ")";
  return os;
The Triangle class is already modified for our purposes. That is, it is already a derived class. In order to go along some systematics, we now introduce the base class – Shape.

As you can see, it is a pure abstract class. That is, it cannot be instantiated. This makes certainly sense for a Shape. Consider the fine-tuning we achieve for the derived class design. First of all, a derived class must overload all members ending with \_\_0. These are \_containsPoint\_, \_getCenterOfMass\_, \_move\_, \_rotate\_, because those need to be implemented differently for different shapes like a triangle, circle, rectangle, etc. We decided also that Shape has a Color data member, and also member functions to set and get it. This is also reasonable, because all shapes have a color – no matter which concrete form it has. So, we can reuse implementation here by inheritance.

\_containsPoint\_ will be needed later to determine, which point on the XYimage shall be draw with which color.

To make it complete, we also introduced a Shape constructor, which takes a Color. So, each shape must be initialized with a color. And because we deleted the default constructor, a shape cannot be created without a color. This enforces all classes derived from Shape to also get initialized with a color, as we will see now in the example of the Triangle class.
We prevent that it can be derived from Triangle by using the final keyword behind the class name specification. Otherwise, we overload all pure virtual functions from Shape as required. Of course, we can add more class members as needed – in this case, the array of three point.

Also look at the constructor, where we first of all initialize the base class Shape!

Now, we are ready to bundle all the ingredients in our drawing framework, which we call DrawFW.

---

6The problem – to determine whether a point is inside a triangle or any other shape – can be tricky! Don’t let you discourage by this! For the triangle, we determine the point’s barycentric coordinates by using Cramer’s rule for solving a linear equation. There are surely other ways.
24.2. VISUALIZATION

```cpp
Point ptl, pbr;
double res;
Color bgcol;
};
#endif

// DrawFW.cxx
#include "DrawFW.h"
#include "XYimage.h"
#include <iostream>

DrawFW::DrawFW(Point pTopLeft, Point pBottomRight,
                double resolution, Color background)
    : ptl{pTopLeft}, pbr{pBottomRight}, res{resolution}, bgcol{background} {}

DrawFW::~DrawFW() {
    for(auto& p: shapeStack) delete p;
    shapeStack.clear();
}

std::vector<Shape*>& DrawFW::getShapeStack() {
    return shapeStack;
}

void DrawFW::Add(Shape* shape) {
    shapeStack.push_back(shape);
}

void DrawFW::SafeAs(std::string filename) {
    unsigned int dimX=static_cast<unsigned int>((pbr.getX()-ptl.getX())/res);
    unsigned int dimY=static_cast<unsigned int>((ptl.getY()-pbr.getY())/res);
    XYimage XYi {dimX, dimY};
    for(unsigned int ix=0; ix<dimX; ++ix)
        for(unsigned int iy=0; iy<dimY; ++iy) {
            Color dc {bgcol};
            for(auto p: shapeStack)
                if(p->containsPoint(Point{ptl.getX()+res*ix,
                                          pbr.getY()+res*(dimY-iy)})) {
                    dc = p->getColor();
                    break;
                }
            XYi.setPixelRGB(ix, iy, dc);
        }
    XYi.writeToFile(filename);
}

As you may notice, we didn’t use Triangle at all – only Shape. That’s the clue to stay generic! The DrawFW has a constructor which creates the drawing plane. It contains a vector of pointers to Shape, to which we can add during run-time as many shapes (derived classes) as we wish. Finally, DrawFW contains a class member, SafeAs, where the image is really rendered into a file (PPM).

The usage of DrawFW is illustrated in the main.cxx.

```cpp
#include "Triangle.h"
#include "Circle.h"
#include "DrawFW.h"
#include <iostream>

int main() {
    DrawFW dwf {Point(0, 10), Point(15, 0), 0.01, Color{30, 255, 255}};
    dwf.Add(new Triangle{Point{0, 0}, Point{1, 0}, Point{0, 1}, Color{255, 0, 255}});
    dwf.Add(new Triangle{Point{1, 0}, Point{2, 0}, Point{1, 1}, Color{0, 0, 255}});
    dwf.Add(new Triangle{Point{2, 0}, Point{3, 0}, Point{2, 1}, Color{255, 255, 0}});
    dwf.Add(new Triangle{Point{2, 1}, Point{3, 1}, Point{2, 2}, Color{255, 255, 255}});
    dwf.Add(new Circle{Point{1, 1}, 1.0, Color{30, 200, 200}});
    dwf.Add(new Circle{Point{2, 2}, 2.0, Color{100, 100, 100}});
    dwf.getShapeStack()[0]->move(Point{2, 2});
    dwf.getShapeStack()[0]->rotate_deg(45.);
    dwf.Add(new Triangle{Point{4, 3}, Point{7, 9}, Point{8, 1}, Color{255, 0, 0}});
    dwf.SafeAs("test.ppm");
}

To see something of the flexibility, we also created a Circle class, derived from Shape.
CHAPTER 24. HANDS-ON EXERCISES FOR PART 4

// Circle.h
#ifndef CIRCLE_
#define CIRCLE_
#include "Shape.h"
class Circle final : public Shape {
public:
  Circle(Point center, double radius, Color col);
  virtual Circle& rotate_deg(double angle) override;
  virtual bool containsPoint(Point const& p) const override;
  virtual Point getCenterOfMass() const override;
  virtual Circle& move(Point const& p) override;
  double getRadius() const;
private:
  Point pa;
  double rad;
};
std::ostream& operator<<(std::ostream& os, Circle const& p);
#endif

// Circle.cxx
#include "Circle.h"
Circle::Circle(Point center, double radius, Color col)
  : Shape{col}, pa{center}, rad{radius} {}
Circle& Circle::rotate_deg(double angle) {return *this;}
Point Circle::getCenterOfMass() {return pa;}
Circle& Circle::move(Point const& p) {
  pa+=p;
  return *this;
}
double Circle::getRadius() {return rad;}
bool Circle::containsPoint(Point const& p) {
  if(distance(pa,p)<rad) return true;
  return false;
}
std::ostream& operator<<(std::ostream& os, Circle const& p) {
  os << "Circle: " << p.getCenterOfMass() << " radius: " << p.getRadius();
  return os;
}

Task:  To see how this flexible drawing works, you can introduce a user request input loop, where on users hitting a t a triangle is added the DrawFW’s stack, and by pressing c, a circle is added. The values needed for the constructors of the different shapes can also be read-in from the standard input. In that way, you can experience how dynamic memory management works.

Task:  Add other shapes like e.g. squares or rectangles. Inherit correctly from Shape! You can use the Point class (and specifically its operators) to simplify your life!

Hint:  The CMakeLists.txt for the above program is as follows.

```
make_minimum_required (VERSION 3.5)
project (DrawFW)
include_directories("${PROJECT_SOURCE_DIR}/include")
add_executable (drawfw src/main.cxx src/XYimage.cxx src/Color.cxx
  src/Point.cxx src/Triangle.cxx src/Circle.cxx
  src/DrawFW.cxx src/Shape.cxx)
```

24.2.2 Visualization – Image Processing

For this exercise, please recycle XYimage and Color from previous exercises!
Task: Add a constructor to XYimage, which takes a file name (std::string) as argument. With this constructor, read an image from a file!

Hint: First, download a PNG file from the internet! Transform it into a PPM via

```sh
$ convert file.png file.ppm
$ display file.ppm
```

The latter is for checking that everything is fine so far. Now, read the PPM file into your XYimage, and immediately save it again as PPM (with a different name, of course)! Check via display whether this was done correctly!

Task: Add member functions to XYimage and Color (what you need) to accomplish a Random Noise – just multiply a random value (around 1 – so a Gaussian with mean 1 and standard deviation 0.5 could do it) to the current color values of the image.

Hint: As this may not be accomplished in-place, it is a good occasion to exercise the copy-construction! Consider memcpy from the cstring header to copy one array to another.

Figure 24.1 shows an example result.

![Figure 24.1: Result from image processing. Left: original, Right: with noise.](image)

24.3 Numeric – PDE Solver (Tutorial)

Partial Differential Equations (PDEs) come in many flavors, such that it is hopeless to introduce methods for all of them in general. Even if we limit ourselves to PDEs of the simple time-evolution form

\[ \frac{\partial}{\partial t} f = \mathcal{F}(f, x, t), \]

where \( f \) is a vector-valued function of time \( t \) and space-location \( x \) (1D, 2D, or 3D), and \( \mathcal{F} \) a spatial differential operator, which are mostly found in engineering and science, the amount of equations is still enormous. And the methods for their solution (finite differences (FD), finite volume (FV), finite elements (FE), discontinuous Galerkin (DG), spectral, ... each of them with many variants like different orders, fluxes (FV), ansatz functions, Fourier transforms (or others), ...) is still too vast to be handled here in any reasonable comprehension. And this first covers the spatial discretization. For the time discretization, again you may have many variants for a handling (FD implicit, explicit, first or higher order; ADER (arbitrary derivative); etc.).

But most of these numerical schemes end up in matrix operations (iterative methods, or one-step; linear equation solvers, or projectors, ...). And matrix handling, we had already.

We will try to exercise the procedure using the Swift-Hohenberg Equation, which represent a model
equation for convection instabilities in thin liquid layers, heated from below (see also Marangoni effect). We use the following form of the equation,

$$\frac{\partial}{\partial t} \psi = [\varepsilon - (1 + \Delta)^2] \psi + A \psi^2 - \psi^3,$$

where

- \( \psi \) is a real function of time, \( t \), and 2D space coordinates, \( x, y \)
- \( \varepsilon \) is a control parameter; instability of solution \( \psi = 0 \) occurs for \( \varepsilon > 0 \)
- \( A \) is another parameter, which determines, whether roles (\( A = 0 \)) or hexagonal convection patterns (e.g. \( A = 0.5 \)) occur (\( A < 1 \)); see Figure 24.2
- \( \Delta \) is the 2D Laplacian, \( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \)

For simplicity, we choose \( \psi = 0 \) as initial condition (with some noise to accelerate the occurrence of the instability). Further, we choose periodic boundary conditions, i.e. \( \psi(x + L, y) = \psi(x, y) \) for some system size \( L \). And ditto for the \( y \)-direction.

**Theoretical Background of Solution**

From a so-called linear stability analysis, one can determine that the homogeneous stationary solution \( \psi = 0 \) becomes unstable if \( \varepsilon > 0 \). For \( A = 0 \) (or at least rather small), roles become the stable configuration. If \( A \) sufficiently large, hexagons become the dominant structure.

The characteristic wave number is equal to the critical one, and here by construction equal to one unit length \( (\varepsilon - (k_{cr} + \Delta)^2); k_{cr} = 1 \). The corresponding wavelength is \( \lambda_{cr} = 2\pi/k_{cr} = 2\pi \).

![Figure 24.2: Solutions of the Swift-Hohenberg equation. Left: Roles (\( \varepsilon = 0.1, A = 0.0 \)); Right: Hexagons: (\( \varepsilon = 0.1, A = 0.5 \)). Array size is 256 × 256, with \( \Delta x = 0.5 \) length units (so, total area is 128 × 128 unit length squared); critical wavelength is \( 2\pi \) per unit length (i.e. \( k_{cr} = 1 \); so around 20 roles or hexagons per 128 unit length should appear).](image-url)

**Numerical Analysis – Discretization and Solution**

We now choose a square region of size \( L \times L = 128 \times 128 \) unit length squared, into which fit sufficiently wavelength, i.e. structures, we would like to see. Furthermore, is this size large enough such that boundary effects can be neglected.
For our analysis, we investigate $\varepsilon = 0.1$, $\Delta x = 0.5$ (spatial resolution), $\Delta t = 0.05$ (time step resolution; implicit methods!), and $A = 0$ (roles) and $A = 0.5$ (hexagons). This means, we deal with arrays of $M \times M = 256 \times 256$ pixels.

We use a finite difference method here (even if others are conceivably better), where the spatial differential operators are discretized directly (see Finite Difference Schemes). Specifically, we set $x_i = i \cdot \Delta x$, $y_j = j \cdot \Delta x$, and $\psi_{i,j} = \psi(x_i, y_j)$ (later, a time index will be added). So, for the linear part (where the differential operators are included), we rewrite

$$(\varepsilon - 1 - 2\Delta - \Delta^2)\psi$$

and use the following discretization schemes,

$$(\Delta \psi)_{i,j} = \frac{1}{(\Delta x)^2} (\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1} - 4\psi_{i,j}),$$

$$(\Delta^2 \psi)_{i,j} = \frac{1}{(\Delta x)^4} \left( (\psi_{i+2,j} + \psi_{i-2,j} + \psi_{i,j+2} + \psi_{i,j-2}) - 8(\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}) 
+ 2(\psi_{i+1,j+1} + \psi_{i-1,j+1} + \psi_{i+1,j-1} + \psi_{i-1,j-1}) + 20\psi_{i,j} \right).$$

The two indexes may be a bit confusing. But if we introduce a so-called multi-index,

$$l = i + Mj, \quad l \in [0, M^2 - 1]$$

and introduce the vector of size $M^2$, $\psi_l$, the spatial differential operator becomes a matrix-vector operation,

$$(\varepsilon - 1 - 2\Delta - \Delta^2)\psi \approx \sum_{k=0}^{M^2-1} Q_{kl}\psi_l.$$ 

Inserting this into the PDE, we obtain

$$\frac{d\psi_k}{dt} \approx \sum_{k=0}^{M^2-1} Q_{kl}\psi_l + NL(\psi_k), \quad k = 0, \ldots, M^2 - 1.$$ 

$NL(x)$ is the non-linear part of the equation,

$$NL(x) = (A - x)x^2.$$ 

We refer to $Q$ as the system matrix, and use the following $\psi$ also to denote the vector. $Q$ is a very sparse matrix, i.e. it has really only few non-zero elements out of the $M^4$ elements. So, writing this equation a bit shorter in vector form, we obtain

$$\frac{d\psi}{dt} = Q\psi + NL(\psi).$$

The effect of the spatial discretization is that we now only have to deal with a system of ordinary differential equations (ODE). There are now several options how to solve it.

**Explicit Scheme**

We could directly employ BOOST ODEINT (see last part) and try to solve this system of equations fully explicitly – even if higher order in time. This would require the time steps usually to be rather small, and it is difficult to get it stably running this explicit way.
Implicit Scheme

So, let us focus on (semi-) implicit methods. We now discretize also in time via (first order, Euler),
\[
\frac{d\psi}{dt} \approx \frac{\psi(t + \Delta t) - \psi(t)}{\Delta t} = \frac{\psi^{t+\Delta t} - \psi^{t}}{\Delta t} .
\]
And we put the linear part of the right-hand side into the new time step, i.e.
\[
\frac{\psi^{t+\Delta t} - \psi^{t}}{\Delta t} = Q\psi^{t+\Delta t} + NL(\psi^{t}) .
\]
When we rearrange this a bit, we obtain the following algebraic linear(ized) system of equations,
\[
(1 - \Delta t \cdot Q) \psi^{t+\Delta t} = \psi^{t} + \Delta t \cdot NL(\psi^{t}) ,
\]
which needs to be solved recursively for each time step, starting with the initial condition $\psi^{0}$. The matrix $(1 - \Delta t \cdot Q)$ is sparse and (depending on $\Delta t$) diagonal dominant. And in principle, it should be symmetric, too.

A direct LU-decomposition will presumably fail (or take rather long) because of the size of the system. Furthermore, a complete LU decomposition will generate a rather fully occupied matrix, which might cause memory consumption issues. An iterative scheme such as Jacobi, or better Gauss-Seidel (see last part), is probably better suited here. Specifically so as the solutions should not change all too much from time-step to time step.

We leave it to you to realize such an iterative scheme (see [33]).

Implicit Spectral Scheme

But we propose here a slightly different, simpler scheme (again a one step solution per time step). We start again with the full PDE. We apply now a 2D spatial Fourier transformation (FT), and obtain (using linearity of the FT)\footnote{We denote the Fourier transformed quantities with a tilde},
\[
\frac{\partial \tilde{\psi}}{\partial t} = \frac{\partial \tilde{\psi}}{\partial t} = (\varepsilon - (1 - k^2)^2)\tilde{\psi} + \tilde{NL(\psi)} .
\]
This is so far still exact! The derivation operators are all transformed in multiplication with some sort of $k$. Background is that
\[
\psi(x) \sim \int e^{-ikx}\tilde{\psi}(k) \, dk
\]
and thus
\[
\frac{\partial}{\partial x} \psi(x) \sim \int \frac{\partial}{\partial x} e^{-ikx}\tilde{\psi}(k) \, dk = \int (-ik)e^{-ikx}\tilde{\psi}(k) \, dk .
\]
Of course, in 2D this goes exactly the same way!

Next, we introduce the time-discretization, where the linear part of the right-hand side is again treated implicit, and obtain,
\[
(1 - \Delta t(\varepsilon - (1 - k^2)^2)) \tilde{\psi}^{t+\Delta t} = \tilde{\psi}^{t} + \Delta t \cdot \tilde{NL(\psi^{t})} .
\]
This scheme would require to start in the $(x,y)$ space with $\psi^{0}$. This, and $NL(\psi^{0})$ are added and Fourier transformed. In the Fourier $(k_x, k_y)$ space, we solve
\[
\tilde{\psi}^{t+\Delta t} = \frac{\tilde{\psi}^{t} + \Delta t \cdot \tilde{NL(\psi^{t})}}{1 - \Delta t(\varepsilon - (1 - k^2)^2)} ,
\]
and transform the $\tilde{\psi}^{t+\Delta t}$ back into the $(x,y)$ space, to obtain $\psi^{t+\Delta t}$. This is then used again for the next time-step, etc.
As we have a discretized scheme, we need to account for this discreteness by using a discrete Fourier transform (DFT). This can be accomplished using the FFTW3 library. Furthermore, we need to discretize the \( k^2 \) and \( k^4 \) values. Without derivation, we present the result.

\[
(k^2)_{ij} = \frac{2}{(\Delta x)^2} \left( 2 - \cos \left( \frac{2\pi}{M} i \right) - \cos \left( \frac{2\pi}{M} j \right) \right) \xrightarrow{M \to \infty} \frac{4\pi^2}{L^2} (i^2 + j^2),
\]

\[
(k^4)_{ij} = \frac{2}{(\Delta x)^4} \left( \cos \left( \frac{4\pi}{M} i \right) + \cos \left( \frac{4\pi}{M} j \right) + 4 \cos \left( \frac{2\pi}{M} i \right) \cos \left( \frac{2\pi}{M} j \right) \right)
- 8 \left\{ \cos \left( \frac{2\pi}{M} i \right) + \cos \left( \frac{2\pi}{M} j \right) \right\} + 10 \xrightarrow{M \to \infty} \frac{16\pi^4}{L^4} (i^2 + j^2)^2.
\]

Realization

To realize such a numerical scheme as a C++ program, we first have to think about a structure. We want to split different tasks and responsibilities into different classes. And we would like to gain some flexibility concerning which equation we would like to solve, and which solver to use. And also, we would like to accept different initial conditions. Those there will become three abstract classes.
As you can see, all three have purely virtual classes, and thus cannot be instantiated. We must derive from them concrete classes. But this makes sense, as we want to add our own equations, initial conditions, and also solvers. We must only serve the interface. Of course, this flexibility does not guarantee that we can use each solver for each equation. And for sure, this is not the only possibility to realized a framework for solving PDEs!

Some word about `Equation`. While the choice of class members in `InitialCond` and `Solver` is hopefully comprehensible, `Lin` and `Link` may represent some riddle. Both return the linear part of the equation – `Lin` in the normal space, `Link` in the Fourier space. Solvers may require the one or the other - but usually not both. In order that an `Equation` is usable for an arbitrary solver, both member functions must be implemented.

Next, we want a more general handle to a PDE, which takes objects of concrete classes of the above classes, to organize and coordinate the solution process. But for flexibility, we want that only dependencies to the base classes are there! We thus introduce a PDE class, depending only on the three classes above.

```cpp
// PDE.h -------------------------------------------- ----------
#ifndef PDE_
#define PDE_
#include "Equation.h"
#include "Solver.h"
#include "InitialCond.h"
#include "MyArray.h"
class PDE {
public:
    PDE(Solver& sol, InitialCond const& init);
    void SetInitialConditions(InitialCond const& init);
    void WriteToVTK(std::string filename);
    void DoTimeSteps(double dt, int NumSteps=1);
private:
    Solver& sl;
    MyArray psi;
};
#endif

// PDE.cxx ------------------------------------------ ----------
#include "PDE.h"
#include <iostream>
#include <stdexcept>
PDE::PDE(Solver& sol, InitialCond const& init)
    : sl(sol), psi{{sl.eq.getSizeX(),sl.eq.getSizeX(),1},
        {sl.eq.getDX(),sl.eq.getDX(),sl.eq.getDX()},
        {0,0,0},"psi"} {
    SetInitialConditions(init);
}
void PDE::SetInitialConditions(InitialCond const& init) {
    init.generate(psi.va);
    sl.Init(psi.va);
}
void PDE::WriteToVTK(std::string filename) {
    return psi.writeToVTK(filename);
}
```
This class shall also handle all the data. So, we use our MyArray class from above (see Paraview example), from which we can write out immediately to Paraview-readable files.

```cpp
void PDE::DoTimeSteps(double dt, int NumSteps) {
    std::cout << "Solve " << NumSteps << " steps ...\n";
    sl.solve(dt,NumSteps,psi.va);
}
```

This class shall also handle all the data. So, we use our MyArray class from above (see Paraview example), from which we can write out immediately to Paraview-readable files.

```cpp
// MyArray.h ---------------------------------------- ----------
#ifndef MYARRAY_
#define MYARRAY_
#include <array>
#include <string>
#include <vector>
struct MyArray {
    MyArray(std::array<int,3> dim, std::array<double,3> spacing,
        std::array<double,3> origin, std::string FieldName);
    double& operator()(int i, int j, int k);
    double operator()(int i, int j, int k) const;
    void writeToVTK (std::string filename);
    std::array<int,3> n;
    std::array<double,3> dx, org;
    std::vector<double> va;
    std::string fieldname;
};
#endif

// MyArray.cxx -------------------------------------- ----------
#include "MyArray.h"
#include <algorithm>
#include <fstream>
MyArray::MyArray(std::array<int,3> dim,
    std::array<double,3> spacing,
    std::array<double,3> origin,
    std::string FieldName) :
    n(dim),dx(spacing),org(origin),va(n[0]*n[1]*n[2]),
    fieldname(FieldName)
{}
double& MyArray::operator()(int i, int j, int k) {
    return va[i+n[0]*(j+n[1]*k)];
}
double MyArray::operator()(int i, int j, int k) const {
    return va[i+n[0]*(j+n[1]*k)];
}
void SwapEndianness(double& var) {
    char varArray = reinterpret_cast<char*>(&var);
    for (long i=0; i<static_cast<long>(sizeof(var)/2); i++)
        std::swap(varArray[sizeof(var)-1-i],varArray[i]);
}
void MyArray::writeToVTK(std::string filename) {
    std::ofstream ofs(filename);
    ofs << "# vtk DataFile Version 2.0\n";
    /* for binary write */
    /* for ASCII write */
    /* for ASCII write */
    /* for ASCII write */
    /* for ASCII write */
    /* for ASCII write */
    /* for ASCII write */
};
```
for(int z=0; z<n[2]; ++z)
    for(int y=0; y<n[1]; ++y)
        for(int x=0; x<n[0]; ++x) {
            double val = (*this)(x,y,z);
            /* for binary write */
            // SwapEndianness (val);
            // ofs.write(reinterpret_cast<char*>(&val),sizeof(double));
            /* for ASCII write */
            ofs << val << "\n";
        }
ofs << "\n";
s.close();
}

That’s actually it! We now only need to concretize the above three classes, and start the simulation. So, let’s start with the equation (we call the derived class SHG for Swift-Hohenberg-Gleichung).

```cpp
#include "Equation.h"
class SHG : public Equation {
public:
    SHG(double epsilon, double A, double deltax, int size);
    double Lin(std::vector<double> const& x, int i, int j=0, int k=0) const override;
    double Link(std::vector<double> const& x, int i, int j=0, int k=0) const override;
    double NL(double x) const override;
private:
    int mi(int i, int j) const;
    int mib(int i, int j) const;
    double eps, a;
};
```

```cpp
SHG::SHG(double epsilon, double A, double deltax, int size)
    : Equation{deltax,size}, eps{epsilon}, a{A} {};
int SHG::mib(int i, int j) const {
    int sz = Equation::getSizeX();
    if(i<0) i+=sz;
    if(j<0) j+=sz;
    if(i>=sz) i-=sz;
    if(j>=sz) j-=sz;
    return i*sz+j;
}
int SHG::mi(int i, int j) const {
    int sz = Equation::getSizeX();
    return i*sz+j;
}
double SHG::Lin(std::vector<double> const& x, int i, int j, int k) const {
    double dx = Equation::getDX();
    int sz = Equation::getSizeX();
    double dx2 = 1./(dx*dx), dx4 = dx2*dx2;
    double cc = eps-1.+8.*dx2-20.*dx4;
    double cp = -2.*dx2+8.*dx4;
    double cpd = -2.*dx4;
    int m = i*sz+j;
    double ret = cc*x[m];
    if((i<2 || i>=sz-2) || (j<2 || j>=sz-2)) {
        ret += cp*(x[m+i+1]+x[m+i-1]+x[m+i+j+1]+x[m+i+j-1])
            + cpd*(x[m+i+j+1]+x[m+i+j-1]+x[m+i-1+j+1]+x[m+i-1+j-1]);
    }
    return ret;
}```
24.3. NUMERIC – PDE SOLVER (TUTORIAL)

```cpp
+ x[mib(i-1,j-1)]
- dx4*(x[mib(i+2,j)]+x[mib(i-2,j)]+x[mib(i,j+2)]
+ x[mib(i,j-2)]);
else {
    ret += cp*(x[mi(i+1,j)]+x[mi(i-1,j)]
+ x[mi(i,j+1)]+x[mi(i,j-1)]
+ dx4*(x[mi(i+2,j)]+x[mi(i-2,j)]+x[mi(i,j+2)]
+ x[mi(i,j-2)]);
}
return ret;
}

double SHG::Link(std::vector<double> const x, int i, int j, int k) const {
    using std::cos;
    const double pi2 = 2.*M_PI;
    double dx = Equation::getDX();
    double Mk = 2.*M_PI/Equation::getSizeX();
    double k2 = 2.*(2.-cos(Mk*i)-cos(Mk*j))/(dx*dx);
    double k4 = 2.*(cos(2*Mk*i)+cos(2*Mk*j)+4.*cos(Mk*i)*cos(Mk*j)
- 8.*(cos(Mk*i)*cos(Mk*j)+10.))/(dx*dx*dx*dx);
    return eps - 1. + 2.*k2 - k4;
}

double SHG::NL(double x) const {
    return (a-x)*x*x;
}
```

Also initial conditions can be realized rather easy. That is just random noise around a mean value. The information of the array size is taken from MyArray (psi).

```
#include "InitRandomUniformNoise.h"

class InitRandomUniformNoise : public InitialCond {
public:
    InitRandomUniformNoise(double mean, double delta, int seed);
    void generate(std::vector<double>& v) const override;
    void setSeed(int seed);
private:
    double mean, delta;
    int seed;
};
```

```
InitRandomUniformNoise::InitRandomUniformNoise(double mean, double delta, int seed) :
    mean(mean), delta(delta), seed(seed)
{}
void InitRandomUniformNoise::setSeed(int theSeed) {
    seed = theSeed;
}
void InitRandomUniformNoise::generate(std::vector<double>& v) const {
    std::mt19937 gen(seed);
    std::uniform_real_distribution<> dis(mean-0.5*delta, mean+0.5*delta);
    std::generate(v.begin(),v.end(),[&gen,&dis](){return dis(gen);});
    double sum = std::accumulate(v.begin(),v.end(),0.0)/v.size();
    std::transform(v.begin(),v.end(),v.begin(),...);
```
Finally, we need a solver. Realizing the spectral implicit method above, we call it SpecIm. Furthermore, we outsource also the FFT, thereby wrapping the FFTW.

```cpp
// SpecIm.h -----------------------------------------
#ifndef SPECIM_
#define SPECIM_
#include "Solver.h"
#include "FFT_2D.h"
class SpecIm : public Solver {
public:
    SpecIm(Equation& eq);
    void Init(std::vector<double>& x) override;
    void solve(double TimeStep, int NumSteps,
               std::vector<double>& x) override;
private:
    FFT_2D fft;
};
#endif
// SpecIm.cxx ---------------------------------------
#include "SpecIm.h"
#include <iostream>
SpecIm::SpecIm(Equation& eq) : Solver{eq}, fft(eq.getSizeX(), eq.getSizeX()) {}
void SpecIm::Init(std::vector<double>& x) {
    fft.SetInput(&x[0]);
}
void SpecIm::solve(double TimeStep, int NumSteps,
                   std::vector<double>& x) {
    int M = eq.getSizeX();
    double dummy;
    fftw_complex *out;
    for(int k=0; k<NumSteps; ++k) {
        for(auto &p: x) p+=TimeStep*eq.NL(p);
        out = fft.forward();
        for(int i=0; i<M/2+1;++i)
            for(int j=0; j<M;++j) {
                dummy = TimeStep*eq.Link(x,i,j);
                out[i+j*(M/2+1)][0]/=(1.-dummy);
                out[i+j*(M/2+1)][1]/=(1.-dummy);
            }
        fft.backward();
    }
}

// FFT_2D.h -----------------------------------------
#ifndef FFT_2D_
#define FFT_2D_
#include <fftw3.h>
class FFT_2D {
public:
    FFT_2D(int N0, int N1);
    FFT_2D();
    fftw_complex* forward();
    void backward();
    void SetInput(double * inp);
private:
    fftw_complex* fsp = nullptr;
    double *in = nullptr;
    fftw_plan pf, pb;
    int N, M;
};
#endif
```
24.3. NUMERIC – PDE SOLVER (TUTORIAL)

// FFT_2D.cxx ----------------------------------------
#include "FFT_2D.h"
#include <cmath>
#include <iostream>
FFT_2D::FFT_2D(int N0, int N1) : N{N0}, M{N1} {
    fsp = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(N/2+1)*M);
}
FFT_2D::~FFT_2D() {
    fftw_destroy_plan(pf);
    fftw_destroy_plan(pb);
    fftw_free(fsp);
}
fftw_complex* FFT_2D::forward() {
    fftw_execute(pf);
    return fsp;
}
void FFT_2D::backward() {
    fftw_execute(pb);
    double N2 = N*M;
    for(int i=0; i<N2; ++i) in[i]/=N2;
}
void FFT_2D::SetInput(double * inp) {
    in=inp;
    pf = fftw_plan_dft_r2c_2d(N,M,in,fsp,FFTW_ESTIMATE);
    pb = fftw_plan_dft_c2r_2d(N,M,fsp,in,FFTW_ESTIMATE);
}

By putting everything in a separate class, each class by itself becomes rather comprehensible. Finally, we only need the main(), which sets up all the ingredients.

// main.cxx -----------------------------------------
#include "PDE.h"
#include "SHG.h"
#include "SpecIm.h"
#include "InitRandomUniformNoise.h"
#include <iostream>
int main() {
    InitRandomUniformNoise irn {0.,1.e-4,3134}; //mean,delta,seed
    SHG eq {0.1, 0.5, 0.5, 256}; //eps,A,dx,N
    SpecIm sol{eq};
    PDE pde {sol, irn};
    pde.WriteToVTK("test_0.vtk");
    for(int i=1; i<=50; ++i) {
        std::cout << "Step " << i << "\n";
        pde.DoTimeSteps(0.05,100);
        pde.WriteToVTK(std::string("test_"+std::to_string(i)+".vtk"));
    }
}

As the system was designed, initial conditions and equations can be set up independently from each other. A solver requires already information from an equation. And the PDE framework puts all the things together. The rest is “only” to perform the time step iterations.

We also give the CMakeLists.txt for completeness.

cmake_minimum_required (VERSION 3.5)
project (PDESolve)
set(CMAKE_CXX_STANDARD 11)
include_directories(${{CMAKE_CURRENT_SOURCE_DIR}/include})
add_executable(pdesolve src/main.cxx src/Equation.cxx src/SHG.cxx
src/InitRandomUniformNoise.cxx src/MyArray.cxx
src/FFT_2D.cxx src/SpecIm.cxx src/PDE.cxx)
target_link_libraries(pdesolve fftw3)
Task: Realize another solver – the iterative Gauss-Seidel solver described above!
Of course, you can also try to implement different equations or initial conditions! Do you see a reasonable way to abstract also the boundary conditions? But of course without spoiling the performance!

24.4 Boost – Named Function Arguments (Tutorial)

Often, functions with long parameter lists are hard to read and use. Of course, one could limit the argument list of a function by introducing a structure, which contains the parameters. But how to handle names? We copy here from the chapter about boost.parameter (from [34]). More information can also be found on the Boost Docu Page: The Boost Parameter Library.

Imagine, you want to write and use a more complex function with a lot of arguments. Wouldn’t it be nice to call it e.g. via the following?

```cpp
double val = func(pi=3.1415,b='A',c=2, sname="Name", e=true);
```

Any attempt to write a function

```cpp
double func(double pi,char b,int c, string sname, bool e) {
    /* do something and return a double */
}
```

will fail to compile. Even worse, if you try to rearrange the arguments, you will earn type conversion errors.

```cpp
double val = func(b='A', e=true, c=2, pi=3.1415, sname="Name");
```

But boost.parameter can help here. Don’t be scared about the clutter involved in that.

```cpp
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(pi)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(sname)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
    (double),
    func,
    tag,
    (required
        (pi, (double))
        (b, (char))
        (c, (int))
        (sname, (std::string))
        (e, *)
    )
)
std::cout.setf(std::ios::boolalpha);
std::cout << pi << ' ' << b << ' ' << c << ' ' << sname << ' ' << e << '
';
return pi;
}

int main() {
    double val;
    val=func(_pi=3.1415,_b='B',_c=3,_sname="Name",_e = true);
    val=func(_c=3,_pi=3.1415,_sname="Name",_b='B',_e = true);
}
```

First of all, each later function argument name must be uniquely declared. That’s done using BOOST_PARAMETER_NAME. Next, a BOOST_PARAMETER_FUNCTION is declared, with return value double, and name func. tag is not important now. The next section consists of parentheses (…) with a required keyword, and the
key/value pair list of function arguments and corresponding types. Afterwards, in usual braces {...}.

The normal function body starts, where the parameter names can be used – e.g. here to output them.

Two things are noteworthy here. First, the `require` keyword. It determines that the following parameters are mandatory to be specified. We see in a moment an alternative. Second, `e` is paired with the type `*`. This means, what you intuitively already anticipated: The type will be deduced. So, changing the above calls to

```cpp
val=func(_pi=3.1415,_b='B',_c=3,_sname="Name",_e = true);
val=func(_c=3,_pi=3.1415,_sname="Name",_b='B',_e = "Some Text");
```

will work as well – even although we set it before to `bool`!

Please, notice the underscore for the argument names when in use. This is unavoidable!

Now, let us say that `c`, `sname`, and `e` could live with some default values. This is done using the keyword `optional` as follows.

```cpp
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(pi)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(sname)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (double),
  func,
  tag,
  (required
   (pi, (double))
   (b, (char)))
  (optional
   (c, (int), 3)
   (sname, (std::string), "name")
   (e, *, true))
) {
  std::cout.setf(std::ios::boolalpha);
  std::cout << pi << "\n"
   << b << '\n'
   << c << '\n'
   << sname << '\n'
   << e << '\n';
  return pi;
}

int main() {
  double val;
  val=func(_pi=3.1415,_b='B',_c=4);
  val=func(_e="name",_b='A',_pi=3.1415);
}
```

As you can see, you don’t need to specify `c`, `sname`, and `e`. The defaults are used in that case. But you can overwrite them, if needed. And `e` even keeps its flexibility to change the type!

**Task:** User `BOOST_PARAMETER_MEMBER_FUNCTION` and `BOOST_PARAMETER_CONST_MEMBER_FUNCTION` instead of `BOOST_PARAMETER_FUNCTION` in order to define class member functions with named arguments!

### 24.5  Algebra – Square Matrix and Vector

**Task:** In the previous part, we stumbled over the problem on how to formulate consistent Matrix-Vector operation, i.e. specifically on how to ensure that both have the correct size. Although there is
CHAPTER 24. HANDS-ON EXERCISES FOR PART 4

A better solution with templates, try to create a consistent framework for matrix-matrix, matrix-vector and vector-vector operations, such that only objects of the correct size and shape can be multiplied, added, ...!

**Remark:** Accomplishing consistency in this way is rather complicated, because it is done intrinsically during run-time. It is usually not necessary to have this flexibility during run-time. And therefore, templates are a better solution.

### 24.6 Statistics – CERN ROOT Fitting (Tutorial)

As you have seen in Part 2, fitting parameters to a histogram is an often recurring topic. We would like to show the ROOT’s versatility in that respect. In part 2 was also the example of energy straggling. In reality, when measuring energy deposits in a sensor, the actual model is not a pure Landau distribution, but a convolution (say, integral transformation) with a Gaussian – the latter represents some random noise (from electronics, e.g.).

\[
GL(x; MPV_L, \sigma_L, \sigma_G) = \int_{-\infty}^{\infty} \text{Landau}(s; MPV_L, \sigma_L) \cdot \text{Gaus}(s; x, \sigma_G) \, ds.
\]

Herein, \(MPV_L\) is the most probable value of the Landau distribution (position parameter). \(\sigma_L\) is the scale parameter of the Landau distribution.\(^8\) The Gaussian as written above is proportional to

\[
\text{Gaus}(s; x, \sigma_G) \sim \exp\left\{ -\frac{(x - s)^2}{2\sigma_G^2} \right\}.
\]

We’ve just put the \(x\) value in the place of the mean parameter. \(\sigma_G\) is thus the standard deviation.

In the following code snippet, we create a function object, GL, which we use (in lack of real data) as data sampling source, and as fit function.

```cpp
#include "TH1D.h"
#include "TCanvas.h"
#include "TF1.h"

struct GL {
TF1 gala;
GL() : gala(TF1("glcore","[0]*TMath::Landau(x,[1],[2],true)*TMath::Gaus(x,[3],[4],true)*",-100.,100.)) {}
double operator()(double *x, double *p) {
    // p0 : norm // p1 : MPV Landau
    // p2 : Sigma Landau // x : Mean Gaus
    // p3 : Sigma Gaus
    gala.SetParameters(p[0],p[1],p[2],x[0],p[3]);
    return gala.Integral(x[0]-5.*p[3],x[0]+5.*p[3]);
}
};

int main() {
TF1 gl("gl","GL[0]*0.512.,4");
gl.SetParameters(1000.,130.,10.,13.);
TH1 h("h","ABC Channels",512,0.,512.);
h.FillRandom("gl",10000);
TCanvas c("c","c",800,600);
h.Draw(); // <- up to here from sample
// Fitting now
gl.SetParNames("norm","MPV_L","sigma_L","sigma_G");
gl.SetParameters(1000.,100.,20.,20.);
gl.FixParameter(0,h.GetEntries()*h.GetBinWidth(5));
gl.SetNpx(200);
h.Fit("gl","ML");
c.SaveAs("GLfit.pdf");
}
```

\(^8\)The Landau distribution can be normalized, but does not have any of its moments – mean, variance, etc.
As you can see, it is rather trivial to create also such function objects. We exploited that TF1 has an `Integral` class member (and the convolution is relevant only in the range of the Gaussian mean \( \pm 5\sigma \)). The `Fit` class member function of TH1D knows some more options. L and M mean that we fit with binned likelihood (instead of least square), and improve the fit result, respectively (see TH1 Fit for more information).

![Figure 24.3: Fitted Gauss-Landau Convolution.](image)

Btw. we introduced also a `norm`, to get the fit-function have the same integral as the histogram. From histograms, the integral is but know – it is just the number of entries times the bin width. So, this parameter should not become fitted. Reducing the number of fit parameters increases the chance for a successful and clean fit! In ROOT, you can say, which parameters are fixed. Figure 24.3 shows the fit result. The output from the fit looks like this.

```
FCN=235.784 FROM HESSE STATUS=OK 18 CALLS 161 TOTAL
EDM=2.19583e-10 STRATEGY= 1 ERROR MATRIX ACCURATE
EXT PARAMETER STEP FIRST
  NO. NAME VALUE ERROR SIZE DERIVATIVE
1 norm 1.00000e+04 fixed
2 MPV_L 1.30161e+02 2.95446e-01 5.64810e-04 4.85101e-05
3 sigma_L 9.94666e+00 2.27557e-01 7.02371e-05 -1.31968e-05
4 sigma_G 1.31225e+01 3.93985e-01 1.12776e-04 -7.88186e-07
ERR DEF= 0.5
```

But the fitted parameters can be obtained also directly from the histogram via TH1D::GetFunction and TF1::Parameters or TF1::Parameter.

**Caution:** Not all functions are however easy to fit – even in ROOT. If you don’t believe, try to fit a **Generalized Extreme Value Distribution** (GEV). Just as an outline: The idea is that if you take the maximum of the sample of fixed size \( N \) from an arbitrary distribution, it will be distributed according to a GEV distribution. That’s rather useful to know if you want to assess rare or extreme events (in contrast to normal events, for which we know the Central Limit Theorem, which always leads to a Gaussian).
Remark: As you could see from the examples, the function strings can contain also functions from the ROOT’s vast special functions repertoire. There are, for instance, also the Bessel functions TMath::BesselJ0 and TMath::BesselJ1.  

You can also use your own functions in function strings, as shown in the following code snippet – as long as you are working in cint, or using ROOT macros.  

```cpp
{  
  TF1 f("myf","x*exp(-x)",0.10);
  TF1 g("g","myf(x)*TMath::BesselJ0(x)",0,10);
  g.Draw();
}
```

With that flexibility, you can design any function for fitting you can conceive.  

Adding Remark: TF1 also has members GetMaximum, GetMaximumX, GetMinimum and GetMinimumX to obtain maximum/minimum value and position. Also derivatives can be easily used. Within cint and macros, one can extend the example above.

```cpp
{  
  [...] 
  TF1 h("h","g.Derivative(x)",0,10);
  h.Draw();
}
```

Please look for the TF1 Class Reference for more information! So, the example of the Gaus-Landau convolution can be realized also as follows (bla.C).

```cpp
{  
  TF1 gala("glcore","[0]*TMath::Landau(x,[1],[2],true)* TMath::Gaus(x,[3],[4],true)*","-100.,100.);
  TF1 gl("gl","[&](double*x, double*p){ gala.SetParameters(p[0],p[1],p[2],x[0],p[3]); return gala.Integral(x[0]-5.*p[3],x[0]+5.*p[3]);}" ,0,512.,4);
  gl.SetParameters(1000.,130.,10.,20.);
  TH1D h("h","ADC Channels",512,0.,512.);
  h.FillRandom("gl",10000);
  TCanvas c("c","c",800,600);
  h.Draw(); // <- up to here from sample
  // Fitting now
  gl.SetParNames("norm","MPV_L","sigma_L","sigma_G");
  gl.SetParameters(1000.,100.,20.,20.);
  gl.SetNpx(200);
  h.Fit("gl","ML");
  c.SaveAs("GLfit.pdf");
}
```

(Please note the line breaks! After \ must not be any white space or tab! But better put it again on a single line!)

What you can always to is to use std::function and Lambda expressions. The following code should compile. So far, that’s the most compact way we found to realize this issue.

```cpp
#include "TH1D.h"
#include "TCanvas.h"
#include "TF1.h"
#include <functional>
int main() {  
  TF1 gala("glcore","[0]*TMath::Landau(x,[1],[2],true)* TMath::Gaus(x,[3],[4],true)","-100.,100.);
  std::function<double(double*, double*)> f = 
    [gala](double*x, double*p){ gala.SetParameters(p[0],p[1],p[2],x[0],p[3]);
```

9Higher order Bessel functions can be generated via recurrence relations like \( J_{\lambda+1}(x) = \frac{2\lambda}{x} J_{\lambda}(x) - J_{\lambda-1}(x) \).

10ROOT macros are just cint sessions written to a file with an ending .C, and called via root -l <macrofile>.C (where <macrofile> must be replaced by your filename).
return gala.Integral(x[0]-5.*p[3],x[0]+5.*p[3]);
);
TF1 gl("gl",f,0.512,.4);
gl.SetParameters(1000.,130.,10.,13.);
TH1D h("h","ADC Channels",512,0.,512.);
h.FillRandom("gl",10000);
TCanvas c("c","c",800,600);
h.Draw(); // <- up to here from sample
// Fitting now

gl.SetParNames("norm","MPV_L","sigma_L","sigma_G");
gl.SetParameters(1000.,100.,20.,20.);
gl.FixParameter(0,h.GetEntries()*h.GetBinWidth(5));
gl.SetNpx(200);
h.Fit("gl","ML");
c.SaveAs("GLfit.pdf");

24.7 Optimization Session (Tutorial)

We illustrate a $gprof$ profiling session using the PDE solving program above. The preparations are minimal. From our build folder, we call $cmake$ with some options, to set the correct compile and linker flags. On GCC command-line, that’s essentially the option $-pg$ (but also the linker must be use with that flag!). Option $-g$ makes the output more readable.

```
$ cmake -DCMAKE_CXX_FLAGS="-pg -g" -DCMAKE_EXE_LINKER_FLAGS=-pg ..
```

Afterwards, we just need to run the program once (so, better a representative case for just few minutes instead of hours, days, or even weeks). This creates a file $gmon.out$, which contains the information about the program’s execution profile. The final call to $gprof$ is just to collect the statistics and represent it in a readable format. We redirect it into a file, and read it later using a pager.

```
$ ./pdesolve
$ ls -l gmon.out
$ gprof ./pdesolve gmon.out > analysis.txt
$ less -r analysis.txt
```

Now, we need to interpret what we see. The $gprof$ Tutorial might help. The first thing we see is the flat profile. We shorten it a bit.

**Flat profile:**

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>seconds</td>
<td>ms/call</td>
<td>ms/call</td>
</tr>
<tr>
<td>42.14</td>
<td>4.55</td>
<td>4.55</td>
<td>0.00</td>
</tr>
<tr>
<td>26.21</td>
<td>7.38</td>
<td>2.83</td>
<td>50</td>
</tr>
<tr>
<td>6.85</td>
<td>8.12</td>
<td>0.74</td>
<td>5000</td>
</tr>
<tr>
<td>5.14</td>
<td>8.68</td>
<td>0.56</td>
<td>327881611</td>
</tr>
<tr>
<td>3.66</td>
<td>9.47</td>
<td>0.40</td>
<td>327942144</td>
</tr>
<tr>
<td>3.06</td>
<td>9.80</td>
<td>0.33</td>
<td>655763222</td>
</tr>
<tr>
<td>1.81</td>
<td>10.00</td>
<td>0.20</td>
<td>165120003</td>
</tr>
<tr>
<td>1.76</td>
<td>10.19</td>
<td>0.19</td>
<td>131072</td>
</tr>
</tbody>
</table>
The first column gives us the time in percent of total run-time that a routine occupied. This refers to the third column, where the self-time is given (the useful work within the function, and not with in e.g. called functions – the latter is respected by the cumulative time). That’s quite clear here. Our class member function \texttt{SHG::Link} requires the most of the time. When you look into the code base, you see that this is the place, where the Fourier transformed linear operator is computed ($\varepsilon - 1 + 2k^2 - k^4$). This routine is called 165120000 times during the run. So, optimizing it will surely be beneficial.

The 5-th and 6-th column give you the elapsed time (in average) per call. A 0.00 does not say that it did not cost time! It only means that the execution time of this function is shorter than a hundredth of a millisecond. But obviously, \texttt{SHG::Link} was called so often, such that in the sum it really matters!

Finally, there is the call graph, which tells you which function is called by which function, and how often.

\textbf{Task:} 256$^2$ times size of a double (8 byte) are just half a Megabyte. In times of cheap memory, we should create lookup tables for $k^2$ and $k^4$, instead of calculating the values only (trigonometric functions are expensive!). Realize this and measure again! Did it help?

Also play around with \texttt{-O2} or \texttt{-O3} optimization options of the compiler (\texttt{CMAKE_CXX_FLAGS}). What do you observe?

There are many entries also for e.g. \texttt{bool __gnu_cxx::operator!=}. What is it?

\textbf{Task:} Analyze the call graph! Try to understand what information does it contain. Is there more potential for optimization?
Part V

Templates, Generic Programming and Design Patterns
Chapter 25

Templates

25.1 Motivation

Which programmer did not yet come across a function, which is supposed to return the maximum\(^1\) of two values. Imagine, we had done this the first time for a pair of integers.

```c++
int max(int x, int y) {
    return (x>y) ? x : y ;
}
int main() {
    // Imagine we had done this the first time for a pair of integers.
    double da=1.1, db=2.2;
    double dc = max(da,db);
    int ia=1, ib=2;
    int ic = max(ia,ib);
    std::cout << dc << " " << ic << "\n";
}
```

In applications, that is perfectly fine, compiles, and works. But now, the maximum of doubles is not quite correct. A useful compiler will complain, of course, that narrowing happens – from double to int. But that’s anyway not what you want and expect. So, you add another function.

```c++
double max(double x, double y) {
    return (x>y) ? x : y ;
}
```

The function overloading mechanism of C++ will do the rest. But it should be also quite natural, whether an integer is smaller or larger than a double. So, you write intuitively the following.

```c++
int main() {
    // But that's anyway not what you want and expect. So, you add another function.
    double dd = max(ib,db);  // compile error
}
```

But you get an error that the compiler cannot (or doesn’t want to) decide, which max function to use. Maybe your first natural reaction (possibly from experiences with other programming languages) is disambiguation.

```c++
int maxi(int x, int y) {
    return (x>y) ? x : y ;
}
double maxd(double x, double y) {
    return (x>y) ? x : y ;
}
int main() {
    // Maybe your first natural reaction (possibly from experiences with other programming languages) is disambiguation.
    ...
```

\(^1\)Or "minimum", or similar.
double dd = maxd(ib,db); // implicit cast
}

Well, this compiles. The overloading mechanism is removed. And we force (by implicit cast) one of
the functions to eat it.

But now, what if the int is indeed larger than the double? Should the return value then be of
type int then?! And at all, we cannot use anymore the same name max for the function, although it
is supposed to do the same thing. So, let’s get back to function overloading, and simply add another
function.

```cpp
int max(int x, int y) {
    return (x>y) ? x : y ;
}
double max(double x, double y) {
    return (x>y) ? x : y ;
}
double max(int x, double y) {
    return (x>y) ? x : y ;
}
int main() {
    [...]
    double dd = max(ib,db);
    [...]
}
```

That is now OK! It compiles, and works! Right?

Well, not exactly. We have now already three functions doing actually exactly the same thing. And
we just started with int and double. What about all the other built-in types in C++, which can be
comparable – including all the user-defined types? For all of those, we would have to write possibly
more max(...) functions!

This represents code multiplication, which perfectly violates the DRY (Don’t Repeat Yourself!) principle. In short: We don’t like it! We don’t want it!

So, what is the alternative?

### 25.2 Function Templates

C++ recognized already early in its history that a kind of type-templating could be a good idea. It
represents a similar idea of extensibility and (compile-time) polymorphism as function overloading\(^2\), on
which it builds. But templates can do more. Much more!

And so, another nomenclature was added.

```cpp
#include <iostream>

//template<typename T>
T max(T x, T y) {
    return (x>y) ? x : y ;
}

int main() {
    double da=1.1, db=2.2;
    double dc = max(da,db); // max<double>
    int ia=1, ib=2;
    int ic = max(ia,ib); // max<int>
    double dd = max(ib,db); // Error: ambiguous
    std::cout << dc << " " << ic << " " << dd << "\n";
}
```

So, there is a new keyword, template, that must be prepended to the function declaration, to mark
it as a function template. And a type placeholder is introduced, T, which must be declared as such a
placeholder by <typename T> behind the template keyword.\(^3\) Also typename is a keyword.\(^4\)

---

\(^2\)which is actually also used here

\(^3\)Of course, T should not be used for other variable names now – unless you like to obfuscate!

\(^4\)Instead, also class could be used; but this will be explained later
In the application, we got rid of the code duplication of the definition of \texttt{max}. And so far, we have made available \texttt{max} also for all types that are comparable by \texttt{operator\textgreater} -- also the user-defined types (you could try it with \texttt{strings}, for instance\footnote{But this will fail as the \texttt{<string>} header defines already its own \texttt{max} function. Oops!}).

But if we mix types, like \texttt{int} and \texttt{double} above, we still get an error. This is because the compiler could still resolve either to have \texttt{T} and \texttt{int} or a \texttt{double}. You can explicitly resolve by specifying the type at the specific place of using the function template.

```cpp
#include <iostream>
template<typename T>
T max(T x, T y) {
    return (x>y) ? x : y ;
}

int main() {
    [...]
    double dd = max<double>(ib,db); // <-- !!!
    std::cout << dc << " " << ic << " " << dd << "\n";
}
```

This is but not the only solution. Templates allow for more. For much more! First of all, we can introduce several template parameters. So, we add \texttt{S}. But now, which type to return? \texttt{C++ 11} introduced the \texttt{decltype} keyword (see \texttt{cppreference: decltype specifier}). It determines and returns the type of expressions during compile time (that type with a wider range is used).

```cpp
#include <iostream>
template<typename T, typename S>
auto max(T const& x, S const& y) -> decltype(x>y ? x : y) {
    return (x>y) ? x : y ;
}

int main() {
    double da=1.1, db=2.2;
    double dc = max(da,db); // double max<double,double>
    int ia=1, ib=2;
    int ic = max(ia,ib); // int max<int,int>
    auto dd = max(ib,db); // double max<int,double>
    std::cout << dc << " " << ic << " " << dd << "\n";
}
```

Return type deduction is a rather tough business (in \texttt{decltype}, you could use also \texttt{x+y}, here, too). And so we tell you that this is still not the end of the story. But this belongs to the more advanced stuff of \texttt{C++}.

Finally, we added \texttt{const} references as parameter types, in order to be \texttt{const}-correct, and efficient (call-by-reference instead of call-by-value). We can compare also user-defined data types. Due to \texttt{auto} and \texttt{decltype}, we are completely flexible and generic here, although we do not violate the strong type system of \texttt{C++}.

The idea of function templates as general type-templated source code reuse and compiler-automated function generation mechanism should by now be comprehensible for you, although you surely still need to become accustomed to the syntax. Mostly, you will use such functions from the \texttt{STL} or from other libraries. But familiarizing with this template syntax will be important also for other business, as will become obvious in the next sections.

**Exercise:** Write a corresponding \texttt{min} function, which returns the minimum of two comparable variables!

### 25.3 Class Templates

The same idea of templating also applies to classes. Consider, for instance, the following \texttt{Rational} class, which shall enable integral arithmetic on rational numbers.
class Rational {
public:
  Rational(int nominator, int denominator = 1) :
    nom(nominator), den(denominator) {}
  int getNominator() { return nom; }
  int getDenominator() { return den; }
private:
  int nom, den;
};

Rational r1 {2,5};

As for the functions in the previous section, the Rational class might operate on different integer types such as long int, unsigned int, short. The nomenclature is more or less the same as for function templates, although many programmers tend to use here class rather than typename.

template <class U>
class Rational {
public:
  Rational(U nominator, U denominator = 1) :
    nom(nominator), den(denominator) {}
  U getNominator() { return nom; }
  U getDenominator() { return den; }
private:
  U nom, den;
};

Rational<int> r1 {2,5};
Rational<unsigned short> r2 {2,5};

Which of both – class or typename – to take is mostly a question of preference. But typename is a bit more general when the business is about so-called template template parameters. So, feel free to use what you like more, until your compiler (i.e. the C++ standard) tells you to change your mind!

Please note that Rational<int> and Rational<unsigned short> are two different, independent types - fully unrelated to each other (unless you change this by conversion constructors and operators). So, they cannot be used interchangeably.

Very good other examples for type-templated classes can be found in the STL std::string, std::vector (and all containers may be mentioned), std::pair, etc. And the rules are exactly the same as for function templates. That is, you can introduce more than one template parameter if needed.

A very special subject is class-member function templateization (see cppreference: Member templates). But with the knowledge acquired to far, this should not be a problem for you anymore, either.

If, like in the following example, there is a templated class member (data or function), the template parameter of the outer class can be forwarded to the class member, as shown here for the std::vector<T>.

template <typename T>
class Stack {
private:
  std::vector<T> elems; // elements
public:
  void push(T const &); // push element
  void pop(); // pop element
  T top() const; // return top element
  bool empty() const { // return true if empty.
    return elems.empty();
  }
};

How to constrain the set of acceptable types is not within the scope of this course. But interested readers can look for cppreference: Type Traits.
Btw. you will often see code with a single letter as template parameter name. This is by no means necessary. You can write normal parameter names! Naming rules are the same as for normal variables (see part 1 of this course). But short names of template parameters is often practice and a good idea, because templates are otherwise already clutter enough.

25.4 Template Non-Type Arguments

C++ also allows template parameters, which are not types. Those are the so-called *non-type template arguments* (see also [cppreference: Template parameters and template arguments](https://en.cppreference.com/w/cpp/language/template_parametrization)). Any integral, pointer and enumeration data type can be used. But not (yet) floating point arguments! For example, imagine we need arrays of definite length. Of course, it’s not any complication at all to have this realized by the means we’ve learned so far. But sometimes, the fixing of such parameters during compile-time might be advantageous for optimizations.

Let’s consider an example.

```cpp
#include <iostream>
template <int N>
class A {
    public:
        A() {
            std::cout << "val=" << val
                << " size = " << size << "\n";
        }
    private:
        int val[N];
        int size=N;
    }; 
    int main() {
        A<4> a;
        A<5> b;
    }
```

As before, `a` and `b` are of different and incompatible types – `A<4>` and `A<5>`, respectively. There are cases, where this is exactly what you want (see Hands-on Vectors and Matrices).

You can also use non-type arguments in function templates, or together with other type template parameters.

```cpp
#include <iostream>
template <typename T, unsigned int N>
T power(T x) {
    T ret = x;
    for(unsigned int i=1; i<N; ++i) ret *= x;
    return ret;
}
int main() {
    std::cout << power<int,3>(3) << "\n";
    std::cout << power<double,2>(3.) << "\n";
}
```

Whether for `int` or for `double`, for whatsoever exponent – we wrote only one template! The compiler creates the special template functions for each specific type and non-type arguments when needed, i.e. when we require one by using it. The reduction in code size cannot be overemphasized, rendering the code maintainability much more easier! They are so compact in the code base! As templates are evaluated during compile-time, a lot of optimization potential can be exploited by compilers. This is the reason, why template libraries such as `boost` and the Standard Template Library (STL) have evolved and established.

---

7Remember `std::array`!
25.5 Template Specialization and Default Template Parameters

25.5.1 Template Specialization

Templates can also be specialized (see cppreference: Template specialization). Imagine that a templated class or function needs a different behavior for a specific type or non-type argument. This can be accomplished as follows.

```cpp
#include <iostream>
template <typename T, typename S>
class A {
  public:
    A() { std::cout << "no specialization\n"; }
  private:
    T v1;
    S v2;
  };
template <typename T>
class A<T, int> {
  public:
    A() { std::cout << "partial specialization\n"; }
  private:
    T v1;
    int v2;
  };
template <>
class A<
  public:
    A() { std::cout << "complete specialization\n"; }
  private:
    long v1; int v2;
  };
int main() {
  A<int,long> a; // no specialization
  A<double,int> b; // partial specialization
  A<long,int> c; // complete specialization
}
```

As can be seen, we specialized twice – once partially (line 10, and following), and once completely (line 18, and following). The partial specialization could be done with the other template parameter, too.

```cpp
[...]
template <typename T>
class A<double,T> {
  public:
    A() { std::cout << "partial specialization\n"; }
  private:
    double v1;
    T v2;
  };
[...]
```

But, of course, not at the same time with the former specialization, which would create an ambiguity that the compiler cannot resolve. So, there can be at most one specialization for each level (each level represented by the number of remaining parameters). Ambiguity must be avoided!

25.5.2 Default Template Parameters

You can also specify default template "values" – types for type parameters, or values for non-type arguments.

```cpp
#include <iostream>
#include <list>
#include <vector>
template <typename T, typename CONT = std::vector<T>>
struct Data { CONT mycont; };
```
25.5. TEMPLATE SPECIALIZATION AND DEFAULT TEMPLATE PARAMETERS

```cpp
int main() {
    Data<double> mydat1;
    mydat1.mycont.push_back(4.3);
    for(auto p : mydat1.mycont) std::cout << p << "\n";
    Data<double, std::list<int>> mydat2;
    mydat2.mycont.push_back(3.4);
    for(auto p : mydat2.mycont) std::cout << p << "\n";
}
```

As can be observed, there is a second type parameter, which defaults to std::vector<T> (with the same type T as the first template parameter). Written in that way, there is nothing that would prevent to create mydat2, which explicitly uses a list of integers, std::list<int>, whereas the first parameter is still double. To create consistency and coupling is more difficult to achieve.

In any case, if a template parameter has a default value, it can be skipped later. In this respect, template specialization and default template arguments work differently. Using the previous example, and the typeid function (cppreference: typeid operator), we can extract type information, and print it during run-time. The following example furthermore illustrates how the defaulted template parameters can be omitted.

```cpp
#include <iostream>

template <typename T=long, typename S=int>
class A {
public:
    A() { std::cout << typeid(T).name() << " " << typeid(S).name() << "\n"; }
private:
    T v1;
    S v2;
};

int main() {
    A<int,long> a; // <int,long>
    A<double> b; // <double,int>
    A<> c; // <long,int>
}
```

In order that this works, it is similar to function default arguments: you must put the default template parameters in sequence from right to left. Those template arguments without default values must all be left of the defaulted arguments, and must be specified when instantiating the template types.

An example for a default non-type template argument.

```cpp
#include <iostream>

template <typename T=double, unsigned int N=2>
T power(T x) {
    T ret = x;
    for(unsigned int i=1; i<N; ++i) ret *= x;
    return ret;
}

int main() {
    std::cout << power<double,2>(3.) << "\n";
    std::cout << power<double>(3.) << "\n";
    std::cout << power<> (3.) << "\n";
    std::cout << power (3.) << "\n";
}
```

All four lines inside main() do the same thing!
Chapter 26

Generic and Template Meta-Programming

26.1 Generic Programming

Generic programming largely aims to code reuse, i.e. to write code that can be used in a maximum scope of applications. The STL is a very successful example, where e.g. templatized containers, iterators, and algorithms play so seamlessly together that it appears as if this must have been already always like this. Next to that we already showed how the containers can be used with any arbitrary user-defined type, too – types, the programmer of a generic function hardly can foresee. That’s sort of difficulty when programming generically.

C++ Core Guidelines: Rules for Templates and Generic Programming gives some examples and guidelines for generic programming. We just want to give two simple examples as illustration.

26.1.1 Simplification of Algorithm Functions

Often, you probably wonder why STL algorithms are so complicated, and that you have to specify iterators – making the code unnecessarily appear long and incomprehensible. You can make it shorter. For instance, sorting can be done directly – here for a vector.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

// Template for sort function

#include <algorithm>
#include <iostream>
#include <vector>

template<typename T>
void sort(std::vector<T>& v) {
    std::sort(std::begin(v), std::end(v));
}

int main() {
    std::vector<double> v {3.2, 5.4, 1.3, 9.5, 3.7, 5.7};
    sort(v);
    for(auto p : v)
        std::cout << p << " ";
    std::cout << "\n";
}
```

The only thing here that needs to be guaranteed is that any type `T` is less-comparable, i.e. via `operator<`. This also holds already for `std::sort`. Of course, you may want to use also other containers to sort. But some are already sorted - like `set`. To sort those is probably a waste of time. So, generalization is not always reasonable.

We also omitted the comparison function, and used the default. This results in an ordered sequence of values, starting with the smallest value. You may want to change this – so, kind of reverse sort.\(^1\)

\(^1\)A comparison function could be added, and defaulted with `std::less<T>()`. In that way, more flexibility can be added to your `sort` function, and no `reverse_sort` function extra is needed.
Exercise: Use std::copy, and wrap it in order to copy one container into another one by just calling copy(from, to)! Take care for const-correctness!

!! Resist the temptation to put your functions into the std namespace!

26.1.2 Example: max Function for Containers

When we started the template part above, we started with a simple global function. Step-by-step, we added template parameters, to use the function in a more and more general context. Now, we would like to write a max function that finds the maximum of a list or vector of values. Fortunately, the STL has already provides such a function (see cppreference: std::max). But std::max_element is not especially handy. So, let us try this.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

template<typename T>
void reverse_sort(std::vector<T>& v) {
    std::sort(std::begin(v), std::end(v), std::greater<T>());
}

Exercise: Use std::copy, and wrap it in order to copy one container into another one by just calling copy(from, to)! Take care for const-correctness!

!! Resist the temptation to put your functions into the std namespace!

26.1.2 Example: max Function for Containers

When we started the template part above, we started with a simple global function. Step-by-step, we added template parameters, to use the function in a more and more general context. Now, we would like to write a max function that finds the maximum of a list or vector of values. Fortunately, the STL has already provides such a function (see cppreference: std::max). But std::max_element is not especially handy. So, let us try this.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

template<typename T, typename CONT=std::vector<T>>
T max(CONT const& c) {
    return *std::max_element(std::begin(c), std::end(c));
}

int main() {
    std::vector<double> v {3.2, 5.4, 1.3, 9.5, 3.7, 5.7};
    std::cout << max(v) << "\n";    // <-- Error
}
```

Per default, max takes a std::vector<T>, where T must be less-comparable. Unfortunately, we cannot simply write max(v), because the compiler cannot deduce the type T from it. So we could write the following.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

template<typename CONT>
typename CONT::value_type max(CONT const& c) {
    return *std::max_element(std::begin(c), std::end(c));
}

int main() {
    std::vector<double> v {3.2, 5.4, 1.3, 9.5, 3.7, 5.7};
    std::cout << max(v) << "\n";    // Now it works!
}
```

But this is really ugly! There must be a better way. And there is! The following works.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

template<typename CONT>
typename CONT::value_type max(CONT const& c) {
    return *std::max_element(std::begin(c), std::end(c));
}

int main() {
    std::vector<double> v {3.2, 5.4, 1.3, 9.5, 3.7, 5.7};
    std::cout << max(v) << "\n";    // Now it works!
}
```

The only template parameter is now CONT – the container. And this can be deduced. This works, because all STL containers are templates, and have a public member type value_type, which is equal to the container’s template argument T. For std::vector, we can imagine it similar like the following.

```cpp
template<typename T>
class vector {
    public:
        using value_type T;
    ...
};
```
26.2. Template Meta-Programming

Template meta-programming appeared first as a kind of obscurity, because templates where introduced for a completely different purpose – namely generic programming. But it revealed soon after their introduction in C++ that the template mechanism represents a fully qualified (Turing-complete) programming language. Essentially, together with the specialization mechanism, templates allow a programming paradigm called functional programming (Wiki: Functional programming), which is essentially based on recursion. It acts on types!

Naturally, as the template mechanism was not really designed for this kind of programming, it is hard to write and read template meta programs. But it can be exploited to perform interesting effects and optimizations. As templates are evaluated during compile-time, one can use it for compile-time execution to produce C++ code that is then compiled subsequently. We will show the programming principle on the example of Fibonacci numbers. But much more is conceivable. Intel's SDLT, for instance, can be used to write programs in the style of array-of-structure (like e.g. arrays of points), which are easier to handle by humans. But internally, the library converts this into a structure-of-array, which is better vectorizable on modern processor architectures. Another application might be the addition of matrices. With operator overloading, you can easily write something like A=B+C+D+E; – all being matrices. Naively, the compiler would resolve this to A=(B+(C+(D+E))); or A=((B+C)+D)+E;.

All expressions in parentheses are temporaries, where each summation of two matrices consists of nested loops over columns and rows. Looking at efficiency, the component-wise summation of all matrix entries and the passing through a nested loops over columns and rows should be done only once. With templates, we can still write A=B+C+D+E;, but internally during compile-time, there is then only one passing through the loops (see e.g. Eigen: Lazy Evaluation and Expression Templates).

26.2.1 Example: Factorial Numbers in Template Meta Language

After imperative, procedural and object oriented programming, functional programming is another possible programming paradigm (see also Wiki: Comparison of programming paradigms) that is supported by C++ (in several flavors). Template meta-programming is essentially functional programming (see e.g. Wiki: Functional programming), which goes back to the so-called lambda calculus (see e.g. Wiki: Lambda calculus).

This kind of mathematical thinking is mostly unfamiliar to most of us. And so is it surely beyond the scope of this course to introduce this kind of programming without larger preparations. (Those who are interested, lisp and haskell, among others, are designed functional programming languages.) We here only illustrate the principle of template meta programming, and refer the interested reader to the literature.

The simplest case to illustrate the working of template meta programming – so-to-say the "Hello, World!" example in that style – is the factorial function. Using a general template, which employs a non-type argument N for referencing a template type for N-1 introduces a recursion. To stop the recursion (and avoid infinite loops), we have to specialize at N=0.

```cpp
#include <iostream>
template <unsigned int N>
struct Factorial {
  enum { value = N*Factorial<N-1>::value; 
  }
};
template <> struct Factorial<0> {
  enum { value = 1 }
};
```
The values for `Factorial<5>::value` and `Factorial<10>::value` are thus calculated by the compiler (during the compilation).

**Exercise:** The Fibonacci sequence of numbers (see also Wiki: Fibonacci number) can be generated by recursion,

\[ x_n = x_{n-1} + x_{n-2}, \quad x_0 = 0, \quad x_1 = 1, \]

resulting in the sequence 0, 1, 2, 3, 5, 8, 13, 21, \ldots. Realize this as template meta program! Please compare also the speed of the template meta program with a normal recursive function call in C++! What do you observe?

### 26.2.2 Short Remark on Compile-time Execution/Computation

The example above is essentially just a compile-time computation, which is executed for performance reasons (possibly). In such cases, since C++ 11, there is the `constexpr` specifier (see [cppreference: constexpr specifier]) for such purposes. Its capabilities are far extended, such that normal functions can be declared `constexpr`, and can be used (under certain restrictions and conditions) to execute compile-time computations. Those are by far easier to comprehend than template programs. But it might be difficult nonetheless to accomplish compile-time computation, i.e. to resolve all dependencies during compile-time in order to get a definite result.

**Exercise:** Try to implement a compile-time Fibonacci number calculator using `constexpr`! Look into the examples of [cppreference: constexpr specifier] if you need ideas.

**As a last remark on Optimization:** What can be computed during compile-time (not exceeding the run-time efforts), does not need to be computed during run-time. Insofar: Speed is not magic!

### 26.3 Application: Variables with Physical Units

C++ 11 offers also a way to write consistently C++ code with variables, which are physical quantities, and which have a distinct physical unit (something like `meter`, `second`, etc.). Even better, the code can be written in such a way that you can hardly spoil computation of physical quantities with units – some of the nightmares pupils have in school. So is a `length` divided by `time` a `speed`, and not a `force`. So, we would like to write something like this.

```cpp
Mass m = 10.0_kg; // mass: 10 kilograms
Length l = 3_m; // length: 3 meters
Time t = 90_s; // time: 90 seconds
Speed v = 1/t; // speed: in meters/second
Force F = m*l/(t*t); // force: in Newton
Energy E = F*t; // Compile Error: Newton*Second is not Joule!
```

So, the compiler shall check for us the correctness of physical units.

We are talking here about **User-defined literals**, and combine them with templates. Cppreference gives the following illustrative example.

```cpp
#include <iostream>

// used as conversion
constexpr long double operator"" _deg ( long double deg ) {
    return deg * 3.14159265358979323846264L / 180;
}
```
26.3. APPLICATION: VARIABLES WITH PHYSICAL UNITS

```cpp
// used with custom type
class mytype {
    unsigned long long m;
};
constexpr mytype operator""_mytype (unsigned long long n) {
    return mytype(n);
}
// used for side-effects
void operator""_print (const char* str) {
    std::cout << str;
}
int main(){
    double x = 90.0_deg;
    std::cout << std::fixed << x << std::endl;
    mytype y = 123_mytype;
    std::cout << y.m << std::endl;
    0x123ABC_print;
}
```

Essentially, this is just overloading the operator "" (which is not really an operator(!), but used only for literals!), followed by an underscore, _, and a rather arbitrary name.\(^2\)

**Some remarks:**
1) `long double` must be used for floating point types, as well as `long long` for integer types. Otherwise, the compiler refuses the compilation.
2) For floating point literals, the decimal point is mandatory! Here, for instance, you could not write `90_deg`. The compiler does not do the integer to floating point conversion here!

The next ingredient is our physical type system. We follow largely here **Compile-time numerical unit dimension checking in C++ 11**. So, first we fix the physical basic units to be that of mass – *kilogram, kg*, of length – *meter, m*, and of time – *second, s*. All other units like *meters per second, m/s* = \(m^1s^{-1}\) or *Newton, N* = \(kgm^1s^{-2}\) can be formed from those. So, we introduce a class template `Quantity` with template parameters representing the exponents on the respective basic units.

```cpp
template<int MassDim, int LengthDim, int TimeDim>
struct Quantity {
    Quantity(long double val) : value(val) {} // constructor
    long double value; // the actual value
};
```

Next, in order to do computations with units, we need to overload the respective operators.

```cpp
// Addition operator
template <int M, int L, int T>
Quantity<M, L, T> operator+(const Quantity<M, L, T>& lhs,
                          const Quantity<M, L, T>& rhs)
{
    return Quantity<M, L, T>({lhs.value + rhs.value});
}
// Subtraction operator
template <int M, int L, int T>
Quantity<M, L, T> operator-(const Quantity<M, L, T>& lhs,
                          const Quantity<M, L, T>& rhs)
{
    return Quantity<M, L, T>({lhs.value - rhs.value});
}
// Multiplication operator
template <int M1, int L1, int T1, int M2, int L2, int T2>
Quantity<M1+M2, L1+L2, T1+T2> operator*(const Quantity<M1, L1, T1>& lhs,
                                      const Quantity<M2, L2, T2>& rhs)
{
    return Quantity<M1+M2, L1+L2, T1+T2>({lhs.value * rhs.value});
}
```

\(^2\)Non-underscore overloading are reserved for future standardization, as is the case for `std::chrono::duration` with e.g. operator"" min since C++ 14. But you can use non-underscore overloading, too, as long as no clashes are to fear.
Here, we exploit that the non-type arguments are integer values, which can be added or subtracted, etc. From this, the new “type” is deduced (meaning the correct exponents). In parallel, meaning within the function body, the quantity’s value is calculated from the operand’s values, and returned. Both things must happen to stay consistent.

A very last thing to do is to define some aliases to ease the reading.

```cpp
using Mass = Quantity<1,0,0>;
using Length = Quantity<0,1,0>;
using Time = Quantity<0,0,1>;
using Speed = Quantity<1,1,-1>;
using Force = Quantity<1,1,-2>;
using Energy = Quantity<1,2,-2>;
```

(You can also use typedefs. But using alias declarations are more modern (see cppreference: C++ keywords: using).) That’s it! Our program above should now work. For illustration, we add still some output (overloading of <<) and the literals with units from above, and get the following.

```cpp
Mass operator""\_kg (long double mass) {
    return Mass{mass};
}
Length operator""\_m (long double length) {
    return Length{length};
}
Time operator""\_s (long double time) {
    return Time{time};
}
std::ostream& operator<<(std::ostream& os, const Speed& v) {
    os << v.value << " m/s";
    return os;
}
std::ostream& operator<<(std::ostream& os, const Force& F) {
    os << F.value << " N";
    return os;
}
```

```cpp
int main() {
    Mass m = 10.0\_kg;
    Length l = 3.0\_m;
    Time t = 90.0\_s;
    Speed v = 1/t;
    std::cout << " speed = " << v << "\n";
    Force F = m*l/(t*t);
    std::cout << " Force = " << F << "\n";
    // Energy E = F*t;  // Compile Error: Newton\_Second is not Joule!
}
```

**Exercise:** Correct the formula for energy (energy = force times length) and compile the code! Add output for Energy! Add also literal operators for speed (mps), force (N) and energy (J), such that numbers with units can be assigned also to them!

Can you conceive any way for printing out the quantities with units for different orders of magnitude (e.g. km for kilometers, instead of meters)?

Add also a square root and a power function, which is capable of handling the correct units (as long as powers are divisible by two for the square root)!
Chapter 27

A Short Glimpse on Design Patterns and Idioms

Not only driven by templates and inheritance, but largely supported by them, it has prevailed to introduce so-called design patterns. In most cases, that is a specific class or class relation design fulfilling a certain dedicated task, which repeatedly occurs in different software design issues. And which mostly are adapted to the good software design principles, which we already mentioned from time to time in this course (remember SOLID!). Some of the design patterns were already mentioned, even if not declared as such, in the chapters above.

Again, a thorough introduction or overview of design patterns would largely exceed the scope of a basic C++ course. But there is a vast literature on that, starting with e.g. [9, 21]. And also the internet provides much information, e.g. Wikibooks: C++ Programming: Code patterns design.

We just introduce here some of the design patterns, which have important representatives in the STL, or are just illustrative.

27.1 Smart Pointers

This is usually not a dedicated design pattern. But it possesses many of design pattern properties in terms of being generic, and representing a class of objects with similar semantics (and thus behavior).

As special example do we have a look on unique pointer, which has the semantic that it maintains a resource (a pointer to an arbitrary type object), and cares for its destruction/release when going out of scope. The STL has realized it as std::unique_ptr (see cppreference: std::unique_ptr). "Unique" means here that the resource is owned exclusively by one object of type unique pointer. It may have different operators overloaded to access the resource in a natural way (like a normal pointer).

```cpp
#include <iostream>
#include <string>
struct Dummy {
    Dummy () { std::cout << "A Dummy was created!\n"; }
    void print_text(const std::string& s) const { std::cout << s << "\n";});

template <typename T>
class unique_pointer {
public:
    unique_pointer(T* pValue) : pD(pValue) {}
    unique_pointer(const unique_pointer&) = delete; // copy c'tor
    unique_pointer(unique_pointer&) = delete;        // move c'tor
    unique_pointer& operator=(const unique_pointer&) = delete; // copy assign
    unique_pointer& operator=(unique_pointer&) = delete; // move assign
};
```
CHAPTER 27. A SHORT GLIMPSE ON DESIGN PATTERNS AND IDIOMS

```cpp
unique_pointer() {
    delete pD;
} // assumes new created
T& operator*() { return *pD; } // dereference
T* operator->() { return pD; } // access to pointer member
private:
    T* pD; // managed resource
};
int main() {
    unique_pointer<Dummy> p {new Dummy};
    (*p).print_text("dereference pointer");
    p->print_text("access to pointer member");
    // unique_pointer<Dummy> p2 {p}; // error: no copy c'tor
    // unique_pointer<Dummy> p2 = p; // error: no copy assign
}

We removed the copy constructor and assignment operator (and also the move constructor and assignment operator, which are part of advanced C++), such that the unique pointer cannot be copied in any way. This also means that you cannot call a function with a unique pointer parameter by value! The last two lines in main (which are commented out) should not compile.

Furthermore, unique pointer’s only constructor takes a pointer of type T (created by new) and assigns it to its only data member. The destructor releases the resource by calling delete. The effect of this construct is that we can use dynamic memory management, i.e. create free store objects via new. But we don’t need to care for correctly releasing the acquired memory again. This is the task of unique pointer (its destructor). This is so, because we handle the unique pointer itself as a value (see in previous parts for value semantics!). That is, the destructor of unique pointer is automatically called, when the scope is left (when the program leaves the function either normally, or when an exception occurs).

Finally, we overload the operators * (unary; empty parameter list) and ->. In this way, although we have a unique pointer in hand, we can do as if we had the raw pointer to the object in hand, which is but truly contained within the unique pointer.

Exercise: So, unique pointer behaves mostly like a normal pointer, but not yet with respect to operator<<, which should print the address the pointer points to. Correct this!

27.2 Singleton

A singleton is an object that is created in memory only once for the whole program. It cannot be copied in any way, and it cannot be destroyed from outside (usually). Such an idiom can have different applications. A central logging stream, or access points for hardware drivers (printers, etc.) can be practically realized as a singleton.

```cpp
#include <iostream>
class Singleton {
public:
    static Singleton* getInstance() {
        if(!instance)
            instance = new Singleton();
        return instance;
    }
    void printInfo() {
        std::cout << "I'm instance " << instance << "\n";
    }
private:
    Singleton() {};
    Singleton(const Singleton&) {};
    Singleton operator=(const Singleton&) { return *this; };
    "Singleton() {};
```

1 That's different in std::unique_ptr!
2 To write thread-safe singletons is but rather a different story, and usually more difficult to accomplish.
We have to make all constructors private such that they cannot be called from outside. An instance of the singleton can only be obtained via the static getInstance member function. And this function returns only instance, unless it is the null pointer, in which case the singleton is instantiated, and than instance is returned.

Of course, singletons can have other class member functions or data members. Whatever you need. You can make a singleton also as base class, from which other classes can be derived in order to become singletons by themselves (can be tricky).

A non-pointer, value-semantics solution (claimed also to be thread-safe) is given by Scott Meyers.

```cpp
#include <iostream>

class Singleton {
public:
    static Singleton& getInstance () {
        static Singleton instance;
        return instance;
    }
    void printInfo() {
        std::cout << "I'm the only one with ID " << ID << " \n" ;
    }
    void setId(int sID) {
        ID = sID;
    }
private:
    Singleton() = default;
    Singleton(const Singleton &)= delete;
    Singleton operator=(const Singleton &)= delete;
    "Singleton() = default;
    int ID {};
    
    void func() {
        Singleton& s = Singleton::getInstance();
        s.printInfo () ;
    }

    int main() {
        Singleton& s = Singleton::getInstance();
        s.setId(4);
        func();
    }
```

The static local variable instance guarantees that there is only one such singleton created (in essence, it is a global variable). So, it exists also after leaving the function. And so returning the reference is safe.

### 27.3 Factories

The business of the Factory design pattern is to create objects via a function, where you actually don’t know anything about the corresponding class or its constructors, i.e. how this object is to be created.

Classically, you have a simple shallow class hierarchy with several classes derived from a single base
class. Additionally, you add a factory class with a public create member function, which takes some parameter that identifies somehow the derived types. This function then creates an object and returns a handle to it. The user of this factory only needs to tell the factory "Give me an object of type X!".

Let’s take as example a role game with characters such as orcs, elves, dwarfs, men, etc. All characters have in common that they can move, or attack another character. So, these are the candidates for member functions of the base class Character. The inheritance hierarchy allows to specify individually how each character type moves or attacks. So, these member functions could be made virtual, and overridden. Or, you can make them also generic – move is move, no matter how. So, we can put the position of a character into the base class, and move determines only that the character moved from location A to location B – no matter how, or how fast. These details are left open, in order not to lose the focus from the Factory concept. So, our class hierarchy is as follows.

```cpp
#include <iostream>
#include <string>
#include <vector>

class Character {
public:
    virtual void speak(const std::string&) const = 0;
    virtual ~Character() {};
};
class OrcCharacter : public Character {
    void speak(const std::string& s) const override final {
        std::cout << "Orc: " << s << std::endl;
    }
};
class ElfCharacter : public Character {
    void speak(const std::string& s) const override final {
        std::cout << "Elf: " << s << std::endl;
    }
};
class ManCharacter : public Character {
    void speak(const std::string& s) const override final {
        std::cout << "Man: " << s << std::endl;
    }
};

class CharacterFactory {
public:
    enum CharacterType {Orc, Elf, Man};
    static Character* create(CharacterType type) {
        switch(type) {
            case Orc: return new OrcCharacter{};
            case Elf: return new ElfCharacter{};
            case Man: return new ManCharacter{};
        }
        return nullptr;
    }
};
```

In a next step, we introduce the factory.

```cpp
class CharacterFactory {
public:
    enum CharacterType {Orc, Elf, Man};
    static Character* create(CharacterType type) {
        switch(type) {
            case Orc: return new OrcCharacter{};
            case Elf: return new ElfCharacter{};
            case Man: return new ManCharacter{};
        }
        return nullptr;
    }
};
```

Finally, we use the factory (we are careless here not to check for nullptr).

```cpp
int main() {
    std::vector<Character*> players;
    players.push_back(CharacterFactory::create(CharacterFactory::Orc));
    players.push_back(CharacterFactory::create(CharacterFactory::Elf));
    players.push_back(CharacterFactory::create(CharacterFactory::Man));
    for(auto p : players) p->speak("I'm ready!");
    for(auto p : players) delete p;
}
```

The idea of a factory becomes clear, hopefully. The initialization is left completely to the factory. More properties can be added for the initialization of each character inside create. And more characters can be added easily.
Exercise: Add dwarf character! Extend the character by some property such as power or lifetime, and initialize this value accordingly inside create (could be also a parameter of create forwarded to the constructors).

A quite interesting generic factory can be created by means of templates, so-called variadic template parameters, and some more STL features.

```cpp
#include <string>
#include <utility>
#include <vector>
using std::initializer_list;

struct MyData {
    MyData (int,double,char) {};
};
template<typename T,typename ... Args>
T createT(Args&& ... args) {
    return T (std::forward<Args>(args)...);
}

int main() {
    int a = createT<int>();
    int b = createT<int>(1);
    std::string s = createT<std::string>("This is a String!");
    MyData myData = createT<MyData>(1,3.1342, 'a');
    std::vector<int> intVec = createT<std::vector<int>>({1,2,3});
}
```

If you don’t understand fully all the tiny details (e.g. rvalue (&&), std::forward), don’t worry! This is indeed already quite advanced C++. The point of interest for you should be that we create objects by means of a single function createT(). And these objects are even reference objects (no pointers as before)! For what you can use it is of course a different story, and your creativity is put into charge!

27.4 PIMPL

Also PIMPL (actually “pointer to implementation”) is not a design pattern, but an idiom, which uses other design patterns — in this case, unique pointer might come to one’s mind. The idea behind this concept is to decouple the implementation from the visible interface. So, imagine you would like to write a library, which has to provide some headers containing the class declarations. Inside this declaration also the private member functions are visible, even if the library user can’t use it. Every time you would like to change something in this interface, every user of your library would need to recompile his/her code. PIMPL helps to avoid this.

Also for PIMPL there are different possibilities to realize it. We show here just one way for illustration. To get a clearer idea, we split this case also into different files. So, imagine you had a class foo (it really doesn’t matter what it represents — could be a widget class, or something arbitrary). Let us start with the header.

```cpp
// foo.h
#ifndef FOO_
#define FOO_

class foo_impl;

class foo {
public:
    foo(int);
    ~foo();
    void print();
private:
    foo_impl* pimpl;
};
#endif
```

It has a public interface, which will be used by users of your library, such as in the following main program.
What now is remarkable is the fact that the complete private interface is hidden in pimpl. In ideal cases, a API user of your library will never know what is behind that. But he/she doesn't need to anyway! For him/her important is only the public interface that is used.

Even better, for the header file, it is completely unknown what foo_impl actually is! The class foo_impl in line 4 is just a forward declaration – say a type announcement. What foo_impl contains, its interface, is not necessary to be known here, because pimpl is just a pointer to foo_impl. The compiler does not need to know, what the implementation details are. (That's an advantage of pointers that references don't have!) So, also no header needs to be included here that would tell the details of foo_impl's interface. It thus really doesn't matter if foo_impl is changed or not, and main.cxx doesn't need to be recompiled if foo_impl changes.

An implementation might now look as follows. Only important is that consistency between foo and foo_impl is present.

```cpp
#include "foo.h"
#include <iostream>

class foo_impl {
    private:
        int value;
    public:
        foo_impl(int val) : value(val) {}
        void print() {
            std::cout << "I'm PIMPLE with value " << value << "\n";
        }
};
// foo's interface implementation
foo::foo(int val) : pimpl(new foo_impl(val)) {}
foo::~foo() { delete pimpl; }
void foo::print() { pimpl->print(); }
```

This idiom should be used if the coupling between a user interface and the implementation behind the scenes should be minimized. But as always! Don't overuse it! It also increases the code complexity.

**Exercise:** Add functions to foo_impl, which get and set the value. Also change foo::print() to increase the value by one, and than print it. What do you observe when you recompile?

### 27.5 Curiously Recurring Template Pattern (CRTP)

A last pattern shall be presented here for illustration what the combination of templates and inheritance can create. CRTP can be used in C++ to realize mixins, which change the behavior of a class by inheritance, but still as static polymorphism (no overhead by virtual mechanism). The basic layout of CRTP is the following (taken from *The Curiously Recurring Template Pattern (CRTP)*).

```cpp
template <typename T>
class Base {
    [...] 
};
class Derived : public Base<Derived> {
    [...] 
};
```

The base class is a class template, and the derived classes give their own type as template parameter to the base class. The point now is that the base class can obtain direct access to the derived class.
struct Base {
    void doSomething() {
        T& d = static_cast<T&>(*this);
        d.printHello();
    }
    virtual void printHello() = 0;
};
struct Derived : public Base<Derived> {
    void printHello() { std::cout << "Hello!\n"; }
};
struct DerivedDe : public Base<Derived> {
    void printHello() { std::cout << "Hallo De!\n"; }
};
template <typename T>
void doSomething(Base<T>& b) {
    b.doSomething();
}
int main() {
    Derived d;
    doSomething(d);
    DerivedDe de;
    doSomething(de);
}

This is not really a IS-A inheritance, and implements static (compile-time) polymorphism. A more useful example can be found here. Can you conceive any other applications, e.g. in computational physics or so? In that respect, also consider once again Sean Parent’s Talk "Inheritance Is The Base Class of Evil".
CHAPTER 27. A SHORT GLIMPSE ON DESIGN PATTERNS AND IDIOMS
Chapter 28

C++ – Advanced Topics

How to go on? Well, take your time to get familiar with the concepts learned so far! Try to apply them where you think they fit. With the time it will feel much more natural and easy to use them.

Of course, there are many more aspects in C++. And C++ is still further evolving. The following list of topics is surely not comprehensive. But it could become more interesting once you become more expert in C++.

- Move Semantics (std::move, std::forward, move constructor and move assignment), Return Value Optimization, Copy Elision
- std::variant, std::visit, std::optional as realizations of design patterns like the visitor design pattern
- Type Erasure (std::any)
- Advanced features of the C++ Standard Library and Standard Template Library [16, 27]
- C++ Design Principles, Patterns, and Guidelines (RAII, SOLID, SFINAE, etc. and [9, 21, 3])
- Generic and Template Meta Programming, [43]
- Parallel and Concurrent Programming (C++ Core Guidelines: Rules for Concurrency and Parallelism, C++ Concurrency in Action)
- C++ Concepts and Type Traits (Wiki: Concepts (C++), cppreference: Concepts library, Type support, cppreference: Standard library header type_traits)
Chapter 29

Hands-On Exercises for Part 5

29.1 Visualization – Simple Ray Tracing

Having the XYimage class, we can also try to render a (more or less) ray-tracing image. Ray-tracing works as, for instance, explained in raytracing algorithm in a nutshell.

**Task:** Use the means you have to realize a simple ray-tracing – meaning, objects are spheres. Model the rays with reasonable objects.

**Hint:** You can reuse the Point, Color and XYimage classes from the former parts for a realization of the final image. In Figure 29.2, you can see the different steps of the program evolution. First, we setup only the shapes (spheres – with position and color), a scene (container for the shapes and other stuff), and a camera. In the subsequent step we added a light source, shadows, head lights, and reflection (finally even individually). The first task, to geometrically navigate via rays through the scenery, can be accomplished recursively. Light (color) models are a bit more difficult to accomplish – mostly they are not physical. But with some intuition, one obtains quite realistic effects.

Figure 29.1 tries to outline the process of ray tracing. A ray is send from the camera (from the camera focus through an image pixel) into the scene. The first step (top left of Figure 29.2) is to geometrically track the ray (until it hits something or nothing). In the next step, at the hit point of an object, the ray picks up a color.

Next, we add light. At a hit point, light (color) comes from direct illumination from the light source. The RGB (red-green-blue) color of the object is scaled with the cosine of the angle between

Figure 29.1: Ray-tracing process scheme.
the normal of the object in that point and the (negative) direction of the incoming light. This gives
the illusion of depth already (top right of Figure 29.2). Here it comes already into the game that the
object is shadowed by other objects. So, if a hit point is located in a shadow, it is made black.

In a next step we add head lights (middle left of Figure 29.2). This is just adding white if the
reflected ray is not much deviating from the (negative) light direction. That is again a cosine of the
an angle – this time between reflected and light direction. The white color is scaled with a Gaussian
with some variance, around “1-cosine(angle)”.

Finally, we only need to add reflectivity, where the reflected ray is now traced to the next object
(recursively). The resulting color from that ray is convex-combined (weighted by the reflectivity \( \in [0, 1] \))
with the local object color. This is shown in middle right of Figure 29.2. The last picture just extended
to that each object has a different reflectivity.
Figure 29.2: Development of RayTracer.
29.2 Algebra

29.2.1 Templated Matrices and Vectors

Task: Realize the matrix and vector class as matrix template class, including all kinds of operations!

Hint: Is one matrix class for all objects sufficient, here? Which problem occurs when you assume non-square matrices? Which operations are then allowed? Which problems cannot be solved in that way?

29.2.2 Eigen Library (Tutorial)

One can use Eigen for e.g. solving partial differential equations (PDEs) – see the previous part hands-on exercises: the solution to the PDE solver example, which resulted in solving a linear equation for the implicit scheme. We will outline here a rather general approach to use Eigen for such problems. For details, we refer to literature and the Eigen documentation.

With respect to the PDE solver example in the previous part’s hands-on session, we had an implicit scheme with a system matrix \( A \), which is a \( N^2 \times N^2 \) matrix, with 13 entries per row (column), and which is symmetric, as we could convince ourselves. We use the multi-index \( k = i + N \cdot j \) for the \( A \) matrix indices.

\[
\begin{align*}
A_{k,k} &= 1 - \Delta t \left( \varepsilon - 1 + 8/(\Delta x)^2 - 20/(\Delta x)^4 \right) \\
A_{k,k\pm1} &= A_{k,k\pmN} = \Delta t \left( 2/(\Delta x)^2 - 8/(\Delta x)^4 \right) \\
A_{k,k\pm1\pmN} &= 2\Delta t/(\Delta x)^4 \\
A_{k,k\pm2} &= A_{k,k\pm2N} = \Delta t/(\Delta x)^4
\end{align*}
\]

With the Eigen library and the MyArray, we can write a rather compact program for the simulation. First of all, Eigen provides matrix and vector classes, which are based on an array class. This includes also operator overloading for matrix-matrix, matrix-vector, scalar-matrix, scalar-vector etc. operation. Furthermore, the array class has overloaded component-wise operations overloaded, for convenience. There are several matrix classes, among others also for sparse matrices. Next, Eigen provides solvers for linear equations, which are represented by a matrix, \( A \), and a right-hand side vector, \( b \). Among those, there are also solvers optimized for sparse systems. These are mostly iterative solvers, which are usually rather slow, but also much more flexible. Eigen also provides preconditioners to accelerate the solution. The usage of such methods is still kind of a black magic. But the become almost the only viable ways to solve very complex problems (complex geometries and boundary conditions, large systems of equations (see e.g. CFD with chemical reactions)). So, for our problem at hand – the single-scalar Swift-Hohenberg equation on a square with an equidistant Cartesian grid, and periodic boundary conditions, an iterative method is sure a waste of time in comparison to the spectral method. But for illustration, it is a good starting point.

Here is the code snippet (MyArray is found in the appendix of the previous hands-on session). A short description follows.

```cpp
#include "MyArray.h"
#include <iostream>
#include <Eigen/Core>
#include <Eigen/IterativeLinearSolvers>

int index(int i, int j, int N) {
    if(i<0) i+=N;
    if(j<0) j+=N;
    if(i>=N) i-=N;
    if(j>=N) j-=N;
    return j+N*i;
}
```
int main() {
    int NN = 256;
    double dt = 0.05;
    double dx = 0.5;
    double eps = 0.1;
    double A = 0.5;

    int n = NN*NN;
    double dx2 = dx*dx;
    double dx4 = dx2*dx2;
    double aa=1.-dt*(eps-1.+8./dx2-20./dx4);
    double bb=dt*(2./dx2-8./dx4);
    double cc=2.*dt/dx4;
    double dd=dt/dx4;

    using T = Eigen::Triplet<double>;
    std::vector<T> tripletList;
    tripletList.reserve(13*n);

    for(int i,j,l=0; l<n; ++l) {
        i = l/NN; // row index
        j = l%NN; // col index
        tripletList.push_back(T(l,l,aa));
        tripletList.push_back(T(l,index(i+1,j,NN),bb));
        tripletList.push_back(T(l,index(i-1,j,NN),bb));
        tripletList.push_back(T(l,index(i,j+1,NN),bb));
        tripletList.push_back(T(l,index(i,j-1,NN),bb));
        tripletList.push_back(T(l,index(i+1,j+1,NN),cc));
        tripletList.push_back(T(l,index(i+1,j-1,NN),cc));
        tripletList.push_back(T(l,index(i-1,j+1,NN),cc));
        tripletList.push_back(T(l,index(i-1,j-1,NN),cc));
        tripletList.push_back(T(l,index(i+2,j,NN),dd));
        tripletList.push_back(T(l,index(i-2,j,NN),dd));
        tripletList.push_back(T(l,index(i,j+2,NN),dd));
        tripletList.push_back(T(l,index(i,j-2,NN),dd));
    }
    std::cout << "Ready Filled Matrix\n";

    Eigen::SparseMatrix<double> S(n,n);
    S.setFromTriplets(tripletList.begin(), tripletList.end());
    S.makeCompressed();

    Eigen::VectorXd b(n), x(n);
    x.setRandom();
    x = 1.e-4*(x.array()-x.mean()).matrix();

    MyArray psi{{NN,NN,1},{dx,dx,dx},{0,0,0},"psi"};
    psi.va.reserve(n);
    for(int k=0; k<n; ++k) psi.va[k] = x[k];
    psi.writeToVTK(std::string("test_0.vtk"));

    Eigen::ConjugateGradient<Eigen::SparseMatrix<double>,
        Eigen::Upper,
        Eigen::IncompleteLUT<double>> solver;
    // Eigen::BiCGSTAB<Eigen::SparseMatrix<double>> solver;
    // Eigen::GMRES<Eigen::SparseMatrix<double>> solver;
    // Eigen::DGMRES<Eigen::SparseMatrix<double>> solver;
    //Eigen::MINRES<Eigen::SparseMatrix<double>> solver;
    solver.analyzePattern(S);
    solver.compute(S);
    if(solver.info()! = Eigen::Success) {
        std::cerr << "Compute failed\n";
        return -1;
    }

    for(int i=0; i<50; ++i) {
        for(int j=0; j<100; ++j) {
            for(int k=0; k<n; ++k) {

            }
        }
    }
}

}
b[k] = x[k] + dt*(A-x[k])*x[k]*x[k];
std::cout << j << "\r" << std::flush;
x = solver.solve(b);
if(solver.info()!=Eigen::Success) {
    std::cerr << "Solution failed\n";
    return -1;
}
}
std::cout << "#iterations: " << solver.iterations()
<< " estimated error: " << solver.error() << 'n';
std::cout << " step " << i << 'n';
for(int k=0; k<n; ++k) psi.va[k] = x[k];
psi.writeToVTK(std::string("test_"+std::to_string(i+1)+".vtk"));
}
}

The function index is just a helper to realize the boundary conditions (BCs). You will surely do it differently for other BCs. The first part of main is devoted to the setup of the parameters for the simulation. aa till dd are the values for the matrix entries. In more complex simulations, it won’t be that easy anymore. In the following loop, we set the matrix entries. The Eigen::Triplet class is made to take the matrix indices, and the corresponding value of the matrix entry. To get this right requires some exercise – but is also really no rocket science. The Triple vector is used to create the SparseMatrix S.

Next, we need some initial condition for the field vector x. We use a random value field (with values in \([-1, 1]\)) and shift and scale it to get mean zero and a maximum amplitude 10^{-4} – this is for comparability with our former spectral method.

We also create a MyArray to easily generate output, which we can visualize using Paraview.

Now starts the fun (or pain, depending on the programmer’s skills). We generate a solver – here a ConjugateGradient solver. Others are also available. We also use a preconditioner, given to the solver via template parameters. We chose an incomplete LU decomposition. Again, also others are available. One advantage now is that you can test different solvers and preconditioners just by changing a line of code. solver.compute(S) sets the solver onto our system represented by S.

The remaining loops are just the outer time-stepping loop, and an inner precision loop. The latter is actually not strictly necessary for implicit schemes. But it increases the precision of the solution, because the right-hand side of the equation contains the non-linearity. For iterative methods, this is a parameter which can also serve numerical stability, because with a smaller time step, the chance is possibly better for a diagonal dominant system matrix, as we have seen. Meant is here the convergence of the iterative solution step of e.g. CG. (In that respect, the spectral method is much more stable than the iterative method!)

**Task:** Play around with the solvers and preconditioners! Also, play with the time step and the other parameter (possibly better not all at the same time), and observe the effect of it!

You can also consider to change the boundary conditions. How would you, for instance, implement a fixed value of \( \psi \) at the boundary? Or, what about derivatives of \( \psi \) at the boundary?\(^1\)

### 29.3 Template Meta-Programming and Type Traits (Introduction)

Read through the two short blogs about Template Meta Programming and Type Traits! This should give you a basic imagination about what else can be done with templates.

The STL largely provides already some of the traits for different purposes (cppreference: type_traits). One example of usage is that you write a e.g. class or a function, which shall be generic in the sense that it could be used with different types. Still, you need some basic assumptions about these types (e.g. that they have a default or a copy constructor, or that some operators are overloaded for them.

\(^1\)See Dirichlet and von Neumann boundary conditions.
Task: Assume you have a vector of integers. Think about an algorithm that can check in \(O(N)\) (complexity) whether two (not the same) elements of the vector add up to a specific value – e.g. 8. Implement this algorithm in the most general way! For which container types is this a reasonable algorithm? What must the type of the elements of the container fulfill such that it works?

29.4 Named Parameters without Boost (Tutorial)

Although STL, boost and others are a very nice and convenient libraries, fundamental understanding of C++ is usually accomplished when you try to re-implement higher-level features for yourself. This starts usually with re-implementing containers and algorithms. We usually recommend beginners to do this, as well. The boost in experience is enormous. And the understanding for the libraries also increases in that course.

Here, we would like to outline in a tutorial fashion how you can accomplish named parameters for functions (also class-members if needed) by the means of object-orientation and templates – so, essentially the basic language features of C++. This essay is essentially based on Fluent C++: Named Arguments in C++.

Usually, you start with a certain function, which has some parameters.

```cpp
void do_something(double d, std::string const & text) {
    std::cout << "text is " << text << " and d = " << d << "\n";
}
```

Standard conform, you have to call it via e.g.

```cpp
do_something(5.31, "bla");
```

What we but want is e.g.

```cpp
do_something(d=5.31, text="bla");
```

This is better readable. And immediately comes the request (again), if that’s working, we would like to have also that the following works.

```cpp
do_something(text="bla", d=5.31);
```

That is the order of arguments should not matter – and indeed, semantically, it doesn’t! Only syntactically.

Okay, let’s not get ahead of ourselves. Let’s start with the names of parameters.

```cpp
#include <iostream>
#include <string>

template<typename T, typename tag>
struct NamedType {
    NamedType(T val) : val{val} {};
    T val;
};

using D = NamedType<double, struct DTag>;
using Text = NamedType<std::string, struct TextTag>;

void do_something(D const & d, Text const & text) {
    std::cout << "text is " << text.val << " and d = " << d.val << "\n";
}

int main() {
    do_something(D(5.21), Text("bla"));
}
```

We created a template class NamedType, which has a (conversion) constructor, and a class data member `val`. This template class has two parameters. The first, `T`, will be the type of our later function argument. The second is just a dummy template argument, such that we can have different named arguments with the same type. For instance, if you need several strings in one function’s argument list.
Via using, we declare new type names – one for each later argument parameter. Therefore, we used upper-case here. The function argument list is now changed to take const references to these types. Inside the function, we just use the NamedType’s class data member val to access the data. That’s not really nice. But the "user" of that function will usually not see it. It’s implementation.

Inside main, we call this function – but now with the respective constructors (and in the correct order).

In the next step, we extend NamedType.

```cpp
#include <iostream>
#include <string>
template<typename T, typename tag>
struct NamedType {
  T val;
  NamedType(T val) : val(val) {}
  struct argument {
    NamedType operator=(T&& value) const {
      return NamedType(std::forward<T>(value));
    }
  }
};
using D = NamedType<double, struct DTag>;
using Text = NamedType<std::string, struct TextTag>;
static const D::argument d;
static const Text::argument text;
void do_something(D d, Text text) {
  std::cout << "text is " << text.val << " and d = " << d.val << "\n";
}

int main() {
  do_something(D(5.21), Text("bla"));
  do_something(d=5.21, text="bla");
}
```

We nest a struct argument type, which has overloaded the assignment operator (operator=), which takes a so-called assignment reference (in oppose to r-value reference) argument of the type of the later function argument. It returns a newly created (constructed) of type NamedType with the correctly specialized template parameters. This operator thus converts from e.g. a double value to a NamedType<double, struct DTag> type. And in order that this works, we create the static const objects d and text. Lower-case now, because there are the names we will use in the function argument list later, in main().

Voila! Named arguments!

To ensure that NamedType::argument is not somehow misused, you can add

```cpp
  argument() = default;
  argument(argument const&) = delete;
  argument(argument&) = delete;
  argument& operator=(argument const&) = delete;
  argument& operator=(argument&) = delete;
```

such that it cannot be copied, moved, assigned, etc.

To make them also position-invariant, as a last step, we have to work still a bit harder. This is now already advanced stuff, and requires that we use C++14! But brave people will nonetheless want to know how it works! We have to add the header <tuple>, and a template function pick. Furthermore, we templatize also do_something.

```cpp
#include <tuple>
[...]
 template<typename TypeToPick, typename... Types>
 TypeToPick pick(Types&&... args) {
   return std::get<TypeToPick>(std::forward<Types>(args)...);
}
```

2Interested reader can have a look on [cppreference: value categories ans cppreference: std::forward. But for a general understanding, it is not urgently needed here!}
Let us start with `do_something`. As we do not know the argument types in advance, we have to parameterize them — `Arg0` and `Arg1`. Inside the function, we define `d` and `text` locally (we can use different names), and access their values as usual. We assign the arguments by type, which is essentially accomplished by `std::get` (see [cppreference: std::get](https://en.cppreference.com/w/cpp/utility/tuple/get)). In short, `std::get` selects from a `std::tuple` — either by position, or — since C++ 14 — by type, if unique.

pick is just a convenience here. We could equally as well write directly

```cpp
template<typename Arg0, typename Arg1>
void do_something(Arg0&& arg0, Arg1&& arg1) {
    auto d = pick<D>(arg0, arg1);
    auto text = pick<Text>(arg0, arg1);
    std::cout << "text is " << text.val << " and d = " << d.val << "\n";
}
```

This would make all parameters copy by value. R-values like in `(Arg0&& arg0, Arg1&& arg1)` avoid this, and references would still stay references. Now, you can use `do_something` in any way you like.

```cpp
int main() {
    do_something(D(5.21), Text("bla"));
    do_something(d=5.21, text="bla");
    do_something(text="bla", d=5.21);
}
```

This is not equivalent to what boost.parameter does (boost.parameter uses preprocessor macros, which can be considered actually not to belong to the C++ language feature scope). And so, wrapping class member functions and constructors, and defining default parameters, is probably more difficult.

We wanted to show you what is possible with the persisting intrinsic C++ language features. And more is to discover for you! Most probably, you will not use these from the outset. But we hope to excite some interest to you, enough to make you curious what else exists.

Try to apply what you have learned so far when it can help you. And try to enjoy it ...
Part VI

Appendix
Appendix A

Setup of Development Environment

A.1 General Considerations

Under ISOC++ Getting Started, you can find a rather comprehensive list of C++ compilers for different platforms. For Windows, for instance, even the GNU Compiler Suite is available — MinGW. It comes with the latest compiler version, boost, git, make, binutils, etc. CMake must (can) be installed separately. It can be be even used for non-admin-privileged accounts. Together with portable IDEs such as CodeBlocks and Eclipse, but also good editors like Notepad++ and Atom, one can easily make a Windows account to a real development environment.

For Linux/Unix systems, the package managers are mostly the most convenient way to go, although pre-compiled binaries are also available, which do not necessitate any admin rights.

Still, the default system package manager often do not supply the latest version of compilers and some libraries. Building them from source is not that difficult, as described in the following section.

A.2 Setup of Development Environment under Linux

The procedures for software installation described in this chapter can be performed on Debian-like systems (including on a Debian WSL (Windows Subsystem for Linux) under Windows 10), which provide the apt package management tool.

A.2.1 Basic System

```
$ sudo apt install vim emacs gnuplot binutils build-essential module-assistant \
g++ imagemagick make cmake cmake-curses-gui git evince ipython3 python3-dev \
python3-pytest python3-numpy python3-scipy python3-matplotlib doxygen time
```

You can install the editor of your choice. (Ask for the root password!)

A.2.2 GCC Compiler

GCC is the GNU C compiler suite with support not only for C++, but also fortran, go, and others. It is a free variant (GPL) of a compiler suite, and as such available for all major operating systems and hardware. Alternatives can be MSVC, clang/llvm, Intel Compiler Suite.

If you need a newer version of GCC compiler, you find all relevant information to build you own compiler on the GCC Web page.
APPENDIX A. SETUP OF DEVELOPMENT ENVIRONMENT

Preparation

$ sudo apt install binutils g++ coreutils flex bison subversion

There are some more required packages. But as coreutils, they are usually already installed with the basic system. binutils, flex and bison are but usually missing. subversion is necessary for downloading the sources.

Next, download the sources via SVN as normal user. Please, have a look, which release, branch or tag you need.

$ svn co svn://gcc.gnu.org/svn/gcc/tags/gcc_9_1_0_release gcc
$ cd gcc
$ contrib/download_prerequisites

Building

$ mkdir ~/build & cd ~/build
$ ../gcc/configure --prefix=$HOME/Progs/gcc/9.1.0 --disable-multilib \
   --enable-languages=c,c++,fortran
$ make -j 2  # <- How many CPU cores do you have?
$ make install

Be very patient!

Post-Installation Configuration

$ echo 'export PATH=$HOME/Progs/gcc/9.1.0/bin:$PATH' >> ~/.bashrc
$ echo 'export LD_LIBRARY_PATH=$HOME/Progs/gcc/9.1.0/lib64:$LD_LIBRARY_PATH' >> ~/.bashrc

Test

First, check that you have the right GCC version.

$ gcc --version

Compiling and running.

$ g++ -o test test.cxx
$ ./test

Hello!

To use it in cmake, you should set the environment variables CC and CXX.
A.2. **SETUP OF DEVELOPMENT ENVIRONMENT UNDER LINUX**

A.2.3 Boost

Boost is a vast test bed for new C++ concepts and packages that might once become or are already part of the STL. Mostly, it is a header-only C++ library (as the STL), and can be used without installation. But some parts require a compilation, specifically to provide runtime-libraries.

**Preparation**

Download the source of the latest release from the [Boost Download Page](https://www.boost.org/users/download.html). Here, I use for example version 1.70.0.

```bash
$ wget https://dl.bintray.com/boostorg/release/1.70.0/source/boost_1_70_0.tar.gz
$ tar xf boost_1_70_0.tar.gz
$ cd boost_1_70_0
```

**Building**

```bash
$ ./bootstrap.sh --prefix=$HOME/Progs/boost/1.70.0
$ ./b2 install
```

*b2* has many options. The more relevant may be

```bash
$ ./b2 cxxflags="-std=c++11" --prefix="..." --layout=versioned \ 
  threading=multi link=shared variant=release,debug address-model=64 \ 
  -j8 install
```

In this way, you can specify which C++ standard you want. We also want versioning, in order to have several boost versions in parallel. We want support for multi-threading, shared libraries (instead of statically linked ones), and both release and debug variants installed. 64-bit addressing, sure! And use 8 threads to build boost, which is finally installed to the folder specified as prefix. Most of this is but already the default!

If you wish to use the Python bindings, you probably have to modify this a bit.

```bash
$ ./bootstrap.sh --prefix=$HOME/Progs/boost/1.70.0 --with-python=$(which python2.7) \ 
  --with-python-version=2.7
$ ./b2 install
```

(./bootstrap.sh has the --help option.) And to get also the binaries installed need to compile projects via *b2* (*bjam*).

```bash
$ cd tools/build
$ ./bootstrap.sh
$ ./b2 install --prefix=$HOME/Progs/boost/1.70.0
```
APPENDIX A. SETUP OF DEVELOPMENT ENVIRONMENT

Post-Installation Configuration

As large parts of the Boost library are header only, you can in those cases use the compiler’s or CMake’s header include mechanisms. For CMake, Boost already provides automatic detection (find). In the simplest case, you only need to set

```bash
$ echo 'export CMAKE_PREFIX_PATH=$CMAKE_PREFIX_PATH:$HOME/Progs/boost/1.70.0' >> ~/.bashrc
$ echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/Progs/boost/1.70.0/lib' >> ~/.bashrc
$ echo 'export CPLUS_INCLUDE_PATH=$HOME/Progs/boost/1.70.0/include' >> ~/.bashrc
$ echo 'export PATH=$PATH:$HOME/Progs/boost/1.70.0/bin' >> ~/.bashrc
```

Test

Select one of the examples from Boost: Getting Started, to test the Boost installation. We provide also some boost hands-on exercises in this course which can be used for this purpose.

RegEx Test Case  Boost provides a test case with the regular expression subsystem. The example.cxx contains a code that checks from input whether it starts with Subject: Re: or Subject: Aw:

```cpp
#include <boost/regex.hpp>
#include <iostream>
#include <string>

int main() {
    std::string line;
    boost::regex pat( "Subject: (Re: |Aw: )*.*" );
    while (std::cin) {
        std::getline(std::cin, line);
        boost::smatch matches;
        if (boost::regex_match(line, matches, pat))
            std::cout << matches[2] << std::endl;
    }
}
```

The CMakeLists.txt file has the following content.

```cmake
cmake_minimum_required(VERSION 3.28)
project(Example)
find_package(Boost 1.70.0 EXACT REQUIRED regex)
add_executable(Example example.cxx)
target_link_libraries(Example Boost::regex)
```

If EXACT is not specified, the specified Boost is interpreted as minimum required version.

In the case of header-only cases of boost, the part after REQUIRED is not needed. And the linking of libraries is not needed in that case (see also CMake: FindBoost). An example for using Boost::odeint (header-only) looks as follows.

```cmake
cmake_minimum_required(VERSION 3.5.2)
project(lorentz)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
set(CMAKE_CXX_EXTENSIONS False)
find_package(Boost 1.70.0 REQUIRED)
add_executable(lorentz boostODE_Lorenz.cxx)
target_include_directories(lorentz PRIVATE ${Boost_INCLUDE_DIRS})
```
A.2. SETUP OF DEVELOPMENT ENVIRONMENT UNDER LINUX

A.2.4 CERN ROOT

CERN Root is a C++ class framework for data analysis in almost the widest sense you can imagine. It comes with classes for functions, graphs, histograms, trees, etc., which can also be plotted to screen or file.

Preparation

Download the sources from the CERN ROOT Homepage. The the tarball from Pro branch.

```
$ sudo apt install git dpkg-dev cmake g++ gcc binutils libx11-dev libxpm-dev \   libxft-dev libxext-dev gfortran libssl-dev libpcre3-dev xlibmesa-glu-dev \   libglew1.5-dev libftgl-dev libmariadb-dev libfftw3-dev libcfitsio-dev \   graphviz-dev libavahi-compat-libdnssd-dev libldap2-dev python-dev \   libxml2-dev libkrb5-dev libgsl0-dev libqt4-dev
```

Building

```
$ tar xf root_6.18.0.tar.gz
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/Progs/root/6.18.0 ../root-6.18.0
$ make -j 2
$ make install
```

The root-version is here taken only as example.

Post-Installation Configuration

```
$ echo 'source $HOME/Progs/root/6.18.0/bin/thisroot.sh' >> ~/.bashrc
```

Restart the terminal, or execute source ~/.bashrc.

Test

A simple test is the following.

```
$ root -l
root [0] TF1 f("f","x*exp(-x)",0,10);
root [1] f.Draw();
```

When you now see a window with a plotted function, everything went right. .q<Enter> finishes the interactive root session.

As a compiling example, you can write a normal C++ program file (main.cxx)

```cpp
#include "TMath.h"
#include "TF1.h"
#include "TCanvas.h"
int main() {
    TCanvas c("c","c",800,600);
    TF1 f("f","x*exp(-x)",0,10);
    f.Draw();
    c.SaveAs("test.png");
    c.SaveAs("test.pdf");
}
```
and compile it either in a simple case via

```
$ g++ -o main $(root-config --cflags --libs) main.cxx
```

or, in a more complicated case using CMake—here the `CMakeLists.txt`

```cmake
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(event)
find_package(ROOT REQUIRED)
include(${ROOT_USE_FILE})
add_executable(main main.cxx)
target_link_libraries(main ${ROOT_LIBRARIES})
```

(see also CERN ROOT: Integrate ROOT into my Project with CMake)

Note here that CMake does not require that project and executable targets to have the same name.

### A.2.5 Eigen

Eigen is a header-only C++ library for linear algebra—matrix and vector operations, and solution of linear equations. To understand (as user) how it works requires knowledge about C++ templates and linear algebra.

#### Preparation

Download the sources from the [Eigen Homepage](http://eigen.tuxfamily.org).

#### Building

```bash
$ tar xf eigen-3.3.7.tar.gz
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/Progs/Eigen/3.3.7 ../eigen-3.3.7
$ make install
```

The Eigen-version is here taken only as example.

#### Post-Installation Configuration

```bash
$ echo 'export CMAKE_PREFIX_PATH=$HOME/Progs/Eigen/3.3.7:$CMAKE_PREFIX_PATH' >> ~/.bashrc
```

Restart the terminal, or execute `source ~/.bashrc`.

#### Test

First of all, Eigen comes with CMake support. So, you can add Eigen to your `CMakeLists.txt` file as described [here](http://eigen.tuxfamily.org).

The Eigen documentation page comes with lot of examples, you can copy and paste into a simple C++ source file. See, for instance, [here](http://eigen.tuxfamily.org) for sparse matrices and matrix free linear equation solvers. A [Getting Started](http://eigen.tuxfamily.org) is but possibly the easier entry point.

Please, use the search capabilities of that page (or Google) to look for dedicated topics or examples. We will give a Tutorial later on some basic usage.
Appendix B

Extended CMake-Tutorial

B.1 Tutorial: How to use CMake to build Software

In this first part, we will outline how to use the CMake tool suite once a CMake managed software project is given. We assume, a version of CMake is already installed on your system, as well as Make (e.g. GNU Make) and a C++ compiler (e.g. GCC).

For the purpose of this tutorial, and for illustration, you can download CMake itself e.g. from https://cmake.org/download/ (Source code, i.e. cmake-<version>.tar.gz or similar).

B.1.1 The simple CMake Configure and Build Process

The first step is – Unpack the tarball:

```bash
$ tar xf cmake-<version>.tar.gz
```

The result is usually a folder cmake-<version>. The complete path to it shall be called "path-to-source-directory" (can also be a relative path). Next, create a new directory "build" – inside the cmake-folder or somewhere else – and change to it,

```bash
$ mkdir build && cd build
```

Finally, you only need to execute

```bash
$ cmake <path-to-source-directory>
$ make
$ make install # That naive way, does not work without administrative rights!!
```

The last step should NOT be done (currently) unless you want to install cmake system-wide (for which you need root/admin privileges)! The Makefile generated in this process (and processed via "make" in the second step) is fully equipped with standard POSIX compliant targets ("make install", "make clean", ...), can be processed in parallel ("make -j 8 all" for 8 parallel threads), and so on. For listing targets, try "make help"! For verbose output, you can also call make via

```bash
$ make VERBOSE=1
```

Building the binaries not inside the source directory of the software package has some advantage. You can simply remove the build folder, and do not pollute the source directory.
B.1.2 Adaptation of Configuration

CMake already admits a lot of fine-tuning and configurations before calling "make"! The most essential one is probably the installation path, which you may want to set to a special folder (where you have access permissions). In order to do that, just issue

```bash
$ cmake -DCMAKE_INSTALL_PREFIX=<install-path> <path-to-source-directory>
```

This is one way to give "cmake" certain adapted variables "-D<variable-name>=<value>". In this way, you can determine the installation location, the compilers (which one, paths) and corresponding flags (warnings, optimization, verbosity, ...) used, but also all kinds of project-specific variables (e.g. to add optional add-on and third-party packages, or switching them off) controlling the built of the software package.

Of course, you usually don’t know these variables in advance. In order to get them, you can use interactive tools like "ccmake" (or "cmake-gui", if installed). It is a good idea to check the CMake configuration anyway – unless you are completely sure that the "-D..." settings comprehended all kinds of variables you need for your built!

```bash
$ ccmake <path-to-source-directory>
```

inside the build directory. This will open a nurese interactive terminal session, showing you the set of variables set or not yet set – that is, automatically found or not. Change the items you wish and issue "c"-key for "configure" until you are satisfied with your choice, and the configuration does not throw any errors anymore. Then enter "g"-key to "generate" the final Makefile (or other build-configuration file).

Errors can generally be investigated in the CMakeFiles/CMakeError.log file. The normal output goes to CMakeOutput.log in the same folder. Paths (e.g. the source path) and other information can be retrieved from CMakeFiles/CMakeDirectoryInformation.cmake.

Per default, you see only a small subset of all variables via "cmake". Use "t"-key to "toggle" the view to see all available variables. These are also the variables implicitly created by cmake, such as CMAKE_CXX_COMPILER or CMAKE_CXX_FLAGS, etc. Those can be used to fine tune the final Makefile configuration. (Specifically for MPI programs using "mpiCC" or similar wrappers, this is the way of choice). Of course, CMake (and the generated Makefile) also reacts on environment variables such as CXX or CXXFLAGS.

Also very important is CMAKE_BUILD_TYPE which can be "Debug" or "Release" (or more). According to this variable, you can easily switch between optimized release and debugging build.

For libraries, you might want to set BUILD_SHARED_LIBS to ON. Default is OFF (unless specified differently in the CMakeLists.txt).

B.1.3 Testing

Most packages come also with some kind of test suite. You can issue

```bash
$ ctest
```

after the built. This works only if the variable BUILD_TESTING was set to ON. You should see a (maybe long) list of tests, and finally a summary of successful and failed tests. You can also execute instead

```bash
$ make test
```

B.1.4 Software Development Process

The configuration step via "cmake" needs to be done only, if files or folders were added to or removed from your software project – or your build configuration is to change. Only changing file contents of source code (i.e. writing more or changing code in the persisting source files) does NOT require
a reconfigure via "cmake"! Execute simply "make" in the build directory. This in turn updates only targets, which directly depend on the changed files, as usual for make.

As you can create as many out-of-source build directories, you can separate different builds e.g. with different configurations (Debug/Release, ...), and all this without polluting the source directory. This makes the development process much easier.

**B.2 Tutorial: How to write CMakeLists.txt Files (aka Project Management with CMake)**

We recommend to pass through the online CMake Tutorial Guide. It develops in steps a CMakeLists.txt file from a simple one-file project up to complex many file projects with external dependencies and even configuration-time code generation/adaptation.

Many of these topics have been introduced in "Mastering CMake", [20] (downloadable from here) as separate issues but seeing how they all work together in an example project can be very helpful.

We would also emphasize that CMake is rather platform independent. It can be used on Linux, Max OS X, and Windows. It supports via its generators also third party IDEs such as CodeBlocks, Eclipse, Ninja, Unix Makefiles (default on Linux), MS Visual Studio, and more.

Furthermore, CMake also supports packaging tools, such that both binary install packages, and source file archives can be generated (possibly after installation of necessary supply software such as compression tools at its simplest - building RPM files requires `rpmbuild`, for instance).
Appendix C

Cheat Sheets, Web-Links, Literature

To google is of course also a valid option!

C.1 Bash/Linux Tools

https://itsfoss.com/linux-commands-cheat-sheets

C.2 CMake

https://cmake.org/documentation
https://cmake.org/cmake-tutorial
http://overapi.com/cmake

C.3 Git

https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

C.4 Doxygen

http://www.doxygen.nl/manual/starting.html
http://mitk.org/images/1/1c/BugSquashingSeminars%242013-07-17-DoxyReference.pdf

C.5 C++

https://isocpp.org/get-started
https://en.cppreference.com
https://cpp.sh
Appendix D

Proposals for Solutions for the Hands-On Exercises of Part 1

D.1 Solution: Statistics

D.1.1 Solution: RNG

According to `cppreference: random`, we can for instance generate random numbers also for a Gamma distribution with parameters $\alpha = 2$ and $\beta = 1$, or an exponential distribution with parameter $\lambda = 1$.

```cpp
#include <iostream>
#include <random>

int main() {
    std::mt19937 e1(1234);
    std::uniform_real_distribution<double> uniform_dist {0,1};
    std::gamma_distribution<> gd {2,1};
    std::exponential_distribution<> ed {1};
    for(int i=0; i<100000; ++i)
        std::cout << uniform_dist(e1) << "\t" << gd(e1) << "\t" << ed(e1) << "\n";
}
```

The resulting histograms look as follows.

![Histograms](image)

Figure D.1: Distributions created using STL. Left: Uniform(0,1); Middle: Gamma(2,1); Right: Exponential(1).

D.1.2 Solution: MCMC

According to the recipe, we generate a test step, and accept is with some probability.

```cpp
#include <iostream>
#include <random>
```
double fgamma(double x) {
    return x*std::exp(-x);
}

int main() {
    std::mt19937 e1(1234);
    std::uniform_real_distribution<double> ud {0,1};
    std::normal_distribution<> nd {0,1};
    double x=1., xtmp;
    int accept=0;
    for(int i=0; i<100000; ++i) {
        xtmp=x+nd(e1);
        if(ud(e1) < fgamma(xtmp)/fgamma(x)){
            x=xtmp;
            accept++;
        }
        std::cout << x << \n;
    }
    std::cerr << " acceptance rate: " << (double)accept/100000. << \n;
}

Compiling and executing goes like this.

$ g++ -o main main.cxx.
$ ./main
acceptance rate: 0.72397

So, we have an acceptance rate of about 72.4 %, which is rather OK. This is confirmed also by
inspection of the resulting histogram (it should roughly look like the middle histogram of Figure D.1).

As mentioned, one could store also fgamma(x) as a variable in order not to permanently recalculate
it.

D.1.3 Solution: Mean, Standard Deviation, Median, Quantiles with Algo-
rithms

We choose a Gaussian (normal) distribution with $\mu = 0$, and $\sigma = 1$. The distribution is plotted in
Figure D.3.
D.1. SOLUTION: STATISTICS

Figure D.3: Gaussian distribution with $\mu = 0$ and $\sigma = 1$.

```cpp
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <random>

int main() {
    using namespace std;
    const int N=100000;
    mt19937 e1(1234);
    normal_distribution<> nd {0,1};
    vector<double> v;
    for(int i=0; i<N; ++i)
        v.push_back(nd(e1));
    // mean (decimal point at 0. is important!!!)
    double mean=accumulate(v.begin(), v.end(), 0.)/N;
    cout << " mean = " << scientific << mean << "\n";
    // standard deviation
    double s = accumulate(v.begin(), v.end(), 0.,
                           [] (double x, double y) {
                               return x+y*y;
                           })/N;
    s = sqrt(s/N-mean*mean);
    cout << " stddev = " << s << "\n";
    // median == 50% quantile
    nth_element(v.begin(), v.begin()+v.size()/2, v.end())
    cout << " median = " << v[v.size()/2] << "\n";
    sort(v.begin(),v.end());
    int mid=v.size()/2;
    double median = mid%2 == 0 ? (v[mid]+v[mid-1])/2. : v[mid];
    cout << " median = " << median << "\n";
    // quantiles: 1. quartile == 25% quantile, ...
    cout << " 1. quartile = " << v[v.size()/4] << "\n";
    cout << " 3. quartile = " << v[3*v.size()/4] << "\n";
    cout << " 5% quantile = " << v[(int)(0.05*v.size())] << "\n";
    cout << "95% quantile = " << v[(int)(0.95*v.size())] << "\n";
    }
```

The results are not very accurate according to the rather low statistics. But the values are comparable
APPENDIX D. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 1

to what can be found e.g. here.

<table>
<thead>
<tr>
<th>mean</th>
<th>-7.118035e-03</th>
</tr>
</thead>
<tbody>
<tr>
<td>stddev</td>
<td>1.000806e+00</td>
</tr>
<tr>
<td>stddev</td>
<td>1.000806e+00</td>
</tr>
<tr>
<td>median</td>
<td>-1.196262e-02</td>
</tr>
<tr>
<td>median</td>
<td>-1.203505e-02</td>
</tr>
<tr>
<td>1. quartile</td>
<td>-6.841260e-01</td>
</tr>
<tr>
<td>3. quartile</td>
<td>6.735468e-01</td>
</tr>
<tr>
<td>5% quantile</td>
<td>-1.648030e+00</td>
</tr>
<tr>
<td>95% quantile</td>
<td>1.636929e+00</td>
</tr>
</tbody>
</table>

We demonstrated several ways e.g. to determine the standard deviation. If you don’t need all quantiles, `nth_element` is probably more efficient than `sort`. So, for instance finding the 25 % quantile, using `nth_element` looks as follows.

```c++
nth_element(v.begin(), v.begin()+((int)(0.25*v.size())), v.end());
```

Instead of the for-loop, to generate the random numbers in the sequence of the vector, you can also initialize the vector with a size `N`, and fill it via `std: : generate_n` (see `cppreference: std: : generate_n`).

```c++
vector<double> v(N);
generate_n(v.begin(), N, [= &e1, &nd](){return nd(e1);});
```

The more you can put into the STL structures, the easier and more generic becomes your code usually. This includes specifically the vast usage of Lambda functions where appropriate. The vector size must be specified at creation time of the vector to initialize the internal array. You can try what happens if you do not do so! It is also important to use parentheses (... ) to specify the size of the vector at initialization! vector<double> v {N}; would create a vector of size one, with the content `v[0]=N`.

**D.1.4 Solution: Integrals using MC Methods**

For $I = \int_{-3}^{3} \sin(x) \, dx$, the MC integration could look like the following.

```c++
#include <cmath>
#include <iostream>
#include <random>

int main() {
    using namespace std;
    const int N = 10000000;
    mt19937 rng(1234);
    uniform_real_distribution<double> ud(-3,3);
    double sum=0;
    for(int i=0; i<N; ++i) sum+=sin(ud(rng));
    sum*=6./N;
    cout << " I = " << scientific << sum << "\n";
}
```

It results in

$I = -6.225863e-04$

Which is within the expected precision, which can be expressed by the standard deviation (approximated by $s$, see above) by $s/\sqrt{N} \approx O(1/\sqrt{N} \sim O(10^{-4})).$

For the following, it is maybe instructive to to write this a bit more generically. Unfortunately, no STL algorithm could be found useful for this task (at least not without substantial efforts to generate dummy iterators). We create a function which performs the accumulation for us, `int_unif`. And look-ahead: `std::vector` also has a constructor, which takes an initializer list – a concept introduced in C++11. And such a list is represented by curly braces. We will meet them later in this course again.

\[\text{Look-ahead: std::vector also has a constructor, which takes an initializer list – a concept introduced in C++11. And such a list is represented by curly braces. We will meet them later in this course again.}\]

\[\text{This results from the Central Limit Theorem!}\]
in order to be a bit more generic, we define (via using declaration) a function pointer, such that int_unif can integrate several different functions.

```cpp
#include <cmath>
#include <iostream>
#include <random>

using fptr = double(*) (double);

double int_unif(int N, double low, double up, fptr f, int seed=1234) {
    using namespace std;
    mt19937 rng(seed);
    uniform_real_distribution<double> ud(low,up);
    double sum=0;
    for(int i=0; i<N; ++i) sum+=f(ud(rng));
    return (up-low)*sum/N;
}

int main() {
    std::cout << "I = " << std::scientific
        << int_unif(10000000,-3,3,std::sin) << "\n";
}
```

In order to integrate \( I = \int_0^\infty x e^{-x} \, dx \), we simply reuse int_unif. All we have to do is to define a function that we can transmit to int_unif (we use a Lambda function here), and the limits, \((0,a)\).

```cpp
... auto gam = [](double x) {return x*std::exp(-x);};
std::cout << " I = " (std::scientific
    << mean_unif(10000000,0,5,gam) (\"n\";
...]
```

The lower limit is 0. For the upper limit, we use 100, 10000, and 100000, and look whether the value for \( I \) converges, what it obviously does not (see Table D.1).

<table>
<thead>
<tr>
<th>a</th>
<th>100</th>
<th>10000</th>
<th>100000</th>
<th>1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0.99952</td>
<td>0.98804</td>
<td>0.9496</td>
<td>2.9835</td>
</tr>
</tbody>
</table>

Table D.1: Different values of \( I \) for different upper limits \( a \).

As outlined in the task description, we use in our next attempt importance sampling, and reformulate the integral as follows.\(^3\)

\[
I = \int_0^\infty x e^{-x} \, dx = \int x \, d\mu(x),
\]

where \( \mu(x) \) is a measure on the real numbers (in that case, the positive real numbers). As \( d\mu = e^{-x} \, dx \), we can identify this measure as exponential function. And because

\[
\int d\mu = \int_0^\infty e^{-x} \, dx = \left[-e^{-x}\right]_{x=0}^{x=\infty} = -0 - (-1) = 1,
\]

the measure is already normalized. That is, \( I \) can be interpreted as a mean value with respect to a measure (probability distribution function) on the real numbers.

\[
I = \int x \, d\mu(x) = E_{\mu(x)}(x).
\]

And this is exactly what we exploit. We throw \( x_i \) according to an Exponential distribution, and average over the remaining function, which is here just \( x \). The final program looks as follows.

```cpp
#include <cmath>
#include <iostream>
#include <random>

using fptr = double(*) (double);

double mean_exp(int N, fptr f, int seed=1234) {
```

\(^3\)Please consult classical texts on Lebesgue integrals!
using namespace std;
mt19937 rng(seed);
exponential_distribution<double> ed(1);
double sum=0;
for(int i=0; i<N; ++i) sum+=f(ed(rng));
return sum/N;
}
int main() {
    auto xf = [](double x) { return x;};
    std::cout << "I = " << std::scientific << mean_exp(1000000,xf) << "\n";
}

And indeed, a result of 0.999892 is rather acceptable.

By this mean, we can now approximate any integral of the form \( \int_0^{\infty} f(x) e^{-x} \, dx \) – something, which is achievable using deterministic methods only with Gauss quadrature, which are presumably more difficult to understand.

If you now acquired a taste for it, you can try out yourself on the following integral types.

\[
I = \int_{-\infty}^{\infty} f(x) e^{-x^2} \, dx.
\]

D.1.5 Solution: Gambling

Solution: Throwing a Die

cppreference: uniform_int_distribution gives the solution already as example. So, let us now only convince that we indeed have a uniform distribution.

```
#include <random>
#include <iostream>
#include <vector>
#define N 10000000
int main() {
    std::mt19937 rng(1234);
    std::uniform_int_distribution<> iud(1, 6);
    std::vector<double> v {0,0,0,0,0,0};
    for(int i=0; i<N; ++i) v[iud(rng)-1]++;
    for(int i=0; i<6; ++i)
        std::cout << i+1 << " : " << std::scientific << v[i]/N << \
                   "\n";
}
```

Appears to be the case. Who is really keen, can now do hypothesis testing!

Solution: Random Shuffling (Permutations)

Fortunately, STL already provides an algorithm with the correct name. The only larger step is to recognize that std::string is a container (so-to-speak a std::vector<char>), such that algorithms can be used directly.

```
#include <algorithm>
#include <iostream>
#include <random>
#include <string>
int main() {
    std::string s{"ABCDEFGABDCXYZ"};
    std::mt19937 rng(1234);
    for(int i=0; i<10; ++i) {
        std::shuffle(s.begin(),s.end(),rng);
        std::cout << s << "\n";
    }
}
```
D.2. Solution: Algebra

D.2.1 Solution: Vector-Vector-Multiplication

You may probably start with something like this.

```cpp
#include <iostream>
#include <vector>
int main() {
    using namespace std;
    vector<double> v1 {1,2,3,4,5,6};
    vector<double> v2 {6,5,4,3,2,1};
    double res = 0;
    for(int i=0, i<6; ++i)
        res += v1[i]*v2[i];
    cout << " v1*v2 = " << res << "\n";
}
```

But using a name algorithm (of a proper name), we can shorten this, and write a bit more expressively.

```cpp
#include <iostream>
#include <numeric>
#include <vector>
int main() {
    using namespace std;
    vector<double> v1 {1,2,3,4,5,6};
    vector<double> v2 {6,5,4,3,2,1};
    cout << " v1*v2 = " << inner_product(v1.begin(),v1.end(),v2.begin(),0.) << "\n";
}
```

This is comparable to the following solution.

```cpp
#include <iostream>
#include <numeric>
#include <vector>
double dot(const std::vector<double>& v1, const std::vector<double>& v2) {
    return inner_product(v1.begin(),v1.end(),v2.begin(),0.);
}
```
In any case, the problem is that we cannot ensure that both vectors are of the same length. Here in our dot function, we could at most check whether the size of v1 and v2 are the same. But first, this happens during run-time. We would prefer a check during compile time. And second, what will you do, if you notice a mismatch?! This will be business of a later chapter in this course.

Let’s have a look on another data structure. Maybe std::array.

This looks good! At least this guarantees that the both structures are of the same definite length. And this can already be checked during compile time!

### D.2.2 Solution: Matrix-Vector-Multiplication

We pick up the idea of the last solution of the previous subsection.

The advantage of the two-dimensional array is that we can recover the matrix dimensions. The disadvantage is, of course, that the matrix rows are possibly not consecutive in the memory – a performance issue.

In that way, we could even make the program type-checking, i.e. that the product of a $4 \times 6$-matrix and a vector of length 6 must result in a vector of length 4. Had we accidentally declared $r$ as Vector_6, the compiler had complained.
You may have wondered, why the initialization of $M$ contains an extra pair braces, {...}. As we define the matrix as sort of array of array, the inner nested braces represent exactly this. The outermost braces are just the initializer list braces.\(^4\)

### D.2.3 Solution: Matrix-Matrix-Multiplication

Without much fuzz – here one solution.

```cpp
#include <array>
#include <iostream>
#include <numeric>

using Matrix_4_6 = std::array<std::array<double, 6>, 4>;
using Matrix_6_2 = std::array<std::array<double, 2>, 6>;
using Matrix_4_2 = std::array<std::array<double, 2>, 4>;

Matrix_4_2 dot(const Matrix_4_6& M, const Matrix_6_2& P) {
    Matrix_4_2 r;
    for (int i = 0; i < 4; ++i)
        std::fill(r[i].begin(), r[i].end(), 0.);
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 2; ++j)
            for (int k = 0; k < 6; ++k)
                r[i][j] += M[i][k] * P[k][j];
    return r;
}

void print(const Matrix_4_2& Q) {
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 2; ++j)
            std::cout << Q[i][j] << ' ';
    std::cout << '
';
}

int main() {
    Matrix_4_6 M {{{1,2,3,4,5,6},
                   {2,3,4,5,6,7},
                   {3,4,5,6,7,8},
                   {4,5,6,7,8,9}}};
    Matrix_6_2 v {{{6,5},
                   {5,4},
                   {4,3},
                   {3,2},
                   {2,1},
                   {1,0}}};
    std::cout << " M*v = \n";
    print(dot(M, v));
}
```

As we decided for row-major order, we cannot easily do the scalar product using `std::inner_product`, anymore. Any by hard-coding the dimension numbers, we made it very difficult to extend this code. We lost any flexibility. But we will regain it with templates, and other means.

### D.3 Solution: Cryptography – Vigenère-XOR Encryption

All three solutions read as sole command-line parameter the password. The plain text is read from `std::cin`, and the cipher text written to `std::cout`. Using input and output redirection from the shell, it can be read from a file, and written to a file.

```
$ ./Vigenere <Password> < infile > outfile
```

The almost standard "C way" is the following.

```cpp
#include <iostream>

Don't worry if you do not comprehend this now!
```
APPENDIX D. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 1

```cpp
#include <cstdio>
#include <cstdlib>

int main(int argc, char * argv[]) {
    if(argc != 2) {
        std::cout << " Usage: " " password < plain-text > cipher-text"
        "\n";
        return 1;
    }
    std::string pwd(argv[1]);
    int ch, ind=0;
    while((ch=std::getchar())!=EOF) {
        ind=(ind+1)%pwd.length();
        std::putchar(chˆpwd[ind]);
    }
}
```

std::putchar requires the header cstdio, which is a old C header (wrapped into the STL).

A bit more C++-like is it to use and algorithm, functors and inheritance (comes later in parts 3 and 4, respectively).

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>

struct Encryptor : public std::unary_function<char, char> {
    Encryptor(std::string password) : pwd{password} {};
    char operator()(const char & ch) {
        index=(index+1)%pwd.length();
        return chˆpwd[index];
    }
};

int main(int argc, char * argv[]) {
    if(argc != 2) {
        std::cout << " Usage: " " password < plain-text > cipher-text"
        "\n";
        return 1;
    }
    std::transform((std::istreambuf_iterator<char>(std::cin)),
                   std::istreambuf_iterator<char>(),
                   std::ostreambuf_iterator<char>(std::cout),
                   Encryptor(std::string(argv[1])))
};
```

This code is longer than the C way, but more consequent in using the STL (especially iterators). Furthermore, the actual work is hidden from the one-liner std::transform call, which represents itself here in a rather readable declarative style.

Really fancy is it to use Lambda functions (also currently in lack of knowledge about functors).

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>

int main(int argc, char * argv[]) {
    if(argc != 2) {
        std::cout << " Usage: " " password < plain-text > cipher-text"
        "\n";
        return 1;
    }
    std::string pwd(argv[1]);
    int ind=0;
    std::transform((std::istreambuf_iterator<char>(std::cin)),
                   std::istreambuf_iterator<char>(),
                   std::ostreambuf_iterator<char>(std::cout),
                   std::transform<char, char, std::string>(
                       std::string(argv[1])));
```
SOLUTION: NUMERICS

In all three solutions, the way is to commonly run through the password and the plain text arrays, and XOR character-wise. The password is usually shorter than the plain text. Thus, hitting its end, the password is read again from the beginning. An alternative could be to read the password from right to left (backward) until reaching the beginning, and then reading from left to right, an so on.

Considering the very last solution with Lambda functions, people often ask "Where is this object oriented?!". In many respects, using a declarative style, we have here more of a functional style of programming. And still, these obey the principles of object-orientation (google: "SOLID Wiki") like encapsulation. It is a good idea to get to know to that style.

Btw., a global free function might not work here, as we need to remember the state (ind), and we need to get the password into the function (pwd). The other way to achieve this – next to Lambda Functions – is to use functors.

D.4 Solution: Numerics

Solution: Zeros of Functions using Bisection

As functions are objects, they should anyway be manageable as parameters. So, we use immediately function pointers.

```cpp
#include <cmath>
#include <iostream>
using Fprt = double(*)(double);
double bisection(double ml, double mr, Fprt f, double eps=1.e-14) {
    if(ml>mr || eps<=0.) return -1e100;
    double fl=f(ml), fr=f(mr), xm, fm;
    while((mr-ml)>eps){
        xm = 0.5*(ml+mr);
        fm = f(xm);
        if(fl*fm<0)
            mr=xm, fr=fm;
        else
            ml=xm, fl=fm;
    }
    return 0.5*(ml+mr);
}
void print_zeros(double min, double max, int N, Fprt f) {
    double delta=(max-min)/N, x=min;
    while(x<=max) {
        if(f(x)*f(x+delta)<0)
            std::cout << bisection(x,x+delta,f) << "\n";
        x+=delta;
    }
}
int main() {
    print_zeros(-4,4,100,std::sin);
}
```

Solution: Derivatives of Functions

This can become a bit tricky with not much more than pure functions, Lambda functions, and function pointers.

```cpp
#include <cmath>
```
In order to obtain for df and d2f a function signature according to Fptr, such that we can apply 
bisection to it, we created a global file pointer gf, which is set to the correct function value inside 
print_min_max. Inside df and d2f, we created a local file pointer lf, which we set to the global 
one. This is necessary to get it captured by the capture clause of the Lambda functions. This is not 
very elegant. But it works. We will see later a good alternative, when using so-called Function Objects 
(aka Functors). The rest is more or less straightforward.

To show how it works, we also changed the bisection routine from a iterative procedure to a recursive 
one. It can be very elegant and shorter to implement than the iterative approach. You can test later, 
which of these both approaches is better performing.
D.5 Solution: STL Containers and Algorithms – Sieve of Eratosthenes

#include <algorithm>
#include <numeric>
#include <iostream>
#include <list>

int main() {
  std::list<int> ilist(100000);
  std::iota(ilist.begin(), ilist.end(), 2);
  for (auto it = ilist.begin(); it != ilist.end();) {
    ilist.erase(std::remove_if(++it, ilist.end(),
      [](int iv){ return iv%(*it)==0; }), ilist.end());
  }
  for (auto p : ilist) std::cout << p << " ";
  std::cout << " # prime numbers: " << ilist.size() << "\n";}

This should work with any container (std::vector, std::deque). The important thing to recognize is the so-called erase-remove idiom. std::remove, or its variant, std::remove_if does not actually remove something. It just rearranges the container in such a way that all elements sorted out are put to the rear of the container. Only std::erase cuts them from the container.

Please, also check the other items (e.g. std::iota) at cppreference!

A slightly improved variant is the following one. We erase only once later, after all removed elements are put to the rear of the container. As you can see, a creative use of eit, and of iterators in general, is not forbidden!

#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

int main() {
  std::vector<int> v(100000);
  std::iota(v.begin(), v.end(), 2);
  auto eit = v.end();
  for (auto it = v.begin(); it != eit;)
    eit = std::remove_if(++it, eit, 
      []{ return iv%(*it)==0; });
  v.erase(eit, v.end());
  for (auto p : v) std::cout << p << " ";
  std::cout << " # prime numbers: " << v.size() << "\n";}

We also used a std::vector<int> instead of a list, just to show that they can be used interchangeably.

D.6 Solution: Computer – Unix Tools

D.6.1 wc – word count

The simplest solution is indeed the straightforward usage of the STL. The most direct way requires a string stream. This will be explained later in the course. But in short, you can make each string being a stream of characters (similar to std::cin).

#include <iostream>
#include <sstream>
#include <string>

int main() {
  int chars=0, words=0, lines=0;
  std::string line, tmp;
  while(std::getline(std::cin,line)) {
    lines++;
    chars += line.length()+1; // +1 <=> '\n'
APPENDIX D. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 1

```cpp
std::istringstream iss(line);
while(iss >> tmp)
    words++;
std::cout << chars << " " << words << " " << lines << '\n';
}

We manually need to count the newline character. Also more complex STL usage such as iterators is feasible. This example uses a Lambda function.

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>

int main() {
    int chars=0, words=0, lines=0;
    char lc;
    std::for_each((std::istreambuf_iterator<char>(std::cin)),
                   std::istreambuf_iterator<char>(),
                   [&chars, &words, &lines, &lc](char c){
                       chars++;
                       if((c=='\\n') && lc!='\\n') words++;
                       if((c=='\\n')) lines++;
                       lc=c;
                   });
    std::cout << chars << " " << words << " " << lines << '\n';
}

Still, it should impress that it is essentially a one-liner! It is slightly more difficult to work exclusively on the basis of a character stream. Therefore, we must check manually when a word and a line finishes.

And, the last character in a line might be a white-space. So, we need to look also on the previous character lc in order to decide whether to count words up, or not.

D.6.2 grep – text search in files

For the purpose, we use the regex library of the STL (see cppreference: Standard library header regex, and also look at the examples). The rest is rather straightforward – only print out lines that contain a match.

```cpp
#include <fstream>
#include <iostream>
#include <regex>

int main(int argc, char *argv[]) {
    if(argc < 3) {
        std::cerr << " Syntax: grep <pattern> <file1 <file2> ...
        return 0;
    }
    std::regex myregex(std::string(argv[1]),
                       std::regex_constants::ECMAScript|std::regex_constants::icase);
    for(int i=2; i<argc; ++i) {
        std::ifstream ifs(argv[i]);
        if(!ifs) {
            std::cerr << " Warning: Could not open file \"
                       << argv[i] << "\": Skipped\n";
            continue;
        }
        std::string line;
        int lineno=0;
        while(std::getline(ifs,line)) {
            lineno++;
            std::smatch sm;
            std::regex_search(line,sm,myregex);
            if(!sm.empty()) {
                std::cout << argv[i] " :\033[1;32m" << lineno " \033[0m:
                do {
                    std::cout << line.substr(0, sm.position(0))
                } while(sm.position(0) < line.length());
            }
        }
    }
    return 0;
}
```
Is it not astonishing with which rather small amount of programming work this can be achieved? The STL provides really versatile features!

Some may have noticed the \033[ characters. These are so-called escape sequences providing here color under bash. For other shells, this may not work!
Appendix E

Proposals for Solutions for the Hands-On Exercises of Part 2

E.1 Solution: C++/Python Coupling – PyBind11

STL containers can be included just by using pybind11/stl.h (PyBind11: STL Containers). That's pretty simple! Let's take as example to square the elements of a STL vector (python list or comparable), and return it (by value – so, no in-place changes). We do this in file square.cxx.

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/stl.h> // <-- !!
#include <string>

std::vector<double> square(std::vector<double> v) {
    for (auto &p : v) p*=p;
    return v;
}
PYBIND11_MODULE(stltest, m) {
    m.def("square", &square, "square a list of floating point elements");
}
```

The corresponding CMakeLists.txt looks as follows.

```cmake
cmake_minimum_required (VERSION 3.5)
project(stltest)
find_package(pybind11 REQUIRED)
pybind11_add_module(stltest square.cxx)
```

Once compiled, the following ipython3 script works immediately.

```python
import matplotlib.pyplot as plt, numpy as np, stltest as st
t = np.arange(0., 5., 0.2)
s = st.square(t)
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.plot(t, s, 'r-')
plt.savefig('python1.pdf')
```

This results in Figure E.1.
E.2 Solution: Statistics

E.2.1 Solution: The CERN ROOT C++ Library

TF1 Formula with Parameters

In ROOT, one can define functions also directly in the constructor. But functions shall be used for fitting, one needs to introduce parameters. We showed already the way via a C++ free function. ROOT can but do more. Look at the following code!

```cpp
TF1 f("f","[0]*x*exp(-[1]*(x-[2]))",0,10);
```

This function has three parameters, which can be estimated via fitting to a histogram! This happens exactly in the same way as outlined in the former examples!

Graphs, Error-Graphs, and std::vector

Next to functions and histograms, there is a rather important class of displaying data – graphs. Graphs can come in several flavors. The simplest is just two arrays $x,y$ representing the data.

```cpp
double x[] = {0,1,2,3,4,5}, y[] = {0,1,4,9,16,25};
TGraph gr(6,x,y);
gr.Draw("AL");
gr.Draw("AC");
```

The TGraph constructor takes the number of pairs, and the two arrays. You can draw a graph similar to drawing functions, or histograms. For graphs, however, you must specify $A$, such that axes are drawn. $L$ and $C$ are for linear interpolated and for smoothly differentiable plotting, respectively.

Of course, you can specify SetLineColor, SetLineWidth and other properties.

Another interesting flavor are error graphs. We show illustratively just the following example. We also illustrate here the combination with std::vector.

```cpp
std::vector<double> x {0,1,2,3,4,5}, y {0,1,4,9,16,25},
ey {0.5, 0.6, 0.5, 0.5, 0.6, 0.7};
TGraphErrors gr(6,&x[0],&y[0],0,&ey[0]);
gr.Draw("AC");
```

Please try this in the CINT environment! The 0 means that there are no error bars in x-direction. You can add another array representing such errors, and complete the graph with such x-direction error bars.
More on ROOT File I/O

Actually, any ROOT object can be stored in a TFile. For TTrees, we’ve already seen this. But also TGraphs, TF1s, TH1Ds, etc. can be stored in the ROOT-file.

```cpp
int main() {
  TFile file("test.root","RECREATE");
  TF1 f("f","x*exp(-x)",0,10);
  f.Write();
  TH1D h("h","h",100,0,10);
  h.FillRandom("f",10000);
  std::vector<double> x(30), y(30);
  double i=0;
  std::generate(x.begin(), x.end(), [&i]() { return i++; });
  std::transform(x.begin(), x.end(), y.begin(),
                 [f](double x) { return f.Eval(x); });
  TGraph gr(x.size(), &x[0], &y[0]);
  gr.Write();
  file.Write();
}
```

Graphs and functions must be written explicitly. Trees and histograms are assigned to a file context automatically. Use the TBrowser to investigate the resulting test.root file.

E.2.2 Solution: Full-Text Analysis – Maps

A solution might look like the following.

```cpp
#include <algorithm>
#include <fstream>
#include <iostream>
#include <map>
#include <string>
#include <vector>
int main(int argc, char* argv[]) {
  using namespace std;
  ifstream ifs(argv[1]);
  if(!ifs) return -1;
  map<string, int> ws;
  string w;
  while(ifs >> w) {
    w.erase(remove_if(w.begin(),w.end(),
      [](unsigned char x){ return ispunct(x); }),w.end());
    transform(w.begin(),w.end(),w.begin(), ::tolower);
    ws[w]++;
  }
  cout << " Size: " << ws.size() << "\n";
  const int NN=20;
  vector<pair<string, int>> vs {ws.begin(),ws.end()};
  partial_sort(begin(vs), begin(vs)+NN, end(vs),
               [](pair<string, int>& p1, pair<string, int>& p2){
                 return p1.second > p2.second;
               });
  for(int i=0; i<min((int)vs.size(),NN); ++i) {
    cout << " " << vs[i].first << " : " " << vs[i].second << "\n";
  }
}
```

In a while loop, we read in the words, transform them (strip punctuation and make all characters lower case), and then use the std::map’s property to specify a key (the string), which must be unique. And each time we meet a key (a word), it will be entered into the map, unless already present. In any case, for this key, the integer value will be counted up.

In the second part, we want to print out the 20 most frequently occurring words. For some reason, the algorithms do not nicely play on maps – at least std::sort refused its service. But one can always
copy a map into a vector, by using the range-constructor. The rest should be almost familiar. As we want to have a sort with respect to the values of the pairs (not the key – this does \texttt{std::map} already automatically), we have to write the predicate (the binary comparison operation) ourselves. We just use a Lambda function for this purpose.

Some last words about the last for-loop. We must convert \texttt{vs.size()} into an \texttt{int}. At this place in \texttt{std::min}, the C++ compiler does not do this automatically, here. In order to do an explicit cast, we use here the C-type cast – prepending the type in parentheses, \texttt{(int)vs.size()}. The modern C++ way is \texttt{static_cast<int>(vs.size())}, instead. We come back to this later in the course.

E.2.3 Solution: A Histogram Class

My initial Solution

```cpp
// Histogram.h
#ifndef HISTOGRAM_
#define HISTOGRAM_
#include <string>
#include <vector>
struct Bin {
    double xmin, xmax, width;
    double content;
};
class Histogram {
public:
    Histogram(unsigned int NoOfBins, double xmin, double xmax);
    void Fill(double val, double weight=1.);
    void WriteToGnuplotFile(std::string filename);
    void Draw(std::string moreToDraw);
    double GetUnderflow();
    double GetOverflow();
    void Normalize(double Norm);
private:
    double overflow=0, underflow=0;
    std::vector<Bin> bins;
    double MinX, MaxX;
};
#endif

// Histogram.cxx
#include "Histogram.h"
#include <fstream>
#include <iostream>
#include <sstream>
Histogram::Histogram(unsigned int NoOfBins, double xmin, double xmax)
    : MinX(xmin), MaxX(xmax)
{
    bins.clear();
    for(unsigned int i=0; i<NoOfBins; ++i) {
        double binwidth = (xmax-xmin)/(double)NoOfBins;
        bins.push_back({xmin+binwidth*i, xmin+binwidth*(i+1), binwidth, 0});
    }
    }
void Histogram::Fill(double val, double weight) {
    if(weight<0) {
        std::cerr << "Warning: Negative weight was not accounted!\n";
        return;
    }
    if(val<MinX) {
        underflow+=weight;
    } else if(val>MaxX) {
        overflow+=weight;
    } else {
        int l=0, r=bins.size(), m=(l+r)/2;
        double binwidth=(MaxX-MinX)/bins.size();
        ```
```cpp
while (r > l + 1) {
    if (val < MinX + m * binwidth)
        r = m;
    else
        l = (l + r) / 2;
} 

bins[m].content += weight;

void Histogram::WriteToGnuplotFile(std::string filename) {
    try {
        std::ofstream ofs(filename.c_str());
        for (auto p : bins)
            ofs << 0.5*(p.xmin+p.xmax) << " " << p.content << "\n";
    } catch (const std::ofstream::failure& e) {
        std::cerr << "Error: Could not open " << filename << std::endl;
    }
}

void Histogram::Draw(std::string moreToDraw) {
    double sum=0.;
    for (auto p : bins)
        sum += p.content;
    if (sum < 1.e-14) {
        std::cerr << "Error: Cannot plot empty histogram!\n";
        return;
    }

    std::string fname = "CrypticFilenameForGnuplot";
    WriteToGnuplotFile(fname);
    std::ostringstream cmd;
    cmd << "gnuplot -p -e "set xrange [" << MinX << ":" << MaxX << "];
<< "set style fill solid 1.0;"
<< "plot " " " << fname << " u 1:2 with boxes title" "
<< moreToDraw
<< "\n";
    system(cmd.str().c_str());
    system((std::string("rm ")+fname).c_str());
}

double Histogram::GetUnderflow() {
    return underflow;
}

double Histogram::GetOverflow() {
    return overflow;
}

void Histogram::Normalize(double Norm) {
    double sum=0.;
    for (auto p : bins)
        sum += p.content*p.width;
    if (sum < 1.e-14) {
        std::cerr << "Error: Cannot normalize empty histogram!\n";
        return;
    }
    for (auto &p : bins)
        p.content*=(Norm/sum);
}

// main.cxx
#include "Histogram.h"
#include <random>

int main() {
    std::mt19937 gen(1234);
    std::normal_distribution<> d(1,1.5);
    Histogram h(100,-4.,6.);
    for(int i=0; i<100000; ++i)
        h.Fill(d(gen));
    h.Normalize(1.);
    h.WriteToGnuplotFile("test.txt");
    h.Draw("* exp(-0.5*(((x-1.)/1.5)**2)/sqrt(2*pi)/1.5 lw 3 lc 3 t ");
```
Remarks for Refactoring

I wanted to put all this nice stuff into STL algorithms, such that it looks "nicer". Some hints were given already while posing this task above. The content of main.cxx stayed the same. And so should also the interface, i.e. Histogram.h, if designed reasonably. Well, at least the public interface should behave like this.

But I noticed soon that I took a wrong way, and I had to revise some of my design decisions. That's specifically for Histogram::Fill the case, but effect the whole Histogram class. I would like to sketch the way of the development in this refactoring I went, in the hope you can learn from my errors.

1. STL Containers and Algorithms

I stripped all includes and include protections to focus on the essential changes. MinX and MaxX where removed, because they can be reconstructed from the first and last bin.

```cpp
// Histogram.h
struct Bin {
    double xmin, xmax, width;
    double content;
};
class Histogram {
public:
    Histogram(unsigned int NoOfBins, double xmin, double xmax);
    void Fill(double val, double weight=1.0);
    void WriteToGnuplotFile(std::string filename) const;
    void Draw(std::string moreToDraw="") const;
    double GetUnderflow() const;
    double GetOverflow() const;
    void Normalize(double Norm);
    double GetNorm() const;
private:
    std::vector<Bin> bins{0};
};

// Histogram.cxx
...
Histogram::Histogram(unsigned int NoOfBins, double xmin, double xmax) {
    bins.clear();
    bins.resize((NoOfBins+2);
    double binwidth = (xmax-xmin)/(double)NoOfBins;
    *std::begin(bins) = Bin(std::numeric_limits<double>::lowest(), xmin, 0, 0);
    std::generate(++std::begin(bins), --std::end(bins),
             [][i=0,&xmin,&binwidth] ()mutable {
                return Bin{xmin+binwidth*i, xmin+binwidth*(++i), binwidth, 0};
             });
    *(--std::end(bins)) = Bin{xmax, std::numeric_limits<double>::max(), 0, 0};
}
void Histogram::Fill(double val, double weight) {
    if(weight<0) {
        std::cerr << "Warning: Negative weight was not accounted!\n";
        return;
    }
    // O(N)
    // std::find_if(std::begin(bins), std::end(bins),
    //     [&val](const Bin & b){ return b.xmin<=val && b.xmax>val; })->content += weight;
    // O(log N) // <-- very clever!!!
    std::lower_bound(std::begin(bins), std::end(bins), Bin{val,0,0,0},
             [](const Bin & a, const Bin & b){return a.xmin<b.xmin; })
             ->content += weight;
```
E.2. SOLUTION: STATISTICS

void Histogram::WriteToGnuplotFile(const string& filename) const {
    try {
        ofstream ofs(filename.c_str());
        for_each(++std::begin(bins), --std::end(bins),
            [ofs] (const Bin& b) {
                ofs << 0.5*(b.xmin+b.xmax) << " " << b.content << "\n";
            });
    } catch (const std::ofstream::failure& e) {
        cerr << "Error: Could not open " << filename << endl;
    }
}

void Histogram::Draw(const string& moreToDraw) const {
    if (GetNorm() < 1.e-14) {
        cerr << "Error: Cannot plot empty histogram!\n";
        return;
    }
    string fname = "CrypticFilenameForGnuplot";
    WriteToGnuplotFile(fname);
    stringstream cmd;
    cmd << "gnuplot -p -e "set xrange [" << (*std::begin(bins)).xmax << ":" << (*--std::end(bins))).xmin << "];" << " set style fill solid 1.0; plot ";" << fname << " u 1:2 with boxes title "" << moreToDraw << "\n";
    system(cmd.str().c_str());
    system(("rm "+fname).c_str());
}

double Histogram::GetUnderflow() const {
    return (*std::begin(bins)).content;
}

double Histogram::GetOverflow() const {
    return (*--std::end(bins)).content;
}

void Histogram::Normalize(double Norm) {
    double sum = GetNorm();
    if (sum < 1.e-14) {
        cerr << "Error: Cannot normalize empty histogram!\n";
        return;
    }
    Norm/=sum;
    for_each(++std::begin(bins), --std::end(bins),
        [Norm] (Bin& b) { b.content*=Norm; });
}

double Histogram::GetNorm() const {
    return std::accumulate(++std::begin(bins), --std::end(bins) , 0.0,
        [](double s, const Bin& b) { return s+b.content*b.width; });
}

I guess everybody agrees that this is not really much more comprehensible! It still works as wished. But the code is really tough to read – just because of all the algorithms. So, what did I do wrong?

2. Take out what's not needed! The first thing I noticed was that abstracting a bin as a separate structure was sort of overkill. So, I got rid of it.

// Histogram.h
class Histogram {
public:
    Histogram(unsigned int NoOfBins, double xmin, double xmax);
    void Fill(double val, double weight=1.0);
    void WriteToGnuplotFile(const string& filename) const;
    void Draw(const string& moreToDraw="") const;
    double GetUnderflow() const;
    double GetOverflow() const;
    void Normalize(double Norm);
    double GetNorm() const;
};
private:
    std::vector<double> lims {};
    std::vector<double> bins {};
    double binwidth;
};

In the same course, I had to create another array for the bin limits. And, for some reason, I wanted to
store a binwidth. Just a moment!! Why that?! Okay, obviously, histograms can have bins of various
bin widths. But that’s not what I used all the time. I only needed a single fixed bin width! This makes
things much simpler!! But that’s a very important point! The specification of the behavior of your
software should be decided before starting to implement!!! That’s of course an ideal, which is
difficult to achieve. But you can figure out this during refactoring.

After that step, I also had to change the Histogram’s member function.

    Histogram::Histogram(unsigned int NoOfBins, double xmin, double xmax) {
        binwidth = (xmax-xmin)/(double)NoOfBins;
        lims.resize(NoOfBins+3);
        lims.front() = std::numeric_limits<double>::lowest();
        std::generate(++lims.begin(), --lims.end(),
                     [&, n=0] () mutable { return xmin+binwidth*n++; });
        lims.back() = std::numeric_limits<double>::max();
        bins.resize(NoOfBins+2, 0.);
    }

    void Histogram::Fill(double val, double weight) {
        if(weight<0) {
            std::cerr << "Warning: Negative weight was not accounted!\n";
            return;
        }
        auto n = std::distance(lims.begin(),
                               std::lower_bound(std::begin(lims),std::end(lims),val));
        bins[--n] += weight;
    }

    void Histogram::WriteToGnuplotFile(std::string filename) const {
        try {
            std::ofstream ofs(filename.c_str());
            for(int i=1, n=lims.size()-1; i<n; ++i)
                ofs << 0.5*(lims[i+1]+lims[i]) << " " << bins[i] << "\n";
            catch (const std::ofstream::failure& e) {
                std::cerr << "Error: Could not open " << filename << std::endl;
            }
        }
    }

    void Histogram::Draw(std::string moreToDraw) const {
        if(GetNorm() < 1.e-14) {
            std::cerr << "Error: Cannot plot empty histogram!\n";
            return;
        }
        std::string fname = "CrypticFilenameForGnuplot";
        WriteToGnuplotFile(fname);
        std::stringstream cmd;
        cmd << "gnuplot -p -e "set xrange [" << lims[1] << " :" << lims[lims.size()-2] << "];"
            "set style fill solid 1.0;"
            "plot " << fname << " u 1:2 with boxes title " << moreToDraw << "\";
        system(cmd.str().c_str());
        system((std::string("rm ")+fname).c_str());
    }

    double Histogram::GetUnderflow() const {
        return bins[0];
    }

    double Histogram::GetOverflow() const {
        return bins[bins.size()-1];
    }

    void Histogram::Normalize(double Norm) {
        double sum = GetNorm();
    }
Indeed! This looks much simpler now! All the member functions are much shorter.

3. Take OUT ALL what’s NOT needed! But still, it felt quite clumsy to see that I had to use
\texttt{std::distance} and \texttt{std::lower\_bound} to fill the histogram. Furthermore, this was just to find
the index of the bin which should be filled. And if something still looks complicated enough to be
incomprehensible, you should think about whether you formulated your problem correctly. All the STL
algorithm business told me here that something is wrong! I hope you can feel similar objections. The
data structures were still too complicated – now were I could assume the bins being all of the same
width.

And eventually, it hit me like a flash! Why to search for something you can simply calculate then?!
So, a last change made almost everything rather trivial.

Again, I changed somewhat the data structures. The \texttt{std::vector<double>} \texttt{bins} just carries the
bin content. \texttt{xmin\_} is the minimum value of the histogram. And \texttt{bw\_} the constant bin width, from
which I can calculate easily all the bin boundaries up to MaxX.
E.2.4 Solution: Physics: MC Toy Model for Energy Straggling

The largest part is rather simple. Only the random energy loss \( \pi(T) \sim \frac{1}{T^2} \) for \( T \in [I, 0.2(E-I) + I] \) might be tricky. One only needs to consider the following.

\[
\int_I^T \frac{1}{T^2} \, dT = \frac{1}{T} - \frac{1}{I} = \frac{P}{N},
\]
where $N$ is the normalization, and $P$ is the probability for energy loss less than $T$. Surely, if $T = I$ (minimum), $P = 0$. On the other hand, $P = 1$ for $T = T_{\text{max}}$. Therefore,

$$\frac{1}{I} - \frac{1}{T_{\text{max}}} = \frac{1}{N} \Rightarrow N = \frac{IT_{\text{max}}}{T_{\text{max}} - I}.$$ 

For all other values of $P \in [0, 1]$, we get

$$T = \frac{I}{1 - (1 - I/T_{\text{max}})P}.$$ 

If we now uniformly create $P \in [0, 1]$, we would by this formula create $T \in [I, T_{\text{max}}]$ with distribution $1/T^2$.

The rest is rather straightforward. We use here CERN ROOT for the histogram – any other tool will do the same.

We also have fitted one of the ROOT’s predefined functions – a Landau distribution function, as that function is approximately describing the energy loss distribution. Figure E.2 shows the result.

### E.2.5 Solution: Bayes MCMC Parameter Fitting

**Posterior** $\lambda^{N-1} e^{-\lambda N \bar{x}}$

```cpp
#include "TH1D.h"
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <random>
#include <vector>
struct MyExpRNG {
    MyExpRNG(double lambda, int seed=1234) : rng(seed), ed{lambda} {}
};
```
APPENDIX E. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 2

![Energy straggling: Energy loss distribution function (and Landau fit).](image)

Figure E.2: Energy straggling: Energy loss distribution function (and Landau fit).

do double getRan() {
    return ed(rng);
}
private:
    std::mt19937 rng;
    std::exponential_distribution<> ed;
};
using fptr = std::function<double(double)>;
struct MCMC {
    MCMC(fptr f, double initval, double step, int seed=1234)
    : _f(f), _x(initval), _dx(step), rng(seed), nd(0,1), ud(0,1) {};
    void setInit(double x) {_x=x;}
    void setStep(double dx) {_dx=dx;}
    double getNextRan() {
        double tmp;
        for(int i=0; i<50; ++i) {
            tmp = _dx*nd(rng);
            if(ud(rng)<_f(tmp)/_f(_x)) _x=tmp;
        }
        return _x;
    }
private:
    fptr _f;
    double _x, _dx;
    std::mt19937 rng;
    std::normal_distribution<> nd;
    std::uniform_real_distribution<double> ud;
};

int main() {
    MyExpRNG erng {0.2};
    std::vector<double> v;
    std::generate_n(std::back_inserter(v),50,[&erng](){return erng.getRan();});
    double xmean = std::accumulate(std::begin(v),std::end(v),0.)/v.size();
    TH1D ht("ht","ht",100,0,20.);
    for(int i=0; i<1000000; ++i)
        ht.Fill(erng.getRan());
    ht.SaveAs("root5.C");
    std::cout << " Xmean=" << xmean << " Lambda"=" << (v.size()-1.)/(v.size()*xmean) << "\n";
    // C++ 14 for capture initialization feature
    std::function<double(double)> f = [N=v.size(), xmean](double x){
        if(x<0) return 0.;
        return std::pow(x,N)*std::exp(-x*N*xmean);
    };
}
E.2. SOLUTION: STATISTICS

```cpp
MCMC mc {f, 1., 0.5};
TH1D h("h", "h", 100, 0, 1.);
double xm=0., xm2=0., xt;
const int N=100000;
for(int i=0; i<N; ++i) {
xm+=xt;
xm2+=xt*xt;
h.Fill(xt);
}

xm/=N;
xm2=xm2/N-xm*xm;
std::cout << " lambda = " << xm << " +/- " << std::sqrt(xm2) << "\n";
```

This should compile via

```
$ g++ -Wall -pedantic-errors -o main main.cxx $(root-config --cflags --libs) -std=c++14
```

There are several simpler possibilities to realize this exercise. However, we wanted to show a solution that is feasible already with the means available (except maybe for std::function from the header functional\(^1\)).

But let’s go through step by step. We have two classes (structs) – MyExpRNG to generate the exponentially distributed random numbers, and MCMC to accomplish the random walk for a given function. MyExpRNG should be easy to comprehend. It has a c’tor that initializes the class member data (the RNG and the distribution), and one member function getNextRan(), which supplies another random number on demand. We use this class to initialize the sample of random data in the std::vector<double> v, for which we immediately also determine the mean (and \( \lambda \) from the maximum posterior estimate). Afterwards, we use a bit of CERN ROOT, in order to check the correctness of the distribution.\(^2\)

The MCMC is a bit more complicate (due to lack of other language features, which will be introduced in later parts). First of all, we have several class data members – a function _f, the state of the MC process _x, the local stepping distribution _dx, rng, nd, and ud. Those are initialized via the only constructor. We supply (but do not use) two member functions setInit and setStep to set _x or _dx, respectively. Finally, the most important part, the member function getNextRan(), which provides another random number from the distribution represented by _f. Here happens the MCMC process. We use a for-loop with 50 iterations to generate as independent as possible random numbers from this posterior distribution. This parameter, and _dx can be used for tuning this!

The rest of the main() function is rather straightforward. With the random numbers from the posterior, we estimate the mean and the standard deviation of the estimate. For checking, we again give out the posterior distribution as histogram.

**Additional Task:** The program is certainly still rather complicated. Do you see opportunities to simplify it (keyword: Refactoring)?

**Curve Fitting**

One can in principle take the above program and adapt it for a multidimensional MCMC random walk. For practicality, it would make sense to include also an acceptance counting vector for each of the degrees of freedom (in the fit parameter space), in order to check the correct running of the MCMC process. Also the length of the de-correlation loop (which we introduced by a fix loop of 50 iterations), could be realized in a bit more adaptive way by calculating the so-called auto-correlation function, and use the length after which this auto-correlation reasonably decreased.

One note finally: Why fitting with MCMC? There are some parameters, which are hard to catch via

\(^1\)This is to circumvent the limitations of pure function pointers, which do not admit capture of Lambda functions easily!

\(^2\)For ROOT, see exercise above! Here, another feature of ROOT is shown – one can save e.g. histograms as ROOT script files (ending with .C). You can simply run them by root -l root5.C, for instance.
maximization. This might include positional or power parameters, for instance, to which a distribution is very sensitive (see, for instance, Generalized Extreme Value Distribution (Wiki)). Automatic fitting can be very difficult. Furthermore, there is almost nothing that prevents the posterior from possessing several local maxima. With the MCMC process, and mean value calculation, one obtains usually a much better control over the fitting process – also in multiple dimensions – and if done correctly, a better statement about parameters of interest (global maximum versus mean)!

E.3 Solution: Algebra

E.3.1 Solution: Square Matrix Class

```cpp
#include <array>
#include <iostream>
#include <string>

const int NN = 3;

class Matrix {
  using MatForm = std::array<std::array<double,NN>,NN>;

  public:
    Matrix() {}
    Matrix(MatForm a) : d(a) {}
    Matrix(const Matrix& a) : d(a.d) {}
    Matrix& add(const Matrix& Q) {
      for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j)
          d[i][j] += Q.d[i][j];
      return *this;
    }
    double getIJ(int i, int j) const { return d[i][j]; }
    void setIJ(int i, int j, double val) { d[i][j]=val; }
    Matrix& transpose() {
      for(int i=0; i<NN; ++i)
        for(int j=0; j<i; ++j)
          std::swap(d[i][j],d[j][i]);
      return *this;
    }
    Matrix& mul(double val) {
      for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j)
          d[i][j]*=val;
      return *this;
    }

  private:
    MatForm d;
};

Matrix add(const Matrix P, const Matrix& Q) {
  return P.add(Q);
}
Matrix transpose(const Matrix Q) {
  return Q.transpose();
}
Matrix mul(const Matrix Q, double val) {
  return Q.mul(val);
}
Matrix mul(double val, const Matrix& Q) {
  return mul(Q, val);
}
Matrix mul(const Matrix& P, const Matrix& Q) {
  Matrix R;
  for(int i=0; i<NN; ++i)
    for(int j=0; j<NN; ++j)
      double res=0.;
    for(int k=0; k<NN; ++k)
      res+=P.getIJ(i,k)*Q.getIJ(k,j);
    R.setIJ(i,j,res);
```
E.4 Solution: Geometry – Geometric Objects and Operations

Again, look carefully at this code and try to understand it! Refactoring is recommended where reasonable! Out the class declaration in a header, and the implementation in a .cxx file! Make a CMake project out of it! Add tests where they fit!

Scrutinize, what are the return values of the different operations! Why does it make sense to specify them in that way? And do you see that the operations are expressed by others? Is this reasonable?

E.3.2 Solution: Operations with Matrices and Vectors

This exercise is simple to realize once you understood the former one.

E.4 Solution: Geometry – Geometric Objects and Operations

We show a solution for the triangle. The other geometric objects can be handled in an analogous fashion. (And 3D can be made similarly!)
class Triangle {
public:
    Triangle(Point const & p1, Point const & p2, Point const & p3)
        : _p1(p1), _p2(p2), _p3(p3) {}
    Triangle& rotate_deg(double angle) {
        Point cmp {getCenterOfMass()};
        Point p1 {subtract(_p1, cmp)};
        Point p2 {subtract(_p2, cmp)};
        Point p3 {subtract(_p3, cmp)};
        angle*=pi/180.;
        double ca = std::cos(angle);
        double sa = std::sin(angle);
        double px, py;
        px = ca*p1.getX()+sa*p1.getY();
        py = -sa*p1.getX()+ca*p1.getY();
        p1.setX(px), p1.setY(py);
        _p1 = add(p1,cmp);
        px = ca*p2.getX()+sa*p2.getY();
        py = -sa*p2.getX()+ca*p2.getY();
        p2.setX(px), p2.setY(py);
        _p2 = add(p2,cmp);
        px = ca*p3.getX()+sa*p3.getY();
        py = -sa*p3.getX()+ca*p3.getY();
        p3.setX(px), p3.setY(py);
        _p3 = add(p3,cmp);
        return *this;
    }
    Point getCenterOfMass() {
        Point cmp {add(_p1, add(_p2,_p3))};
        cmp.setX(cmp.getX()/3.);
        cmp.setY(cmp.getY()/3.);
        return cmp;
    }
    Triangle& move(Point const & p) {
        _p1.add(p);
        _p2.add(p);
        _p3.add(p);
        return *this;
    }
    std::string toString() {
        return std::string {"Triangle: P1 "+_p1.toString() +", P2 "+_p2.toString() +", P3 "+_p3.toString();
    }
private:
    Point _p1, _p2, _p3;
};

int main() {
    Triangle tr {Point{0,0},Point{1,0},Point{0,1}};
    std::cout << tr.toString() << `\n';
    std::cout << `CM point: " << tr.getCenterOfMass().toString() << `\n';
    tr.move(Point{2,2});
    std::cout << tr.toString() << `\n';
    std::cout << `CM point: " << tr.getCenterOfMass().toString() << `\n';
    tr.rotate_deg(90.);
    std::cout << tr.toString() << `\n';
    std::cout << `CM point: " << tr.getCenterOfMass().toString() << `\n';
}

First introducing Point makes certainly sense, because handling single points is much easier. And all operations on more complex objects like triangles can be lead back to operations of points. If you do not believe, you can add a scaling function to triangle, which uniformly scales the triangle larger (with the center of mass point in the center).

We introduced several functions like add and subtract – both, as class members and as free functions. Please, think about this, and why it makes sense! Please, consider also the return value of
these functions! Why and when a Point or a Point&, respectively? The same also for the operations of the Triangle. Does this make sense? Discuss with your neighbor! And if the result is inconclusive, ask the instructor!

Why did we use toString() instead of directly printing?

Finally: Try to refactor this code! What would you do differently, what rename, and why? (And better, just do it!) Don’t worry if you can’t find anything to refactor. This means you understand the code as it is. For different programmers, the code might look differently in this respect.

You may have noticed the underscore on the Point’s class members, _x and _y. You still will see this habit, which came into fashion to disambiguate variable names – in the early times of C++. Nowadays, with namespaces and this pointers, that’s really not necessary anymore. The Point class can be written as follows.

```cpp
class Point {
public:
    Point(double x, double y) : x{x}, y{y} {}
    Point(const Point& p) : x{p.getX()}, y{p.getY()} {}
    double getX() const { return this->x; }
    double getY() const { return this->y; }
    void setX(double x) { this->x = x; }
    void setY(double y) { this->y = y; }
    Point& add(Point const& p) {
        this->x += p.x,
        this->y += p.y;
        return *this;
    }
    Point& subtract(Point const& p) {
        this->x -= p.x,
        this->y -= p.y;
        return *this;
    }
    std::string toString() {
        return std::string{
            "(" + std::to_string(this->x) + "," + std::to_string(this->y) + ")"};
    }
private:
    double x,y;
};
```

### E.5 Solution: STL Containers and Algorithms

```cpp
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <numeric>
#include <random>
#include <string>
#include <vector>

enum Gender { male=true, female=false }

struct Person {
    std::string firstName;
    std::string lastName;
    int age;
    Gender sex;
    std::string profession;
};

void print_person(const Person& p) {
    using namespace std;
    cout << setw(12) << p.lastName << "," << setw(10) << p.firstName
        << " (" << p.age << "," << (p.sex ? "m" : "f")
        << ") : " << p.profession << "\n";
}
```

---

3This is of course already a prelude to operator overloading in the next part!
void print(const std::vector<Person>& v) {
    for (const auto& p : v) print_person(p);
}

void print_help() {
    using namespace std;
    cout << "a) Random Shuffle\n";
    cout << "b) Sort First Name\n";
    cout << "c) Sort Last Name\n";
    cout << "d) Sort Age\n";
    cout << "e) Stable Sort Profession\n";
    cout << "f) Print male\n";
    cout << "g) Find the oldest person\n";
    cout << "h) Find the third youngest person\n";
    cout << "i) Partition into group with last names [A-M]\n";
    cout << "j) Stable Partition into group with last names [A-M]\n";
    cout << "k) Check for at least one king\n";
    cout << "l) Shift all persons a place to right\n";
    cout << "m) Print all persons with an age in [30,40]\n";
    cout << "n) Sum of all ages\n";
    cout << "p) Print DB\n";
    cout << "Enter a character (Ctrl+D to stop) : ";
}

using pvec = std::vector<Person>;

int main() {
    using namespace std;
    pvec p;
    p.push_back(Person{"Eddard", "Stark", 60, male, "Lord of Winterfell"});
    p.push_back(Person{"Robert", "Baratheon", 55, male, "King"});
    p.push_back(Person{"Jaime", "Lannister", 49, male, "Chief of Kingsguard"});
    p.push_back(Person{"Cersei", "Lannister", 46, female, "Queen"});
    p.push_back(Person{"Daenerys", "Targaryen", 32, female, "Queen"});
    p.push_back(Person{"Jon", "Snow", 32, male, "Black Guard"});
    p.push_back(Person{"Sansa", "Stark", 23, female, "Lady"});
    p.push_back(Person{"Arya", "Stark", 22, female, "Killer"});
    p.push_back(Person{"Bran", "Stark", 20, male, "Three-eyed Raven"});
    p.push_back(Person{"Joffrey", "Baratheon", 27, male, "King"});
    p.push_back(Person{"Tyrion", "Lannister", 50, male, "Consultant"});
    p.push_back(Person{"Catelyn", "Stark", 56, female, "Lady"});
    p.push_back(Person{"Jorah", "Mormont", 58, male, "Knight"});
    p.push_back(Person{"Sandor", "Frey", 50, male, "Knight"});
    p.push_back(Person{"Samwell", "Tarly", 31, male, "Consultant"});
    std::mt19937 g(1234);
    char ch = '\0';
    do {
        switch (ch) {
        case 'a': shuffle(begin(p), end(p), g); print(p); break;
        case 'b': sort(begin(p), end(p), [](const Person& a, const Person& b){
                    return a.firstName<b.firstName;}); print(p); break;
        case 'c': sort(begin(p), end(p), [](const Person& a, const Person& b){
                    return a.lastName<b.lastName;}); print(p); break;
        case 'd': sort(begin(p), end(p), [](const Person& a, const Person& b){
                    return a.age<b.age;}); print(p); break;
        case 'e': stable_sort(begin(p), end(p), 
                    [](const Person& a, const Person& b){
                    return a.profession<b.profession;}); print(p); break;
        case 'f': for_each(begin(p), end(p), [](const Person& a){
                    if (a.sex) print_person(a); }); break;
        case 'g': print_person(*max_element(begin(p), end(p), 
                    [](const Person& a, const Person& b){
                    return a.age<b.age;})); break;
        case 'h': nth_element(begin(p), begin(p)+2, end(p), 
                    [](const Person& a, const Person& b){
                    return a.age<b.age;}); break;
        case 'i': partition(begin(p), end(p), 
                    [](const Person& a){
                    return a.birthYear>=2000;}); break;
        case 'j': stable_sort(begin(p), end(p), 
                    [](const Person& a, const Person& b){
                    return a.profession<b.profession;}); break;
        case 'k': check_for_at_least_one_king(p); break;
        case 'l': shift_all_persons_right(p); break;
        case 'm': print_all_persons_with_age_in_range(p, 30, 40); break;
        case 'n': sum_of_all_ages(p); break;
        case 'p': print_DB(p); break;
        default: cout << "Enter a character (Ctrl+D to stop) : ";
        } break;
    } while (ch != '
');
}
return a.lastName[0] <= 'M'; }) ); print(p); break;

case 'j' : stable_partition(begin(p),end(p),[](const Person& a){
    return a.lastName[0] <= 'M'; }); print(p); break;

case 'k' : if(any_of(begin(p),end(p),[](const Person& a){
    return a.profession == "King"; }))
    cout << " We've a king!
"; break;

case 'l' : rotate(begin(p),begin(p)+1,end(p)); print(p); break;

case 'm' : for_each(begin(p),end(p),[](const Person& a){
    if(a.age>=30&&a.age<=40) print_person(a); }); print(p); break;

case 'n' : cout << " Total Age : " << accumulate(begin(p),end(p),0,
    [] (double s, const Person& a){return s+=a.age;})
    << '
';break;

case 'p' : print(p);
}

print_help();
while(cin >> ch);
cout << "\n";
APPENDIX E. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 2
Appendix F

Proposals for Solutions for the Hands-On Exercises of Part 3

F.1 Solution: Visualization

F.1.1 Solution: Image Class – PPM File Format

Let's start with the Color class. As it is actually rather simple, we decide for just one header `Color.h`.

```cpp
#ifndef COLOR_
#define COLOR_
class Color {
    using uchar = unsigned char;
public:
    Color(uchar red=0, uchar green=0, uchar blue=0)
        : r{red}, g{green}, b{blue} {};
    void setColor(uchar red, uchar green, uchar blue) {
        r = red, g = green, b = blue;
    };
    uchar getRed() const { return r; };
    uchar getGreen() const { return g; };
    uchar getBlue() const { return b; };
private:
    uchar r, g, b;
};
#endif
```

A constructor, and setters and getters – nothing special here.

The next class is the XYimage class, which shall take our image data. Let's have a look on `XYimage.h`.

```cpp
#ifndef XYIMAGE_
#define XYIMAGE_
#include <memory>
class Color;
class XYimage {
    using uint = unsigned int;
public:
    XYimage(uint DimensionX, uint DimensionY);
    void setPixelRGB(uint ix, uint iy, const Color& c);
    void writeToFile(const char* filename);
private:
    uint dimX, dimY;
    std::unique_ptr<unsigned char []> color;
};
#endif
```

We have include the header memory to use the std::unique_ptr smart pointer. The sense of it will
become clearer later in the course. A `std::vector` could be taken here as well – but would require later some more intrusive casting when writing the binary data to the PPM file. A constructor, a setter, and the write-to-file class member function.

The implementation file `XYimage.cxx` does also not reveal much new stuff.

```cpp
#include "XYimage.h"
#include "Color.h"
#include <iostream>
#include <fstream>

XYimage::XYimage(uint DimensionX, uint DimensionY)
    : dimX(DimensionX), dimY(DimensionY),
     color(std::unique_ptr<unsigned char[]>(new unsigned char[3*dimX*dimY])) {}

void XYimage::setPixelRGB(uint ix, uint iy, const Color& c) {
    color[(ix+dimX*iy)*3] = c.getRed();
    color[(ix+dimX*iy)*3+1] = c.getGreen();
    color[(ix+dimX*iy)*3+2] = c.getBlue();
}

void XYimage::writeToFile(const char* filename) {
    std::ofstream file(filename);
    if(!file) {
        std::cerr << "Could not open file " << filename << "!
        return;
    }
    file << "P6
" << dimX << " " << dimY << "255
";
    file.write(reinterpret_cast<const char*>(color.get()),3*dimX*dimY);
}
```

What remains is the main program, `main.cxx`.

```cpp
#include "XYimage.h"
#include "Color.h"

int main() {
    const uint dimX = 800, dimY = 800;
    XYimage XYi(dimX,dimY);
    Color c;
    for(uint ix=0; ix<dimX; ++ix)
        for(uint iy=0; iy<dimY; ++iy){
            c.setColor((255.*ix/dimX),
            (255.*iy/dimY),
            (255.*(ix+iy)/(dimX+dimY)));
            XYi.setPixelRGB(ix, iy, c);
        }
    XYi.writeToFile("test.ppm");
}
```

Also this is pretty straightforward. We create a `XYimage`. We create a `Color` object – for performance reasons outside of the loop, such that we don’t need to created it in each loop iteration. And the loops themselves pass just over the pixel array. A bit tricky is here not to fall into the trap to perform integer operation e.g. in `ix/dimX`, because this would be zero. As we have to scale anyway to 255, we simply add a decimal point, such that `255.` is a `double`. The rest of the operations then, too. The result is then correctly cast into unsigned char, as required by `Color::setColor`. `XYi.setPixelRGB(ix, iy, c)` finally sets the color to the pixel with the coordinate `ix, iy`. Finally, to write into a file is rather simply accomplished.

The result is shown in Figure F.1.

F.1.2 Solution: Image Class – Color Map, Mandelbrot Set

We rely here on the `Color` and `XYimage` classes from above – an example, how such class design increases the reusability!

Let us start with the Mandelbrot class in a single header `Mandelbrot.h`.

```cpp
#ifndef MANDELBROT_
#define MANDELBROT_
#include <complex>
```
F.1. SOLUTION: VISUALIZATION

Figure F.1: The simple gradient plot in a PPM file.

```cpp
class Mandelbrot {
    using cplx = std::complex<double>;

public:
    Mandelbrot(double MinX, double MaxX, double MinY, double MaxY, uint DimX, uint DimY, uint NumIter)
        : xmin{MinX}, ymin{MinY}, dx{(MaxX-MinX)/DimX}, dy{(MaxY-MinY)/DimY}, NUM_ITER{(double)NumIter} {}

doctor operator()(uint ix, uint iy) {
    cplx z {0}, c {(xmin+ix*dx,ymin+iy*dy)};
    for(uint i=0; i<NUM_ITER; ++i) {
        z = z*z + c;
        if (std::abs(z)>2.)
            return i/NUM_ITER;
    }
    return -1;
}

private:
    double xmin,ymin,dx,dy, NUM_ITER;
};
```

We designed it as a function object, i.e. operator() is overloaded, which takes the pixel coordinates. The ColorScheme, which maps double numbers onto Colors, is also a function object. Here, the content of ColorScheme.h.

```cpp
#ifndef COLORSCHEME_
#define COLORSCHEME_ #include "Color.h" #include <cmath>
class ColorScheme {
public:
    ColorScheme(double red, double green, double blue, double sigma) 
        : r(red), g(green), b(blue), s(sigma) {}
    Color operator()(double cindex) {
        if(cindex<0 || cindex>1.)
            return Color(0);
As you can see, we use some Gaussian functions with a mean and a standard deviation. This function was made private, because it is needed only inside of the class. When we had written this with header and implementation file separately, the Gaussian would have become a free function in the implementation file (invisible from outside), and had vanished from the class interface altogether. Making it private, we made it at least inaccessible.

Finally, the `main.cxx` – putting everything together.

```cpp
#include "ColorScheme.h"
#include "XYimage.h"
#include "Mandelbrot.h"

int main() {
    const uint dimX = 800, dimY = 800;
    ColorScheme pc(1.,2./3.,1./3.,1./3.);
    Mandelbrot mb(-1.7,0.5,-1.1,1.1,dimX,dimY, 30);
    XYimage XYi(dimX,dimY);
    for(uint ix=0; ix<dimX; ++ix)
        for(uint iy=0; iy<dimY; ++iy)
            XYi.setPixelRGB(ix, iy, pc(mb(ix, iy)));
    XYi.writeToFile("mandelbrot.ppm");
}
```

Small and comprehensible as it should be! The result is shown in Figure F.2.
F.1.3 Solution: Paraview File I/O

A Solution of the posed Task

Our MyArray class shall have a constructor, an element access operator overload (we misuse here [], because [] at most at most one variable (see cppreference: Array subscript operator)), and the writeToVTK member function. Here is MyArray.

```cpp
#ifndef MYARRAY_
#define MYARRAY_
#include <algorithm>
#include <array>
#include <string>
#include <vector>
struct MyArray {
    MyArray(std::array<int, 3> dim, std::array<double, 3> spacing, 
             std::array<double, 3> origin, std::string FieldName) 
        : n(dim), dx(spacing), org(origin), va(n[0]*n[1]*n[2]), 
          fieldname(FieldName) {} 
    double& operator[](int i, int j, int k) { 
        return va[i+n[0]*(j+n[1]*k)]; }
    double operator[](int i, int j, int k) const { 
        return va[i+n[0]*(j+n[1]*k)]; }
    void writeToVTK(std::string filename); 
private: 
    std::array<int, 3> n; 
    std::array<double, 3> dx, org; 
    std::vector<double> va; 
    std::string fieldname; 
};
#endif
```

The constructor is shorter when placed into header. The overloaded operator (if feasible to be done as is here the case) inside of the header is OK. It allows the compiler to inline it, and thus optimize its execution and the access to the vector elements.

The content of MyArray.cpp is straightforward.

```cpp
#include "MyArray.h"
#include <fstream>
void SwapEndianness(double& var) { 
    char* varArray = reinterpret_cast<char*>(&var); 
    for(long i = 0; i < static_cast<long>(sizeof(var)/2); i++) 
        std::swap(varArray[sizeof(var) - 1 - i], varArray[i]);
}
void MyArray::writeToVTK(std::string filename) { 
    std::ofstream ofs(filename); 
    ofs << "# vtk DataFile Version 2.0\n" << "Some Test File\n\n" << "DATASET STRUCTURED_POINTS\n" << "DIMENSIONS " << n[0] << " " << n[1] << " " << n[2] << " \n" << "SPACING " << dx[0] << " " << dx[1] << " " << dx[2] << " \n" << "ORIGIN " << org[0] << " " << org[1] << " " << org[2] << " \n" << "POINT_DATA " << n[0]*n[1]*n[2] << " \n" << "SCALARS " << fieldname << " double 1\n" << "LOOKUP_TABLE default\n"; 
    for(int z=0; z<n[2]; ++z) 
        for(int y=0; y<n[1]; ++y) 
            for(int x=0; x<n[0]; ++x) { 
                double val = (*this)(x,y,z); 
                SwapEndianness(val); 
                ofs.write(reinterpret_cast<char*>(&val), sizeof(double));
            }
    ofs << "\n"; 
    ofs.close();
}
```
writeToVTK has exactly one purpose! Good!

Finally, inside main, we only use this MyArray. This is the convenience and simplicity we should strive for!

```cpp
#include "MyArray.h"
#include <cmath>
#define N 41
define dx 0.1

double f(int i, int j, int k) {
    double x = (i/(double)(N-1)-0.5)*dx;
    double y = (j/(double)(N-1)-0.5)*dx;
    double z = (k/(double)(N-1)-0.5)*dx;
    return x*x*std::sin(50.*y)*std::exp(10*z);
}

ing main() {
    double ldx = -((N-1)/2.)*dx;
    MyArray ar {{N,N,N},{dx,dx,dx},{ldx,ldx,ldx}, "some_data"};
    for(int z=0; z<N; ++z) {
        for(int y=0; y<N; ++y) {
            for(int x=0; x<N; ++x) {
                ar(x,y,z) = f(x,y,z);
            }
        }
    }
    ar.writeToVTK("test.vtk");
}
```

Extra: Time Series Data in Paraview

Paraview allows the analysis of time-variable data. We show here the "legacy" way, which still works. Please, have a look into the more modern "XML" way, if interested!

We actually modify only the main.cxx from above somewhat. Furthermore, we create now 2D data – for illustration. But the time sequencing can also be done with 3D data.

The time-stepping is accomplished by creating another file for each time step. The filename must only contain a number (sequence). See e.g. under Animating legacy VTK file series for the possibilities how to do this!

```cpp
#include "MyArray.h"
#include <cmath>
#define N 41
define dx 0.1

double f(int i, int j, int k, double t) {
    double x = (i/(double)(N-1)-0.5)*dx;
    double y = (j/(double)(N-1)-0.5)*dx;
    double z = (k/(double)(N-1)-0.5)*dx;
    return x*x*std::sin(50.*(y-t))*std::exp(10*z);
}

ing main() {
    double ldx = -((N-1)/2.)*dx;
    MyArray ar {{N,N,1},{dx,dx,dx},{ldx,ldx,0}, "some_data"};
    for(int t=0; t<N; ++t) {
        for(int y=0; y<N; ++y) {
            for(int x=0; x<N; ++x) {
                ar(x,y,0) = f(x,y,0,5.e-3*t);
            }
        }
    }
    ar.writeToVTK(std::string("test_"+std::to_string(t)+".vtk");
}
```

When you open the file test_...vtk (paraview shows it so in the file-open dialog), paraview will open all the files belonging to this time series. It will open it in Slice view, which you can switch to Surface in order to get a nicely smooth, interpolated image. Furthermore, you will notice that paraview opens 2D data in 2D mode (the shown plane cannot be rotated). You can switch to 3D whenever you want/need.

The motion starts, once you press the green "Play" button at the top of the GUI (in the VCR Controls).

Last Remark: Paraview not only allows for scalar data like pressure or temperature. One can also
use vector data (velocity, electromagnetic fields), or even tensor data (tension). If you want, try to create data for such objects via C++, and analyze them with Paraview!

F.2 Solution: Algebra

F.2.1 Solution: Operator Overloading for Matrices and Vectors

We use the matrix class of the previous part, where we already added operations as class member and free functions. But we refactor this a bit. We introduce a header Matrix.h with the following content.

```cpp
#ifndef MATRIX_
#define MATRIX_
#include <array>
#include <string>
class Matrix {
   using MatForm = std::array<std::array<double,NN>,NN>;
public:
   const int NN = 3;
   Matrix();
   Matrix(MatForm a);
   Matrix(const Matrix& a);
   double getIJ(int i, int j) const;
   void setIJ(int i, int j, double val);
   Matrix& add(Matrix const& Q);
   Matrix& mul(double val);
   Matrix& transpose();
private:
   MatForm d;};
Matrix add(Matrix P, Matrix const& Q);
Matrix transpose(Matrix Q);
Matrix mul(Matrix Q, double val);
Matrix mul(double val, Matrix const& Q);
Matrix mul(Matrix const& P, Matrix const& Q);
std::string display(Matrix const& Q);
#endif
```

The implementation looks like this Matrix.cxx.

```cpp
#include "Matrix.h"
Matrix::Matrix() {}
Matrix::Matrix(MatForm a) : d(a) {}
Matrix::Matrix(const Matrix& a) : d(a.d) {}
double Matrix::getIJ(int i, int j) const { return d[i][j]; }
void Matrix::setIJ(int i, int j, double val) { d[i][j]=val; }
Matrix& Matrix::add(Matrix const& Q) {
   for(int i=0; i<NN; ++i)
      for(int j=0; j<NN; ++j)
         d[i][j] += Q.d[i][j];
   return *this;
}
Matrix& Matrix::mul(double val) {
   for(int i=0; i<NN; ++i)
      for(int j=0; j<NN; ++j)
         d[i][j]*=val;
   return *this;
}
Matrix& Matrix::transpose() {
   for(int i=0; i<NN; ++i)
      for(int j=0; j<i; ++j)
         std::swap(d[i][j],d[j][i]);
   return *this;
}
Matrix add(Matrix P, Matrix const& Q) {return P.add(Q);}
Matrix transpose(Matrix Q) {return Q.transpose();}
```
312APPENDIX F. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 3
Matrix mul(Matrix Q, double val) {return Q.mul(val);}
Matrix mul(double val, Matrix const& Q) {return mul(Q, val);}
Matrix mul(Matrix const& P, Matrix const& Q) {
Matrix R;
for(int i=0; i<NN; ++i)
for(int j=0; j<NN; ++j) {
double res=0.;
for(int k=0; k<NN; ++k)
res+=P.getIJ(i,k)*Q.getIJ(k,j);
R.setIJ(i,j,res);;
}
return R;
}
std::string display(Matrix const& Q) {
std::string s;
for(int i=0; i<NN; ++i) {
for(int j=0; j<NN; ++j)
s+=std::to_string(Q.getIJ(i,j))+", ";
s+="\n";
}
return s;
}

Finally, the main.cxx reduces to the following.
#include "Matrix.h"
#include <iostream>
int main() {
Matrix M {{{{1,2,3},
{4,5,6},
{7,8,9}}}};
std::cout << "M = \n" << display(M) << "\n";
Matrix P = transpose(M);
std::cout << "P = \n" << display(P) << "\n";
Matrix S = mul(0.5,add(M,P));
std::cout << "S = P+M =\n" << display(S) << "\n";
Matrix Q = mul(M,P);
std::cout << "Q = M*P =\n" << display(Q) << "\n";
}

Actually, everything is still the same. We only split the code into different translation units, and created
a header as interface.
First, we replace add by operator+= when a class member, and by operator+ when a free function.
Accordingly is mul replaced by operator*= and operator*, respectively. The following restructurings
are maybe a bit more intricate. Please, take your time to go through them. First, we create also
a matrix-matrix multiplication in-place (operator*=(Matrix const& Q) as class member function).
The free function is then expressed by this in-place operation. Furthermore, semantically, we assume
that the free operations (non-class members) create new objects, while the class member operators
change the current object.
We also put the const int N=3; as static variable into the class, such that it is not exposed
anymore to the outside. This should be considered good object oriented programming, as a property
(data) is encapsulated into the class!
We also change the display function, and use operator<<. Let’s look onto the result. The
Matrix.h looks as follows.
#ifndef MATRIX_
#define MATRIX_
#include <array>
#include <fstream>
class Matrix {
static const int NN = 3;
using MatForm = std::array<std::array<double,NN>,NN>;
public:
Matrix() = default;
Matrix(MatForm a);


The non-member operators return the object by value, i.e. they are copied. This is correct as they acquire an own individuality in this way. Also, often we give the parameters by value. Here, when it fits, use that the object is copied and is an own entity. Returning it by value does not incur a performance penalty (often), as the compiler can do the so-called Return Value Optimization (RVO).

The Matrix.hxx changes accordingly.

```cpp
#include "Matrix.h"
Matrix::Matrix(MatForm a) : d(a) {}
void Matrix::setIJ(int i, int j, double val)
Matrix& Matrix::operator+=(Matrix const& Q)
Matrix& Matrix::operator*=(Matrix const& Q)
Matrix& Matrix::operator*=(double val)
Matrix& Matrix::transpose()
int Matrix::getDim() const
private:
MatForm d;
};
Matrix operator+(Matrix P, Matrix const& Q)
Matrix transpose(Matrix Q)
Matrix operator*(Matrix Q, double val)
Matrix operator*(double val, Matrix Q)
Matrix operator*(Matrix P, Matrix const& Q)
std::ostream& operator<<(std::ostream& ofs, Matrix const& Q)
```
APPENDIX F. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 3

```cpp
for(int i=0; i<Q.getDim(); ++i) {
    for(int j=0; j<Q.getDim(); ++j) {
        ofs << Q.getIJ(i,j) << " ", ";
        ofs << 'n';
    }
    return ofs;
}
```

And so does `main.cxx`.

```cpp
#include "Matrix.h"
#include <iostream>

int main() {
    Matrix M {{{{1,2,3},
        {4,5,6},
        {7,8,9}}}};
    std::cout << "M = " << M << "n";
    Matrix P = transpose(M);
    std::cout << "P = " << P << "n";
    Matrix S = 0.5*(M+P);
    std::cout << "S = " << S << "n";
    Matrix Q = M*P;
    std::cout << "Q = " << Q << "n";
}
```

The usage of the matrix class get that way much more intuitive.

So, now we would like to add a vector, which has its own algebra (add, subtracting, dot product, ...), but which shall also be used to multiply a matrix with (forget here co- and contra-variance). This bears several immediate issues. Vector will be an own class. So, we put it into a separate header file. But there are operations (operators) that bind the vector to the properties of a matrix. This would mean that we put everything into one header! But if we need only the vector part, we would "load" still also the matrix part.

Even more crucial is that the vector must be of a certain size to be multiplied by a matrix. Currently, we have no mean to enforce consistency here! We will handle this in the following parts!

Here, we decide for a single header file variant. Consistency between matrix and vector is achieved via a global `NN` – to the prize of exposure! We can change this later. In the course, we introduce also for `Matrix` an `VecForm& operator[](int)` and remove `getIJ` and `setIJ`. What do you think, why is there also a `VecForm operator[](int) const` variant? The resulting `Matrix.h` looks as follow.

```cpp
#include <iostream>
#include <array>
#include <fstream>

const int NN = 3;
using MatForm = std::array<std::array<double,NN>,NN>;
using VecForm = std::array<double,NN>;

class Matrix {
public:
    Matrix() = default;
    Matrix(MatForm a);
    Matrix(const Matrix& a) = default;
    VecForm& operator[](int i); 
    VecForm operator[](int i) const;
    Matrix& operator+=(Matrix const& Q);
    Matrix& operator-=(Matrix const& Q);
    Matrix& operator*=(double val);
    Matrix transpose();
    int getDim() const;
private:
    MatForm d;
};
Matrix operator+(Matrix P, Matrix const& Q);
Matrix operator*(Matrix Q, double val);
```
Matrix operator*(double val, Matrix Q);
Matrix operator*(Matrix P, Matrix const& Q);
Matrix transpose(Matrix Q);
std::ostream& operator<<(std::ostream& ofs, Matrix const& Q);

// ====== Vector =====
class Vector {
public:
    Vector() = default;
    Vector(VecForm a);
    Vector(const Vector& a) = default;
    double& operator[](int i);
    double operator[](int i) const;
    void setIJ(int i, int j, double val);
    Vector& operator+=(Vector const& Q);
    double operator**=(Vector const& Q);
    Vector& operator+=(double val);
    int getDim() const;
private:
    VecForm d;
};
Vector operator+(Vector P, Vector const& Q);
Vector operator*(Vector Q, double val);
Vector operator*(double val, Vector Q);
std::ostream& operator<<(std::ostream& ofs, Vector const& Q);

// ====== Matrix-Vector =====
Vector operator*(Matrix const& Q, Vector const& w);
Vector operator*(Vector const& w, Matrix const& Q);
#endif

The final implementation file Matrix.cxx looks as follows.

#include "Matrix.h"
Matrix::Matrix(MatForm a) : d(a) {} 
VecForm& Matrix::operator[](int i) { return d[i]; } 
VecForm Matrix::operator[](int i) const { return d[i]; } 
Matrix& Matrix::operator+=(Matrix const& Q) {
    for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j)
            d[i][j] += Q.d[i][j];
    return *this;
}
Matrix& Matrix::operator**=(Matrix const& Q) {
    Matrix R;
    for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j) {
            double res=0.;
            for(int k=0; k<NN; ++k)
                res+=d[i][k]*Q.d[k][j];
            R.d[i][j] = res;
        }
    std::swap(*this,R);
    return *this;
}
Matrix& Matrix::operator+=(double val) {
    for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j)
            d[i][j]=val;
    return *this;
}
Matrix& Matrix::transpose() {
    for(int i=0; i<NN; ++i)
        for(int j=0; j<i; ++j)
            std::swap(d[i][j],d[j][i]);
    return *this;
}
int Matrix::getDim() const {return NN;}
//-------------------------------
Matrix operator+(Matrix P, Matrix const& Q) { return P+=Q; }
Matrix transpose(Matrix Q) { return Q.transpose(); }
Matrix operator*(Matrix Q, double val) { return Q*=val; }
Matrix operator*(double val, Matrix Q) { return Q*=val; }
Matrix operator*(Matrix P, Matrix const& Q) { return P*=Q; }

std::ostream& operator<<(std::ostream& ofs, Matrix const& Q) {
    for(int i=0; i<Q.getDim(); ++i) {
        for(int j=0; j<Q.getDim(); ++j)
            ofs << Q[i][j] << " ", ";
        ofs << '\n';
    }
    return ofs;
}

// ====== Vector ======
Vector::Vector(VecForm a) : d{a} {}
double& Vector::operator[](int i) { return d[i]; }
double Vector::operator[](int i) const { return d[i]; }
Vector& Vector::operator+=(Vector const& w) {
    for(int i=0; i<NN; ++i) d[i] += w.d[i];
    return *this;
}
double Vector::operator*=(Vector const& w) {
    double sum=0;
    for(int i=0; i<NN; ++i) sum+=d[i]*w.d[i];
    return sum;
}
Vector& Vector::operator*=(double val) {
    for(int i=0; i<NN; ++i) d[i] *= val;
    return *this;
}
int Vector::getDim() const { return NN; }

// ----------------------------------------
Vector operator+(Vector v, Vector const& w) { return v+=w; }
Vector operator*(Vector w, double val) { return w*=val; }
Vector operator*(double val, Vector w) { return w*=val; }
double operator*(Vector v, Vector const& w) { return v *= w; }

std::ostream& operator<<(std::ostream& ofs, Vector const& w) {
    for(int i=0; i<w.getDim(); ++i)
        ofs << w[i] << " ", ";
    return ofs;
}

// ====== Matrix-Vector ======
Vector operator*(Matrix const& Q, Vector const& w) {
    Vector r;
    for(int i=0; i<Q.getDim(); ++i) {
        r[i]=0.;
        for(int j=0; j<Q.getDim(); ++j) r[i] += Q[i][j]*w[j];
    }
    return r;
}
Vector operator*(Vector const& w, Matrix const& Q) {
    Vector r;
    for(int i=0; i<Q.getDim(); ++i) {
        r[i]=0.;
        for(int j=0; j<Q.getDim(); ++j) r[i] += w[j]*Q[i][j];
    }
    return r;
}

And finally, we use the main.cxx to test the implementation.

#include "Matrix.h"
#include <iostream>
int main() {
    Matrix M {{{{1,2,3}, {4,5,6}, {7,8,9}}}};
    std::cout << "M = \n" << M << "\n";
F.2. Solution: CERN ROOT Linear Equation Solver

In all of the above, we’ve assumed to always have fully occupied matrices. In numerical problems, we usually deal with so-called sparse matrices – such matrices have a lot more zero components than non-zero components. For such matrices, it is worth to think about a different data representation – Condensed Sparse-Matrix Representation or GSL User Guide: Sparse Matrix Storage Formats.

Root has already such a representation (see Linear Algebra in ROOT) – TMatrixDSparse. But a LU decomposition would create more or less fully occupied matrices. And is of complexity class $O(N^3)$ to $O(N^3)$. With sparse-matrix formats, multiplication with a vector can be boiled down to about $O(N)$ (dot product of vectors, if the matrix is diagonal). This is, where iterative solvers become very interesting (see, for instance, Saad: Iterative Methods for Sparse Linear System). As a simple (and probably not the most elegant) example, we solve again the equation from the task before, $\varphi'' = f$.

We use a Gauss-Seidel Method (Actually Jacobi-Gauss-Seidel). After discretization, we have the matrix equation

$$A\varphi = b.$$  

We can rearrange this according to Jacobi to

$$\varphi_i = \frac{1}{2}(\varphi_{i+1} + \varphi_{i-1} - (\Delta x)^2 f(x_i)), \quad 1 \leq i \leq N.$$  

In matrix form, this would look like a fix-point equation.

$$\varphi = B\varphi - b/2,$$

where $\varphi$ is the vector of with components $\varphi_i$, $B = A/2 - 1$ (1 is the unit matrix), and $b$ is the right-hand side (essentially $f(x)$ discretized at $x_i$). The advantage is that we need only do matrix vector multiplications (with a sparse matrix!). And in this ideal case (components of $A$ are constant), we can even avoid generating the matrix explicitly. To be more efficient, we do the update in-place (though incorporating already updated components into the update of other components) – that’s the Gauss-Seidel part.

```cpp
#include <cmath>
#include <iostream>
#include <vector>
#include <TAxis.h>
#include <TCanvas.h>
#include <TGraph.h>
define PI 3.14159265359
define N 50
int main() {
std::vector<double> v(N), x(N), rhs(N);
double dx=1./(N-1.);
std::cout << "\n";
Matrix P = transpose(M);
std::cout << "P = \n" << P << "\n";
Matrix S = 0.5*(M + P);
std::cout << "S = M+P =\n" << S << "\n";
Matrix Q = M * P;
std::cout << "Q = M*P =\n" << Q << "\n";
Vector v {{1,2,3}};
std::cout << "v = " << v << "\n";
Vector w {2*v};
std::cout << "w = " << w << "\n";
Vector x = v+w;
std::cout << "x = v+w = " << x << "\n";
std::cout << "x*v = " << x*v << "\n";
Vector y = M*v;
std::cout << "y = M*v = " << y << "\n";
double val = v*M*v;
std::cout << "v*M*v = " << val << "\n";
}
```
APPENDIX F. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 3

```cpp
for(int i=0; i<N; ++i) {  // initialization
    rhs[i]=-dx*dx*std::sin(PI*i*dx);
    v[i]=0.0;
    // v[i]=std::sin(PI*i*dx)/(PI*PI);
    x[i]=i*dx;
}
double sum=1.;
while(sum>1.e-8) {  // solution
    for(int i=1; i<N-1; ++i) v[i]=0.5*(v[i+1]+v[i-1]-rhs[i]);
    for(int i=N-2; i>0; --i) v[i]=0.5*(v[i+1]+v[i-1]-rhs[i]);
    sum = 0.;
    for(int i=1; i<N-1; ++i)
    sum+=std::fabs(v[i+1]+v[i-1]-2.*v[i]-rhs[i]);
    std::cout << " residual: " << sum << "\n";
}
}
```

For the solution, only two vectors of length \(N\) are needed here. The disadvantage with that method is that it converges slowly. It is now a business of using clever iteration techniques in order to not spoil the memory efficiency by bad convergence. Krylov Subspace methods are such possibilities. Or, good initial conditions are another possibility (Increase \(N\) a bit, 500! And uncomment the line above).

F.2.3 Solution: Function Object Algebra

Using exclusively Lambda expressions and std::function, one solution may be.

```cpp
#include <functional>
#include <iostream>
using Func = std::function<double(double)>;
Func operator+(const Func& f1, const Func& f2) {
    return [=](double x){ return f1(x)+f2(x);};
}
// -,*,/ ditto
int main() {
    Func f1 = [](double x){ return x*x;};
    Func f2 = [](double x){ return x*x*x;};
    Func f3 = f1 + f2;
    std::cout << " f1(2) = " << f1(2) << "\n";
    std::cout << " f2(2) = " << f2(2) << "\n";
    std::cout << " f3(2) = " << f3(2) << "\n";
}
```

But we cannot create compositions \(f(g(x))\), because we cannot overload operator() as free function, and also not as member of std::function. So, let’s create an own function object – essentially wrapping std::function.

```cpp
#include <functional>
#include <iostream>
using Func = std::function<double(double)>;
class F1 {
public:
    F1(const Func& fun) : f(fun) {}
    double operator()(double x) const {
        return f(x);
    }
    Func operator()(Func fun) const {
        return [this,fun](double x){ return f(fun(x)); };}
};
```
**F.3 Solution: STL Algorithms – Functors instead of Lambda Functions**

We notice: Lambda functions are function objects! As example.

```cpp
struct SortFirstName {
    bool operator()(const Person& a, const Person& b) {
        return a.firstName < b.firstName;
    }
};
```

Remark: One can now easily extend this to achieve a full algebra on a function vector space, including a scalar product, which induces a norm, and a metric, etc. The functions are now the types, we are working with (similar to the matrices above).

Remark: Another aspect is the lazy evaluation – which can also be used as a mean for performance optimization. We can compose functions first, and only later we evaluate them. This can be used together with a parser to generate so-called Abstract Syntax Trees – the beginning of a compiler.

**F.4 Solution: Numeric**

**F.4.1 Solution: Boost.odeint – ODE Solver**

The harmonic oscillator example is refined to

```cpp
#include <iostream>
#include <vector>
#include <boost/numeric/odeint.hpp>
typedef std::vector<double> state_type;
```
320

**APPENDIX F. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 3**

double gam = 0.15;
void HO(const state_type& x, state_type& dxdt, double t) {
    dxdt[0] = x[1]; // x' = v
    dxdt[1] = -x[0]-gam*x[1]; // v' = -x-gam*v
}
struct observer {
    void operator()(const state_type& x, double t) {
        using std::cout;
        cout << t ;
        for(int j=0, n=x.size(); j<n; ++j) cout << '\t' << x[j];
        cout << 'n';
    }
};
int main() {
    using boost::numeric::odeint::integrate;
    state_type x(2);
    x[0] = 1.0; // init x=1.0
    x[1] = 0.0; // init v=0.0
    // f(x) x0 t=0 t=end dt observer
    integrate( HO, x, 0.0, 35.0, 0.1, observer());
}

Instead as function, the right-hand-side can also be implemented as function object (a functor).

```cpp
struct HO {
    double gam;
    HO(double gamma) : gam(gamma) {}  
    void operator()(const state_type& x, state_type& dxdt, double t) {
        dxdt[0] = x[1]; // x' = v
        dxdt[1] = -x[0]-gam*x[1]; // v' = -x-gam*v
    }
};
integrate(HO(0.15), x, 0.0, 35.0, 0.1, observer());
```

The Lorenz-Attractor can be realized by the following.

```cpp
#include <iostream>
#include <vector>
#include <boost/numeric/odeint.hpp>
typedef std::vector<double> state_type;
struct Lorenz {
    Lorenz() {}  
    void operator()(const state_type& x, state_type& dxdt, double t) {
        dxdt[0] = 10.*(x[1]-x[0]);
        dxdt[1] = x[0]*(28.-x[2])-x[1];
        dxdt[2] = x[0]*x[1]-2.6666666*x[2];
    }
};
struct observer {
    void operator()(const state_type& x, double t) {
        using std::cout;
        cout << t ;
        for(int j=0, n=x.size(); j<n; ++j) cout << '\t' << x[j];
        cout << 'n';
    }
};
int main() {
    using boost::numeric::odeint::integrate;
    state_type x(3);
    x[0] = 1.0; // init x=1.0
    x[1] = 1.0; // init y=1.0
    x[2] = 1.0; // init z=1.0
    // f(x) x0 t=0 end dt observer
    integrate(Lorenz(), x, 0.0, 50.0, 0.1, observer());
}
```
F.4. SOLUTION: NUMERIC

Redirecting the output into a file,

```
$ ./boostODE_Lorenz > result.txt
```

we can plot the attractor using Gnuplot by

```
gnuplot> splot 'result_Lorenz.txt' u 2:3:4 w l t ''
```

resulting in the following Figure F.3.

![Lorenz Attractor](image)

**Figure F.3: Lorenz Attractor**

Instead of Gnuplot, you can also use ROOT. The script reading the data from a file, looks as follows.

```cpp
void test() {
    TCanvas *c1 = new TCanvas("c1");
    std::ifstream ifs("data.txt");
    double t, x, y, z;
    std::vector<double> vx, vy, vz;
    while (ifs >> t >> x >> y >> z) {
        vx.push_back(x);
        vy.push_back(y);
        vz.push_back(z);
    }
    TPolyLine3D *l = new TPolyLine3D(vx.size(), &vx[0], &vy[0], &vz[0]);
    double rmin[] = {*std::min_element(vx.begin(), vx.end()),
                    *std::min_element(vy.begin(), vy.end()),
                    *std::min_element(vz.begin(), vz.end())};
    double rmax[] = {*std::max_element(vx.begin(), vx.end()),
                    *std::max_element(vy.begin(), vy.end()),
                    *std::max_element(vz.begin(), vz.end())};
    TView3D *view = new TView3D(1, rmin, rmax);
    view->ShowAxis();
l->Draw();
}
```
322APPENDIX F. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 3

F.4.2 Solution: Aerodynamics and Complex Numbers

Listing for simple Flow Potentials

```cpp
const std::complex<double> J={0,1};
double const_flow(double* x, double* p) {
    // p[0] = v_0, p[1] = phi_0
    std::complex<double> z {x[0],x[1]}; // z=x+iy
    return std::imag(p[0]*exp(J*p[1])*z);
}
double source_flow(double* x, double* p) {
    // p[0] = x_0, p[1] = y_0, p[2] = k
    std::complex<double> z {x[0],x[1]}; // z=x+iy
    std::complex<double> z0 {p[0],p[1]}; // z_0=x_0+iy_0
    return std::imag(p[2]*log(z-z0));
}
double circulation_flow(double* x, double* p) {
    // p[0] = x_0, p[1] = y_0, p[2] = k
    std::complex<double> z {x[0],x[1]}; // z=x+iy
    std::complex<double> z0 {p[0],p[1]}; // z_0=x_0+iy_0
    return std::imag(J*p[2]*log(z-z0));
}
double dipole_flow(double* x, double* p) {
    // p[0] = x_0, p[1] = y_0, p[2] = Re(p_0), p[3] = Im(p_0)
    std::complex<double> z {x[0],x[1]}; // z=x+iy
    std::complex<double> z0 {p[0],p[1]}; // z_0=x_0+iy_0
    std::complex<double> p0 {p[2],p[3]}; // p_0=px_0+ipy_0
    return std::imag(p0/(z-z0));
}
int main() {
    std::vector<double> lev {-14,-12,-10,-8,-6,-4,-2,0,2,4,6,8,10,12,14};
    TCanvas *c = new TCanvas("c","c",3200,800);
    c->Divide(4,1);
    c->cd(1);
    TF2* fu = new TF2("uniform flow",const_flow,-10,10,-10,10,2);
    fu->SetParameters(1.,-45.*(TMath::Pi()/180.)); // 45 degree
    fu->SetContour(lev.size(),&lev[0]);
    fu->SetLineWidth(1);
    fu->Draw("cont3");
    c->cd(2);
    TF2* fs = new TF2("source flow",source_flow,-10,10,-10,10,3);
    fs->SetParameters(1.,0.,1.);
    fs->SetContour(20);
    fs->SetLineWidth(1);
    fs->Draw("cont3");
    c->cd(3);
    TF2* fc = new TF2("circulation flow",circulation_flow,-10,10,-10,10,3);
    fc->SetParameters(1.,0.,1.);
    fc->SetNpx(100);
    fc->SetNpy(100);
    fc->SetContour(40);
    fc->SetLineWidth(1);
    fc->Draw("cont3");
    c->cd(4);
    TF2* fd = new TF2("dipole flow",dipole_flow,-10,10,-10,10,3);
    fd->SetParameters(1.,0.,1.);
    fd->SetNpx(100);
    fd->SetNpy(100);
    fd->SetContour(100);
    fd->SetLineWidth(1);
    fd->Draw("cont3");
    c->SaveAs("flowprofiles.pdf");
}
```

lev defines the contour levels – aka streamlines.
Listing for Flow around a Cylinder (with circulation)

```cpp
const std::complex<double> J={0,1};
double complex_flow(double* x, double* p) {
    std::complex<double> z (x[0],x[1]); // z=x+iy
    return std::imag(p[0]*(exp(J*p[1])*z+p[2]/z)+J*p[3]*log(z));
}

int main() {
    TEllipse * el = new TEllipse(0,0,1);
    el->SetLineColor(1);
    el->SetLineWidth(1);
    el->SetFillColor(10);
    el->SetFillStyle(1001);
    TCanvas *c = new TCanvas("c","c",1600,800);
    c->Divide(2,1);
    c->cd(1);
    TF2* fc = new TF2("flow around a cylinder",complex_flow,-5,5,-5,5,4);
    fc->SetParameters(1.,0.,1.,0.);
    fc->SetNpx(100);
    fc->SetNpy(100);
    fc->SetLineWidth(1);
    fc->Draw("cont3");
    el->Draw("same");
    c->cd(2);
    TF2* fc2 = new TF2("flow around a cylinder with circulation",complex_flow,-5,5,-5,5,4);
    fc2->SetParameters(1.,0.,1.,1.);
    fc2->SetNpx(100);
    fc2->SetNpy(100);
    fc2->SetLineWidth(1);
    fc2->Draw("cont3");
    el->Draw("same");
    c->SaveAs("complexflow.pdf");
}
```

Listing Joukowski Transform

```cpp
#define N 100
std::complex<double> Joukowski(std::complex<double> z) {
    return 0.5*(z+1./z);
}

int main() {
    TCanvas *c = new TCanvas("c","c",1600,800);
    c->SetGridx();
    c->Divide(2,1);
    c->GetPad(1)->SetGridx();
    c->GetPad(1)->SetGridy();
    c->GetPad(2)->SetGridx();
    c->GetPad(2)->SetGridy();
    std::vector<double> x,y,xz,yz;
    std::vector<double> x1,y1,xz1,yz1;
    std::vector<double> x2,y2,xz2,yz2;
    std::vector<double> x3,y3,xz3,yz3;
    double dfi = 2*TMath::Pi()/N, r=1.;
    for(int i=0; i<N; ++i) {
        x.push_back(r*cos(dfi*i));
        y.push_back(r*sin(dfi*i));
        auto z = Joukowski((std::complex<double>{x[i],y[i]}));
        xz.push_back(real(z));
        yz.push_back(imag(z));
        x1.push_back(r*cos(dfi*i)+0.5);
        y1.push_back(r*sin(dfi*i));
        z = Joukowski((std::complex<double>{x1[i],y1[i]}));
        xz1.push_back(real(z));
        yz1.push_back(imag(z));
    }
}```
Listing for Flow around an Airplane Wing

```c++
const double v0 = 1.0; // speed
const double ang = -15.*TMath::Pi()/180.; // angle of attack
const double circ = 0.8; // circulation
const double del = 0.1; // delta
const int N = 100; // # points
const std::complex<double> J={0,1}; // imaginary unit
double complex_flow(double* x, double* p) {
    std::complex<double> z (x[0],x[1]); // z=x+iy
    std::complex<double> z0 = -p[3]+p[3];
    double r0 = 1+p[3];
    if(abs(z-z0)<r0) return 0;
    return std::imag(p[0]*(exp(j*p[1])*z+exp(-j*p[1])*r0*r0/(z-z0))
    +j*p[2]*log(z-z0));
}
```
The TPolyLine is used again to plot the wing profile.

F.5 Solution: Debugging Session

Corrected Solution

Corrected code is approximately the following.

```cpp
double airfoil_flow(double* x, double* p) {
  std::complex<double> w(x[0], x[1]);
  std::complex<double> z(w + sqrt(w * w - 1.));
  if (x[0] < 0.) z = w - sqrt(w * w - 1.);
  double zz[2] = {real(z), imag(z)};
  return complex_flow(&zz[0], p);
}

int main() {
  TEllipse* el = new TEllipse(-del, del, 1.+del);
  el->SetLineColor(1);
  el->SetLineWidth(1);
  el->SetFillColor(10);
  el->SetFillStyle(1001);
  TCanvas* c = new TCanvas("c", "c",1600,800);
  c->Divide(2,1);
  c->GetPad(1)->SetGridx();
  c->GetPad(1)->SetGridy();
  c->GetPad(2)->SetGridx();
  c->GetPad(2)->SetGridy();
  c->cd(1);
  TF2* fc = new TF2("flow around a cylinder", complex_flow, -5.5,-5,5,4);
  fc->SetParameters(v0, ang, circ, del);
  fc->SetNpx(200);
  fc->SetNpy(200);
  fc->SetLineWidth(1);
  fc->SetContour(20);
  fc->Draw("cont3");
  el->Draw("same");
  c->cd(2);
  TF2* fa = new TF2("flow around an airfoil", airfoil_flow, -2.2,-2,2,4);
  fa->SetParameters(v0, ang, circ, del);
  fa->SetNpx(500);
  fa->SetNpy(500);
  fa->SetLineWidth(1);
  fa->SetContour(50);
  fa->Draw("cont3");
  std::vector<double> vx,vy;
  double dd = 2.*TMath::Pi()/N, r0=1+del;
  for (int i=0; i<=N; ++i) {
    std::complex<double> z = (r0*cos(dd*i)-del, r0*sin(dd*i)+del);
    z=0.5*(z+1./z);
    vx.push_back(real(z));
    vy.push_back(imag(z));
  }
  TPolyLine *af = new TPolyLine(vx.size(), &vx[0], &vy[0]);
  af->SetLineWidth(1);
  af->SetLineColor(4);
  af->SetFillColor(4);
  af->SetFillStyle(1001);
  af->Draw("f");
  c->SaveAs("flowAirfoil.pdf");
}
```

// main.cxx
#include "my_sqrt.h"
#include <algorithm>
#include <cmath>
APPENDIX F. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 3

```cpp
#include <fstream>
#include <numeric> // <-- compile-time (std::iota)
#include <vector>

int main() {
    using namespace std;
    vector<double> v1(200), v2(200); // <-- run-time (seg-fault @ transform)
    generate(begin(v1), end(v1), [d = -10.] () mutable {
        return d+=0.1;
    });
    transform(begin(v1),end(v1),begin(v2),
        [](double x){
            x = sin(x)/x+my_sqrt(x); // <-- run-time (undetected: nan)
            if(isnan(x)) return 1.; // <--
            return x;
        });
    std::ofstream ofs("data.dat");
    for(unsigned int i=0; i<v1.size(); ++i) // <-- compile-time warning (-Wall)
        ofs << v1[i] << " \t " << v2[i] << "\n";
}

// my_sqrt.h
#ifndef MY_SQRT_H
#define MY_SQRT_H

double my_sqrt(double x) {
    return std::sqrt(std::fabs(x));
}
#endif
// my_sqrt.cxx
#include "my_sqrt.h"
#include <cmath>

```
F.5. SOLUTION: DEBUGGING SESSION

```
$ ulimit -c unlimited
$ ./data_gen
segmentation violation (core dumped)
$ gdb ./data_gen core
 [...] 
Core was generated by `./data_gen'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x000055fbb676c508 in __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > > std::transform<__gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > >, __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > >, main::{lambda(double)#2}>(__gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > >, __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > >, main::{lambda(double)#2}, main::{lambda(double)#2}) ()
```

First of all, q+<Enter> will leave the debugger.

We read that the program terminated with SIGSEGV (signal segmentation violation), and below at which address in the program. The rest is hard to read because of the template syntax – a little bit better might it get if you compile with the -g flag, i.e. CMAKE_BUILD_TYPE=DEBUG.

```
#0 0x000055c410a24508 in std::transform<__gnu_cxx::__normal_iterator<double*, std::vector<double> >, __gnu_cxx::__normal_iterator<double*, std::vector<double> >, main()::<lambda(double)> >(__gnu_cxx::__normal_iterator<double*, std::vector<double> >, __gnu_cxx::__normal_iterator<double*, std::vector<double> >, __gnu_cxx::__normal_iterator<double*, std::vector<double> >, <lambda(double)>) (...)
```

But reading it cautiously, you will soon recognize std::transform, which gives the indication, where the problem occurs. The last two lines above tell you that the problem is inside stl_algo.h on line 4304. That is the STL! Don’t panic! In that line 4304, you see the implementation of the loop inside transform, which you may have seen already as implementation proposal in cppreference: std::transform.

Now, you only need to think a bit, why a memory access violation might occur here. There are only iterators, and dereferencing operations (the unary operation you defined yourself). The simplest solution is always that you write to where you are not allowed to. This all memory that was not acquired by your program. And as the only place, where something is written to, is v2, we probably forgot to acquire sufficiently large memory for it. That’s also cured very fast, once we identified this issue.

No the program runs, and does not abort prematurely. But later, inside the data.dat file, we see a nan. Experienced programmers know exactly what happens here. And identify the division-by-zero. Actually, C++ does not treat this as a floating point error, but just sets the number in the state nan.

A debugger might nonetheless helpful here. So, basic GDB syntax! Start (the DEBUG version of) the program without any core file!

```
$ gdb ./data_gen
 [...] 
```

You are now in the GDB shell. You can use list or list <line-no> to let you show the line source code at this line number. You can also specify list <func name> to list a special function. help list will tell you the options, list accepts.
Via `run` or `run <prog>` you can let the program run inside GDB.\(^1\)

Instead, you can issue `start` to let the program start, and jump to the begin of `main()`. Using `step` (s)\(^2\) and `next` (n) (like `step`, but does not enter subroutines when they are called) you can now step through your program.

You can use `break my_sqrt` to set a break point at the function. Via `continue` (c), you continue normal execution (until the next break point).

Inside a function, you can get information about the variables. `info args` tells you about the function arguments and their values. `print <variable>` shows the value of a variable. Via `set <variable> = <name>` you can even set a variable value. This has real effect in the program then! Via `backtrace` (bt), you can print the function stack trace.

It requires some practice in order to work fluently with a debugger. And GDB can do a lot of more things! More often, you’ll probably be faster by using the *Hello!-debugging technique*, i.e. by looking and testing the source code itself.

A final remark. A lot of commercial software is highly optimized (inlining, static linkage, etc.) and stripped (no readable debugging). To do debugging on the level of a debugger is probably impossible! When you ask us, it is not worth the time to try! Contact the vendors user support!

---

\(^1\)Look for `file` to specify the executable, possibly. But it should be automatically loaded, when you started GDB with the program name as command-line parameter.

\(^2\)GDB knows abbreviations!
Appendix G

Proposals for Solutions for the Hands-On Exercises of Part 4

G.1 Solution: Cryptography – RSA Encryption

Choosing the Public-Private-Keys

The program to get the right values for $n, k, l$ is shown below. $q$ and $p$ must be prime. $k$ is chosen randomly. You can change the seed for the RNG to get other $k$ and $l$ values. $k$ and $l$ are so-called modular inverses. In order that an inverse exists, $k$ and $\phi$

```cpp
#include <algorithm>
#include <cmath>
#include <numeric>
#include <iostream>
#include <vector>
#include <random>

std::vector<int> ivector(100000);

void create_primes() {
    std::iota(ivector.begin(), ivector.end(), 2);
    for (auto it = ivector.begin(); it != ivector.end();)
        ivector.erase(std::remove_if(++it, ivector.end(),
            [it](int iv){ return iv%(*it)==0; }), ivector.end());
}

long gcd(long a, long b) {
    if (b==0)
        return a;
    else
        return gcd(b,a%b);
}

struct ret { long d, s, t; }

ret ext_ea(long a, long b) {
    ret tmp {ext_ea(b,a%b)};
    return ret {tmp.d, tmp.t, tmp.s-(a/b)*tmp.t};
}

int main() {
    create_primes();
    long q=10103;
    long p=17491;
    long n=p*q;
    long f=(p-1)*(q-1);
    long k=p-1;
    std::vector<int> ivector(100000);
    for(auto p : ivector)
        std::cout << p << " ";
    std::cout << "\n";
}
```
APPENDIX G. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 4

```cpp
std::mt19937 e1(18375);
std::uniform_int_distribution<long> unif(2,100000);
while(gcd(k,f)!=1)
    k=unif(e1);
ret tmp {ext_ea(k,f)};  // gcd,
long l=tmp.s;  // k*l = 1 mod f --> l = k^{-1} mod f
if(l<0) l+=f;  // l>0!
l r = (k*l) % f;  // test (should be 1)
std::cout << " p=" << p << " \n";
std::cout << " q=" << q << " \n";
std::cout << " n=" << n << " \n";
std::cout << " f=" << f << " \n";
std::cout << " k=" << k << " \n";
std::cout << " l=" << l << " \n";
std::cout << " r=" << r << " \n";
```

A preliminary Program for Testing

The best approach is to start small and step-wise. And only if each step is understood, the loops should be implemented. Several design decisions are to be made. For testing, for instance, we just hard-coded the keys. Later, of course, keys should be read from a file, or the command-line. If you want, you can add all this. The bare test code looks like the following.

```cpp
#include <fstream>
#include <iostream>
#include <random>
#include <string>

long power(long x, unsigned long y, long n) { // x^y mod n
    long long res = 1;
x%=n;
    while (y > 0) {
        if (y & 1) res=(res*x)%n;
y>>=1;
x=(x*x)%n;
    }
    return static_cast<long>(res);
}
#define NumToRead 10
int main () {
    long n = 176711573;
    unsigned long k = 72097;
    unsigned long l = 45393253;
    // --- read and encrypt
    std::ifstream ifs {"test.dat"};
    std::ofstream fs1 {"test.crypt"};
    char buf[NumToRead+1];
    buf[NumToRead]=\0; // safety margin
    long t1;
    int t2, cnt;
    do {
        ifs.read(buf, NumToRead*sizeof(char));
cnt = ifs.gcount();
        if(cnt==0) break;
        if(cnt%2!=0) buf[cnt]=\0;  
        for(int i=0; i<(cnt+cnt%2)/2; ++i) {
            t1 = reinterpret_cast<short int*>(buf)[i];
t2 = static_cast<int>(power(t1,k,n));
fs1.write(reinterpret_cast<const char*>(&t2), sizeof(t2));
        }
    } while(cnt==NumToRead);
    ifs.close();
fs1.close();
    // --- read and decrypt
    ifs.clear();
    ifs.open("test.crypt");
```
We included in the main (commented out now) also a loop to test whether the `power` function (modular exponentiation) works.

**The final Program**

To make this a reasonably useful program, we split all the tasks (command-line parsing, IO, and transforming) in different objects. For systematics, we create a project directory with the folders `src` and `include`, and a `CMakeLists.txt`.

```cmake
# CMakeLists.txt
cmake_minimum_required (VERSION 3.5)
project (rsa_encode)
set(CMAKE_CXX_FLAGS "-Wall")
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
set(CMAKE_CXX_EXTENSIONS False)
include_directories ("${PROJECT_SOURCE_DIR}/include")
add_executable (rsa_encode src/main.cxx src/CMDpar_Handler.cxx src/IO_stream.cxx)
include(CTest)
file (WRITE "$\{PROJECT_BINARY_DIR\}/Test/test.dat"
    "Rather arbitrary ASCII text with lot of stuff
    not relevant except for just this test!
    Some number 13468, # $$\" and special characters!""
)
add_test (NAME encrypt COMMAND rsa_encode -k=72097 -n=176711573
    "$\{PROJECT_BINARY_DIR\}/Test/test.dat")
add_test (NAME decrypt COMMAND rsa_encode -d -k=45393253 -n=176711573
    "$\{PROJECT_BINARY_DIR\}/Test/test.encrypt")
add_test (NAME compare COMMAND "$\{CMAKE_COMMAND\} -E compare_files
    "$\{PROJECT_BINARY_DIR\}/Test/test.dat $\{PROJECT_BINARY_DIR\}/Test/test.decrypt")
```

As you can see, we also added some simple test to check some correctness.¹

Specifically, we added `$\{PROJECT_SOURCE_DIR\}/include` as include path, and added all relevant source files in the `src` folder. The two headers are `CMDpar_Handler.h` and `CMDpar_Handler.h`

```
// CMDpar_Handler.h
ifndef CMDPAR_HANDLER_
define CMDPAR_HANDLER_
```

¹We faked somewhat, ... did you notice where?
#include <string>
class CMDpar_Handler {
public:
    CMDpar_Handler(int argc, char *argv[]);
    std::string getInFilename() const { return infilename;};
    std::string getOutFilename() const { return outfilename;};
    bool isEncrypt() const { return encrypt;};
    long getk() const {return k;}
    long getn() const {return n;}
private:
    std::string infilename, outfilename;
    bool encrypt=true;
    long k,n;
};
#endif

and IO_stream.h.

// IO_stream.h
#ifndef IO_STREAM_
#define IO_STREAM_
#include "CMDpar_Handler.h"
#include <string>
#include <vector>
class IO_stream {
public:
    IO_stream(const CMDpar_Handler& cmdh);
    ~IO_stream();
    std::vector<long>::iterator begin() { return vl.begin(); }
    std::vector<long>::iterator end() { return vl.end(); }
private:
    std::string ofsname;
    std::vector<long> vl;
    bool enc;
};
#endif

The corresponding implementation files are CMDpar_Handler.cxx
G.1. SOLUTION: CRYPTOGRAPHY – RSA ENCRYPTION

```cpp
return fname;
}

CMDpar_Handler::CMDpar_Handler(int argc, char *argv[]) {
    using namespace std;
    if(argc!=4 && argc!=5)
        throw runtime_error(show_help(file(string(argc[0]))));
    vector<string> sv;
    for(int i=1; i<argc; ++i) sv.push_back(string(argv[i]));
    sv.erase(remove_if(begin(sv),end(sv),
        [=](const string& cmp){
            encrypt=false;
            return true;
        }),end(sv));
    sv.erase(remove_if(begin(sv),end(sv),
        [=](const string& cmp){
            if(cmp == "-d"){
                encrypt=false;
                return true;
            }
            return false;
        }));,end(sv));
    if(sv.size()!=1)
        throw runtime_error(string{"[Error] Parameters not correctly specified!"});
    infilename = sv[0];
    outfilename = path(sv[0]) + base(sv[0])+(encrypt :
        "encrypt": "decrypt";)
    if(k>=n)
        throw runtime_error(string{"[Error] k>=n not permitted!"});
    if(k<0 || n<0)
        throw runtime_error(string{"[Error] k,n <0 not permitted!"});
    cout << " k=" << k << "n=
" << " n=" << n << "\n"
    << " enc=" << boolalpha << encrypt << "\n"
    << " infile=" << infilename << "\n"
    << " outfile=" << outfilename << "\n";
}
```

and `IO_stream.cxx`

```cpp
#include "IO_stream.h"
#include <fstream>
#include <exception>
#define IN_BUF_SIZE 5

IO_stream::IO_stream (const CMDpar_Handler& cmdh)
    : ofsname{cmdh.getOutFilename()}, enc{cmdh.isEncrypt( )}
{
    std::ifstream ifs{cmdh.getInFilename()};
    if (!ifs)
        throw std::runtime_error(std::string{"[Error] Could not open input file!"});
    char buf[IN_BUF_SIZE*4+1]; // 4 because of int limit
    buf[IN_BUF_SIZE*4+1]="\0"; // safety margin
    int cnt;
    do {
        ifs.read(buf,IN_BUF_SIZE*4+sizeof(char));
        cnt = ifs.gcount();
        if (cnt==0) break ;
        while (cnt%2!=0) buf[cnt++]="\0";
        if(enc)
            for(int i=0;i<cnt/2; ++i)
                vl.push_back(reinterpret_cast<short int*>(buf)[i]);
        else
            for(int i=0;i<cnt/4; ++i)
                vl.push_back(reinterpret_cast<int*>(buf)[i]);
    }
```
APPENDIX G. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 4

while (cnt==IN_BUF_SIZE*4*sizeof(char));
}
IO_stream::IO_stream() {
    std::ofstream ofs {ofname};
    if(enc)
        for(auto p : vl) {
            int t = static_cast<int>(p);
            ofs.write(reinterpret_cast<const char*>(&t),sizeof(t));
        }
    else
        for(auto p : vl) {
            short int t = static_cast<short int>(p);
            ofs.write(reinterpret_cast<const char*>(&t),sizeof(t));
        }
}

A lot of clutter is just for the convenience. Command-line option parsing, for instance, takes substantial place. In essence, we filled argv into a string vector, and reduced by STL algorithms when according options where found (remove_if). In the course, we used as side-effect the capture clause of the Lambda function, to set the CMD_Handlers variables.

The input and output of the streams (encryption is excluded – why? becomes clear in a moment), is essentially reduced to a constructor, which reads the file, and the destructor, which outputs the file with the encrypted/decrypted data. As we put all data conveniently into a std::vector, we can now use its container property, which are also forwarded by begin and end. The remaining main.cxx, i.e. the main program, looks as follows.

```cpp
#include "IO_stream.h"
#include <iostream>
int main(int argc, char* argv[]) {
    try {
        CMDpar_Handler cmdh {argc, argv};
        IO_stream io {cmdh};
        std::transform(io.begin(), io.end(), io.begin(),
            // capture initializer C++14
            [k=cmdh.getk(), n=cmdh.getn()](long x) {
                long long res = 1;
                unsigned long y = k;
                x %= n;
                while (y>0) {
                    if (y & 1) res=(res*x)%n;
                    y >>= 1;
                    x = (x*x)%n;
                }
                return static_cast<long>(res);
            });
    } catch(std::runtime_error e) {
        std::cerr << e.what() << "\n" ;
    }
}
```

As you can see, the essential part is again just a transformation. More elegant solutions are certainly conceivable.

When compiling this, you will get the following warning.

```
... src/main.cxx: In function ‘int main(int, char**)’:
src/main.cxx:12:7: warning: lambda capture initializers only available with -std=c++14 or -std=gnu++14
    [k=cmdh.getk(), n=cmdh.getn()](long x) {
    ...
```

You can switch the variable CMAKE_CXX_STANDARD inside CMakeLists.txt to 14 for C++ 14 standard, and the warning will vanish. After (successful) compilation, you can issue make test or ctest. You should see something like this
Test project build

Start 1: encrypt
1/3 Test #1: encrypt .......................... Passed 0.00 sec

Start 2: decrypt
2/3 Test #2: decrypt .......................... Passed 0.01 sec

Start 3: compare
3/3 Test #3: compare .......................... Passed 0.02 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) = 0.04 sec

fortunately, also successful.

G.1.1 Addendum: Boost Program Options

Quite a bit of the code had to be spent for the reading and interpretation of command-line options. It might delight you to experience that Boost already supplies a more convenient alternative. Instead of using the RSA en-/decoding program, we outline the application of boost program-options just by using an example.

```cpp
#include <boost/program_options.hpp>
#include <iostream>

int main(int argc, char** argv) {
    namespace po = boost::program_options;
    po::options_description desc("Allowed options");
    desc.add_options()
        ("help", po::value<int>(), "produce help message")
        ("compression", po::value<int>(), "set compression level")
    ;
    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if (vm.count("help")) {
        std::cout << desc << '\n';
        return 1;
    }
    if (vm.count("compression")) {
        std::cout << "Compression level was set to "
                   << vm["compression"].as<int>() << ",\n";
    } else {
        std::cout << "Compression level was not set.\n";
    }
}
```

One has to include also some stuff into the CMakeLists.txt.

```cmake
cmake_minimum_required(VERSION 3.5.2)
project(Example)
find_package(Boost 1.71.0 REQUIRED COMPONENTS program_options)
include_directories(${Boost_INCLUDE_DIRS})
add_executable(Example example.cxx)
target_link_libraries(Example LINK_PUBLIC ${Boost_LIBRARIES})
```

As usual, the option --help prints the list of available options. vm is a key-value pair list, such that the value is already of the desired type – here an int.
APPENDIX G. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 4

G.2 Solution: Visualization

G.2.1 Solution: Visualization – Simple 2D Drawing (Tutorial)

We guess, you manage a solution by yourself. The picture, you should obtain with the given program should look like Figure G.1

![Figure G.1: Drawing simple shapes on a XYimage (here with resolution 0.02).](image)

G.2.2 Solution: Visualization – Image Processing

The XYimage file read-in constructor looks as follows.

```cpp
XYimage(std::string filename) {
    std::ifstream ifs(filename);
    if (!ifs)
        throw std::runtime_error(std::string("Could not open file \n" + filename + "!
"));
    std::string dummy;
    ifs >> dummy;
    if (dummy != "P6")
        throw std::runtime_error(std::string("File " + filename + " is corrupted!
"));
    ifs >> dimX >> dimY >> dummy;
    std::getline(ifs, dummy);
    color = std::unique_ptr<unsigned char[]>(new unsigned char[3*dimX*dimY]);
    ifs.read(reinterpret_cast<char*>(color.get()), 3*dimX*dimY);
    ifs.close();
}
```

The corresponding copy-constructor has the following shape.

```cpp
#include <cstring>

XYimage(XYimage const& xyi) :
    dimX(xyi.dimX), dimY(xyi.dimY),
    color(std::unique_ptr<unsigned char[]>(new unsigned char[3*dimX*dimY]))
{
    std::memcpy(color.get(), xyi.color.get(), sizeof(unsigned char)*3*dimX*dimY);
}
```
For the random noise experiment, we have added `operator*=(double)` to the `Color` class, accompanied by a free `operator*(Color, double)`.

```cpp
// Color.h
#ifndef COLOR_
#define COLOR_
#include <ostream>
class Color {
    using uchar = unsigned char;
public:
    Color(uchar red=0, uchar green=0, uchar blue=0);
    void setColor(uchar red, uchar green, uchar blue);
    uchar getRed() const;
    uchar getGreen() const;
    uchar getBlue() const;
    Color& operator**(double val);
private:
    uchar r,g,b;
};
Color operator*(Color c, double val);
#endif

// Color.cxx
[...]
uchar mult(uchar c, double val) {
    val*=c;
    if(val>255.) val=255.;
    if(val<0.) val=0.;
    return static_cast<uchar>(val);
}
Color& Color::operator**(double val) {
    r = mult(r,val);
    g = mult(g,val);
    b = mult(b,val);
    return *this;
}
Color operator*(Color c, double val) {
    return c*=val;
}
```

The `main()` looks then rather unspectacular.

```cpp
#include "Color.h"
#include "XYimage.h"
#include <random>
int main() {
    std::mt19937 gen{1234};
    std::normal_distribution<> d(1.,0.2);
    XYimage ximg("LRZ-2011-08.ppm");
    for(int i=0; i<ximg.getDimX(); ++i)
        for(int j=0; j<ximg.getDimY(); ++j)
            ximg.setPixelRGB(i,j,ximg.getPixelRGB(i,j)*d(gen));
    ximg.writeToFile("test.ppm");
}
```

**Further Task:** Another image processing filter is Gaussian Blur, where a pixel’s color is convoluted with the color of neighboring pixels in such a way that the farther away (spatial distance) a pixel is, the smaller is its weight.

This is just one side of the medal. Filters doing the opposite are, of course, much more interesting. If you know how, you can also realize filters that remove noise, or deconvolute (sharpen) an image. Because you have now an original image, and a filter causing the noise/blur for mocking such effects (usually else caused by physical effects in cameras etc.), you can directly play and check your own improving filters by comparing with the original image!
G.3 Solution: Numeric – PDE Solver (Tutorial)

We have to solve

\[(1 - \Delta t Q) \psi^{t+\Delta t} = \psi^t + \Delta t NL(\psi^t),\]

For a given time-step (or initial condition), the system matrix \(Q\), and \(\Delta t\), this equation is of type

\[Ax = b,\]

where \(A\) is a square matrix (in this case even symmetric and sparse). \(b\) is the right-hand side, containing all the non-linearity of the equation, taken at the old time step – i.e. for this inner loop (within a time step), \(b\) is a given right-hand side (considered constant). And, eventually, \(x\) is the desired value for the field in the new time step – once it fulfills this equation. This equation can be solved in many different ways, as was already outlined.

LU, Cholesky, and similar other decomposition methods (single-step) were already ruled out due to the memory requirements.

Iterative methods are much more appropriate to account for the sparseness of the matrix. We set \(x_0 = \psi^t\) as initial value for the inner (within a time step) loop. If we rewrite \(A\) as

\[A = L + D + U,\]

where \(D\) is the diagonal matrix containing the diagonal elements of \(A\), and \(L\) and \(U\) are the sparse lower and upper triangular parts of \(A\) (\(L^T = U\) if \(A^T = A\)), then the Jacobi iterative method is done as follows,

\[x_{k+1} = D^{-1}(-(L + U)x_k + b).\]

The inversion of \(D\) is cheap, as it is diagonal. Written as a fixed point equation, this method converges if \(A\) is strongly diagonal-dominant, i.e.

\[|a_{ii}| > \sum_{j \neq i} |a_{ij}|.\]

So, let’s consider \(A\). It is in 2D a \(N^2 \times N^2\) matrix, each line corresponding to one point \((x_i, y_j)\), with the multi-index \(l = i \cdot N + j\). According to the stencil – which has 13 fields, see Figure G.2 – each line of \(A\) has 13 entries (when using periodic boundary conditions).

According to the colors, only four different values are there.

\[(i, j) : 1 - \Delta t \left(\varepsilon - 1 + 8/(\Delta x)^2 - 20/(\Delta x)^4\right)\]
\[(i \pm 1, j), (i, j \pm 1) : \Delta t \left(2/(\Delta x)^2 - 8/(\Delta x)^4\right)\]
\[(i \pm 1, j \pm 1) : 2\Delta t/(\Delta x)^4\]
\[(i \pm 2, j), (i, j \pm 2) : \Delta t/(\Delta x)^4\]

Strong diagonal dominance is present, if

\[\left|1 - \Delta t \left(\varepsilon - 1 + 8/(\Delta x)^2 - 20/(\Delta x)^4\right)\right| \geq 4 \frac{\Delta t}{(\Delta x)^4} \left(3 + 2(\Delta x)^2 - 4\right).\]

Using \(\Delta x = 0.5\) and \(\varepsilon = 0.1\), this would require that \(\Delta t \leq 0.00261\) is. Fortunately, the strong diagonal dominance (Gershgorin’s theorem) is just a rather loose condition, and not a strict one for a matrix to be non-singular. So, we can hope that larger values of \(\Delta t\) will also lead to convergence, i.e. there is a solution to our equation.

In the Gauss-Seidel method (which has similar convergence criteria as the Jacobi, but slightly faster convergence speed), we use already updated field values.

\[(D + U)x_{k+1} = (-Lx_k + b) \quad \text{or} \quad (L + D)x_{k+1} = (-Ux_k + b).\]
G.3. SOLUTION: NUMERIC – PDE SOL VER (TUTORIAL)

(i,j) (i-1,j) (i+1,j) (i+2,j) (i-2,j)
(i,j-1)
(i,j-2)
(i,j+1)
(i,j+2)
(i-1,j-1)
(i+1,j-1)
(i-1,j+1)
(i+1,j+1)

Figure G.2: Left: Stencil of the FD-discretized Swift-Hohenberg equation. Right: The system matrix for a $10 \times 10$ system. That’s the system matrix structure. All white fields are zeros. Periodic boundary conditions are already incorporated.

Or, even better, both in a sequence, to avoid asymmetries. As $D + U$ and $L + D$ are of triangle shape, they can be resolved using backward substitution. Practically, this is accomplished by updating the field $x$ in-place instead of on another array.

As $A$, and thus $L, D, U$ are sparse matrices, all matrix vector operations are thus of complexity $O(N)$ instead of $O(N^2)$. One should not spoil this by using non-sparse matrix storage formats. Accordingly, the matrix-vector operations (multiplication) should be implemented most efficiently. See also the Eigen library example in the next part.

**Convergence:** We start the iteration with $x_0 = \psi$, which for sure does not fulfill the matrix equation $Ax = b$. We thus introduce the residue, $r_k$, by

$$Ax_k - b = r_k .$$

By some norm, the residue should go down to zero with increasing values of $k$. Such a norm can be the maximum norm,

$$\|r_k\|_{\infty} = \max |r_k| ,$$

i.e. the component-wise maximum. Other norms are but also conceivable. For instance the $L^2$-norm,

$$\|r_k\|_2 = \sqrt{r_k \cdot r_k} = \sqrt{(Ax_k - b) \cdot (Ax_k - b)} .$$

Because this shall go to zero, which is the minimum, we can use another interesting method here – minimizing $\|r_k\|_2$ with respect to the components of $x_k$ (the field components). Methods like Steepest Descent or Conjugate Gradient can be used. If $A$ is symmetric and positive definite (spd), one can also directly minimize

$$x^T Ax - x^T b \to \min!$$

You can find information about such methods under the acronyms GMRES, BICG or BICGSTAB, etc. See also [33], specifically for so-called Krylov methods.
Boundary Conditions: Periodic boundary conditions (BC) on such a simple domain are of course a gift! Usually, your BCs are different, and presumably more complicated – constant field values on a boundary are nice. But they can effectively prevent methods like the spectral one. Also, gradients of the field quantities can be given on boundaries, which requires to calculate the boundary values from those of domain-internal points.

More complicated does it become, if the boundaries represent geometrically complicated shapes. In those cases, you probably can even not use structured meshes anymore. However, the principle way of solving PDEs, as outlined above, still works. Only the structure of the system-matrix, $A$, – even if it is still sparse – will change, and usually not be symmetric anymore. In extreme cases, $A$ can also be time dependent (see i.e. fluid-structure interaction).

This shall suffice as an introduction. Of course, a lot of libraries already exist – also for C++ projects – that can and should be used. OpenFOAM, or LibMesh.

G.4 Solution: Boost – Named Function Arguments (Tutorial)

Please, have a look into the tutorials on the boost docu: Parameter-Enabled Member Functions, and following!

G.5 Solution: Algebra – Square Matrix and Vector

Please compare the following solution with that of the previous part’s hands-on exercise of this topic!
Vector(int N);
Vector(VecForm a);
Vector(const Vector& a) = default;
double& operator[](int i);
double operator[](int i) const;
void setIJ(int i, int j, double val);
Vector& operator+=(Vector const& Q);
double operator==(Vector const& Q);
Vector& operator==(double val);
int getDim() const;
private:
VecForm d;
Vector operator+(Vector P, Vector const& Q);
Vector operator*(Vector Q, double val);
Vector operator*(double val, Vector Q);
std::ostream& operator<<(std::ostream& ofs, Vector const& Q);
// ====== Matrix-Vector =====
Vector operator*(Matrix const& Q, Vector const& w);
Vector operator*(Vector const& w, Matrix const& Q);
}
#endif
// Matrix.cxx
#include "Matrix.h"
#include <algorithm>
#include <stdexcept>
#include <string>
using VecForm = std::vector<double>;
Matrix::Matrix(int N) : MatFormBase(N), d(N) {
    std::for_each(std::begin(d), std::end(d), [N](VecForm &v) {v.resize(N);});
}
Matrix::Matrix(MatForm a) : MatFormBase((int)a.size()), d(a) {}
VecForm& Matrix::operator[](int i) { return d[i]; }
VecForm Matrix::operator[](int i) const { return d[i]; }
Matrix& Matrix::operator+=(Matrix const& Q) {
    if(NN!=Q.getDim())
        throw std::runtime_error(std::string("Error: Mat+=Mat - wrong size!");
    for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j)
            d[i][j] += Q.d[i][j];
    return *this;
}
Matrix& Matrix::operator*(Matrix const& Q) {
    if(NN!=Q.getDim())
        throw std::runtime_error(std::string("Error: Mat*=Mat - wrong size!");
    Matrix R {Q.getDim()};
    for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j) {
            double res=0.;
            for(int k=0; k<NN; ++k) res+=d[i][k]*Q.d[k][j];
            R.d[i][j] = res;
        }
    std::swap(*this, R);
    return *this;
}
Matrix& Matrix::operator*(double val) {
    for(int i=0; i<NN; ++i)
        for(int j=0; j<NN; ++j) d[i][j] *= val;
    return *this;
}
Matrix& Matrix::transpose() {
    for(int i=0; i<NN; ++i)
        for(int j=0; j<i; ++j) std::swap(d[i][j], d[j][i]);
    return *this;
}
int Matrix::getDim() const { return NN;
Matrix operator+(Matrix P, Matrix const& Q) { return P+=Q; }
Matrix transpose(Matrix Q) { return Q.transpose(); }
Matrix operator*(Matrix Q, double val) { return Q*=val; }
Matrix operator*(Matrix P, Matrix const& Q) { return P*=Q; }
std::ostream& operator<<(std::ostream& ofs, Matrix const& Q) {
    for(int i=0; i<Q.getDim(); ++i) {
        for(int j=0; j<Q.getDim(); ++j)
            ofs << Q[i][j] << " ", ;
        ofs << \n;
    }
    return ofs;
}

// ====== Vector =====
Vector::Vector(int N) : MatFormBase{N}, d(N) {}
Vector::Vector(VecForm a) : MatFormBase{((int)a.size())}, d(a) {}
double& Vector::operator[](int i) { return d[i]; }
double Vector::operator[](int i) const { return d[i]; }
Vector& Vector::operator+=(Vector const& w) {
    for(int i=0; i<N; ++i) d[i] += w.d[i];
    return *this;
}
double Vector::operator==(Vector const& w) {
    if(N!=w.getDim())
        throw std::runtime_error(std::string("Error: Vec==Vec - wrong size!"));
    double sum=0;
    for(int i=0; i<N; ++i) sum+=d[i]*w.d[i];
    return sum;
}
Vector& Vector::operator*=(double val) {
    for(int i=0; i<N; ++i) d[i] *= val;
    return *this;
}
int Vector::getDim() const { return N; }

// ----------------------------------------
Matrix operator+(Matrix P, Matrix const& Q) { return P+=Q; }
Matrix transpose(Matrix Q) { return Q.transpose(); }
Matrix operator*(Matrix Q, double val) { return Q*=val; }
Matrix operator*(Matrix P, Matrix const& Q) { return P*=Q; }
std::ostream& operator<<(std::ostream& ofs, Matrix const& Q) {
    for(int i=0; i<Q.getDim(); ++i) {
        for(int j=0; j<Q.getDim(); ++j)
            ofs << Q[i][j] << " ", ;
        ofs << \n;
    }
    return ofs;
}

// main.cxx
The `main()` function has not changed. Also the changes in `Matrix` and `Vector` are rather comprehensible. We added the inheritance for consistency. This involves checks for dimensions of matrix and vector at strategically useful places (mostly the class member operators, as the free operator overloading use the former ones), which also throw if necessary. Setup in that way, `MatFormBase` as base class ensures both, the correct size of matrix and vector when used in operations, and the flexibility during run-time. Such matrices and vectors could be initialized also during run-time by a user – even with (nearly) arbitrary size.

G.6 Solution: Statistics – CERN ROOT Fitting (Tutorial)

Here the GEV example.

```cpp
#include "Matrix.h"
#include <iostream>
int main() {
    Matrix M {{{{1,2,3},
           {4,5,6},
           {7,8,9}}}};
    std::cout << "M = \n" << M << "\n";
    Matrix P = transpose(M);
    std::cout << "P = \n" << P << "\n";
    Matrix S = 0.5*(M + P);
    std::cout << "S = P+M =\n" << S << "\n";
    Matrix Q = M * P;
    std::cout << "Q = M*P =\n" << Q << "\n";
    Vector v {{1,2,3}};
    std::cout << "v = \n" << v << "\n";
    Vector w {2*v};
    std::cout << "w = \n" << w << "\n";
    Vector x = v+w;
    std::cout << "x = \n" << x << "\n";
    Vector y = M*v;
    std::cout << "y = \n" << y << "\n";
    double val = v*M*v;
    std::cout << "v*M*v = \n" << val << "\n";
}
```
In this case, luckily, we know that the mean and the variance of the Gumbel distribution ($\xi = 0$) exists, and is related to the position parameter $\mu$ and the scale parameter $\sigma$. Mean and variance are estimated by the sample estimators (arithmetic mean, essentially).

![Figure G.3: Fitted GEV distribution from the example.](image)

Figure G.3 shows the result of this fit.

### G.7 Solution: Optimization Session (Tutorial)

The realization of a lookup table is as follows.

```cpp
// SHG.h -------------------------------------------- -------------------------
#include <vector>
class SHG : public Equation {
public:
    SHG(double epsilon, double A, double deltax, int size);
    ~SHG();
    double Link(std::vector<double> const & x, int i, int j=0, int k=0) const override;

private:
    [...]}
};

// SHG.cxx ------------------------------------------ -------------------------
SHG::SHG(double epsilon, double A, double deltax, int size)
    : Equation(deltax, size), eps(epsilon), a(A)
{
using std::cos;
const double dx = Equation::getDX();
const double Mk = 2.*M_PI/size;
k2.resize(size*(size/2+1));
k4.resize(size*(size/2+1));
for(int i=0; i<size/2+1; ++i)
    for(int j=0; j<size; ++j) {
```
\[
\begin{align*}
k_2[i+j*(\text{size}/2+1)] &= 2\cdot(2.-\cos(Mk*i)-\cos(Mk*j))/(dx*dx); \\
k_4[i+j*(\text{size}/2+1)] &= 2\cdot(\cos(2*Mk*i)+\cos(2*Mk*j)+4.\cdot\cos(Mk*i)\cdot\cos(Mk*j)) \\
&\quad-8.\cdot(\cos(Mk*i)+\cos(Mk*j))+10.)/(dx*dx*dx*dx);
\end{align*}
\]

The rest stays the same. And indeed, we can reduce the self time from 4.55 seconds to 1.72 seconds.\(^2\) Now, SpecIm::solve becomes the dominant hot-spot. In that way, one can try to optimize more and more. But be warned that it becomes more and more difficult to really make speed-ups. This should be the moment to think about other ways like parallelization. Because otherwise you might lose yourself in more and more small scale optimization, which rarely helps, and often clutters the code until it cannot be maintained anymore.

\[\text{bool } \_\_\text{gnu_cxx}\_\_\text{operator!=} \text{ is the comparison operator for iterators. Everywhere, where you call for(auto& p : v), such iterators become active. The optimization flags -O2 or -O3 make the many bool \_\_\text{gnu_cxx}\_\_\text{operator!=} calls vanish. The deeper reason is probably that during optimization, they become inlined (that is the function overhead vanishes).}\]

\(^2\)On my desktop. On other machines, you may see different results of the optimization!
APPENDIX G. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 4
Appendix H

Proposals for Solutions for the Hands-On Exercises of Part 5

H.1 Solution: Visualization – Simple Ray Tracing

We give only the final step – also for users that are maybe only interested in the extension (e.g. create a cube shape or a plane (inheriting from Shape), and look how it works). Of course, keen programmers can add also transparency, i.e. when rays can pass through objects!

```cpp
// Color.h --------------------------------------------------------------
#ifndef COLOR_
#define COLOR_
#include <ostream>
class Color {
    using uchar = unsigned char;
public:
    Color(uchar red=0, uchar green=0, uchar blue=0);
    void setColor(uchar red, uchar green, uchar blue);
    uchar getRed() const;
    uchar getGreen() const;
    uchar getBlue() const;
    Color operator*(double v);
    Color operator*(Color const& c);
private:
    uchar r,g,b;
};
Color operator*(Color c, double v);
std::ostream& operator<<(std::ostream& os, Color const& c);
#endif
// Color.cxx --------------------------------------------------------------
#include "Color.h"
#include <algorithm>
#include <cmath>
using uchar = unsigned char;
Color::Color(uchar red, uchar green, uchar blue)
    : r(red), g(green), b(blue) {}
void Color::setColor(uchar red, uchar green, uchar blue) {
    r = red, g = green, b = blue;
}
uchar Color::getRed() const { return r; }
uchar Color::getGreen() const { return g; }
uchar Color::getBlue() const { return b; }
uchar mult(uchar f, double v) {
    double p = std::fabs(v)*f;
    return static_cast<uchar>(std::min(p,255.));
}
Color& Color::operator*(double v) {
```
r = mult(r,v);
g = mult(g,v);
b = mult(b,v);
return *this;
}
Color& Color::operator+=(Color const& c) {
  r=std::min(r+c.r,255);
g=std::min(g+c.g,255);
b=std::min(b+c.b,255);
return *this;
}
Color operator*(Color c, double v) {
  return c*=v;
}
std::ostream& operator<<(std::ostream& os, Color const& c) {
  os << ColRGB(" << static_cast<int>(c.getRed())
    << "," << static_cast<int>(c.getGreen())
    << "," << static_cast<int>(c.getBlue()) << ");
  return os;
}
H.1. SOLUTION: VISUALIZATION – SIMPLE RAY TRACING

```cpp
this->x += p.x, this->y += p.y, this->z += p.z;
return *this;
}
Point& Point::operator-=(Point const& p) {
  this->x -= p.x, this->y -= p.y, this->z -= p.z;
  return *this;
}
Point& Point::operator*(double v) {
  this->x *= v, this->y *= v, this->z *= v;
  return *this;
}
Point& Point::operator/=(double v) {
  this->x /= v, this->y /= v;
  return *this;
}
Point Point::perp(const Point& p) {
  Point u (*this);
  return p - u*((p*u)/(u*u));
}
double Point::length() const {
  return sqrt(x*x+y*y+z*z);
}
Point operator+(Point p1, Point const& p2) {
  return p1+=p2;
}
Point operator-(Point p1, Point const& p2) {
  return p1-=p2;
}
Point operator*(Point p1, double v) {
  return p1*=v;
}
Point operator/(Point p1, double v) {
  return p1/=v;
}
double operator*(Point const& p1, Point const& p2) {
  return p1.getX()*p2.getX()+p1.getY()*p2.getY()+p1.getZ()*p2.getZ();
}
Point operator^(Point const& p1, Point const& p2) {
  return Point{ p1.getY()*p2.getZ()-p1.getZ()*p2.getY(),
                p1.getZ()*p2.getX()-p1.getX()*p2.getZ(),
                p1.getX()*p2.getY()-p1.getY()*p2.getX()};
}
std::ostream& operator<<(std::ostream& os, Point const& p) {
  os << "(" << p.getX() << "," << p.getY() << "," << p.getZ() << ")";
  return os;
}
double distance(Point const& p1, Point const& p2) {
  return (p2-p1).length();
}
```

// XYimage.h ---------------------------------------- ----------------------
#ifndef XYIMAGE_
#define XYIMAGE_
#include <memory>
#include <string>
class Color;
class XYimage {
  using uint = unsigned int;
public:
  XYimage(uint DimensionX, uint DimensionY);
  void setPixelRGB(uint ix, uint iy, const Color& c);
  void writeToFile(std::string filename);
private:
  uint dimX, dimY;
  std::unique_ptr<unsigned char []> color;
};
#endif
APPENDIX H. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 5

// XYimage.cxx -------------------------------------- ----------------------
#include "XYimage.h"
#include "Color.h"
#include <iostream>
#include <fstream>

XYimage::XYimage(uint DimensionX, uint DimensionY)
: dimX{DimensionX}, dimY{DimensionY},
  color{std::unique_ptr<unsigned char[]>(new unsigned char[3*dimX*dimY])}
{}

void XYimage::setPixelRGB(uint ix, uint iy, const Color& c) {
  color[(ix+dimX*iy)*3 ] = c.getRed();
  color[(ix+dimX*iy)*3+1] = c.getGreen();
  color[(ix+dimX*iy)*3+2] = c.getBlue();
}

void XYimage::writeToFile(std::string filename) {
  std::ofstream file(filename);
  if(!file) {
    std::cerr << "Could not open file " << filename << "!" << std::endl;
    return;
  }
  /* for binary write */
  // file << "P6\n" << dimX << " " << dimY << "255\n";
  // file.write(reinterpret_cast<const char*>(color.get ()),3*dimX*dimX);
  /* for ASCII write */
  file << "P3\n" << dimX << " " << dimY << "255\n";
  for(int i=0; i<3*dimX*dimY; ++i)
    file << (int)color.get()[i] << " ";
}

// Shape.h ------------------------------------------ ----------------------
#ifndef SHAPE_
#define SHAPE_
#include "Point.h"
#include "Color.h"
#include "Ray.h"
class Shape {
public:
  Shape(Color color, double Reflectivity = 0.2);
  virtual ~Shape();
  virtual void setColor(Color const& color);
  virtual Color getColor();
  virtual bool Hit(Ray const& r) const = 0;
  virtual Point getClosestPoint(Ray const& r) const = 0;
  virtual Vector getNormal(Ray const& r) const = 0;
  double getGetReflectivity();
private:
  Color c;
  double reflty;
};
#endif

// Shape.cxx ---------------------------------------- ----------------------
#include "Shape.h"
#include <algorithm>
using namespace std;
Shape::Shape(Color color, double Reflectivity = 0.2){
  c{color}, reflty{max(min(Reflectivity,1.),0.)} }
Shape::~Shape() {}
void Shape::setColor(Color const& color) { c=color; }
Color Shape::getColor() { return c; }
double Shape::getGetReflectivity() { return reflty; }

// Sphere.h ----------------------------------------- ----------------------
#ifndef SPHERE_
#define SPHERE_
#include "Shape.h"
class Sphere : public Shape {
H.1. SOLUTION: VISUALIZATION – SIMPLE RAY TRACING

Point cen;
double rad;

public:
    Sphere(Point Center, double Radius, Color col, double Reflectivity=0.2);
    bool Hit(Ray const& r) const override;
    Point getClosestPoint(Ray const& r) const override;
    Vector getNormal(Ray const& r) const override;
};
#endif

#include "Sphere.h"
#include <cmath>
Sphere::Sphere(Point Center, double Radius, Color col, double Reflectivity):
    Shape{col,Reflectivity}, cen{Center}, rad{Radius} {}

bool Sphere::Hit(Ray const& r) const {
    double dist = ((cen-r.getPosition())ˆr.getDirection()).length() /
    r.getDirection().length();

    if(dist < rad) return true;
    return false;
}

Point Sphere::getClosestPoint(Ray const& r) const {
    using rte = std::runtime_error;
    Point a {r.getPosition()};
    Point b {r.getDirection()};
    double p = b*(cen-a)/(b*b);
    double q = ((cen-a)*(cen-a)-rad*rad)/(b*b);
    if(p*p-q<0.)
        throw rte("Error: Tried to get point on Sphere not hit by ray!
    double t = p-std::sqrt(p*p-q);
    return a+b*t;
}

Vector Sphere::getNormal(Ray const& r) const {
    Point a = getClosestPoint(r);
    return (a-cen)*=(1./sqrt((a-cen)*(a-cen)));
}

class Ray {
    Point orig;
    Vector dir;

public:
    Ray(Point origin, Vector direction);
    Point const& getPosition() const;
    Vector const& getDirection() const;
};

class Camera {
    Vector focalVec, upVec;
APPENDIX H. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 5

Point pos;
double res;
double width, height;
public:
Camera(Point Position = Point(0,0,0),
double resolution = 0.0008,
Vector FocalDirection = Vector(1,0,0),
Vector UpwardsDirection = Vector(0,0,1),
double WindowWidth = 1., double WindowHeight = 0.7);
void RenderToFile(Scene const& scene, std::string const& filename) const;
};
#endif
// Camera.cxx --------------------------------------- ----------------------
#include "Camera.h"
#include "XYimage.h"
#include "Ray.h"
#include <cmath>
#include <exception>
using namespace std;
Camera::Camera(Point Position,
double resolution,
Vector FocalDirection, Vector UpwardsDirection,
double WindowWidth,
double WindowHeight)
: focalVec{FocalDirection}, upVec{FocalDirection.perp(UpwardsDirection)},
pos{Position}, res{resolution}, width{WindowWidth}, height{WindowHeight} {
if(fabs(upVec*upVec)<1.e-13)
throw runtime_error(std::string("Error: Upward Vector Zero!\n"));
}
void Camera::RenderToFile(Scene const& scene, string const& filename) const {
unsigned int dimX = static_cast<unsigned int>((width)/res);
unsigned int dimY = static_cast<unsigned int>((height)/res);
Vector vx = focalVec/(focalVec*focalVec);
Vector vz = upVec/(upVec*upVec);
Vector vy = vx^vz;
XYimage XYi {dimX,dimY};
for(unsigned int ix=0; ix<dimX; ++ix)
for(unsigned int iy=0; iy<dimY; ++iy) {
Ray r {pos, focalVec + (vy*(res*(ix-0.5*dimX)))
+ (vz*(res*(0.5*dimY-iy)))};
XYi.setPixelRGB(ix, iy, scene.RenderRayColor(r));
}
XYi.writeToFile(filename);
}

// Scene.h ------------------------------------------ ----------------------
#ifndef SCENE_
#define SCENE_
#include "Shape.h"
#include "Ray.h"
#include <vector>
class Scene {
std::vector<Shape*> scene;
Color bgcol; // background color
Vector litDir; // light direction
Color litcol; // light color
Shape* findClosestObject(Ray const& r, Shape *s = nullptr) const;
public:
Scene(Color BackgroundColor, Vector LightDirection,
Color LightColor = Color(255,255,255));
~Scene();
void AddShape(Shape* s);
std::vector<Shape*> const& GetShapes() const;
Color RenderRayColor(Ray const& r, double depth=0) const;
};
#endif
// Scene.hxx .................................................................
#include "Scene.h"
#include <algorithm>
#include <cmath>

Scene::Scene(Color BackgroundColor, Vector LightDirection,
              Color LightColor)
: bgcol{BackgroundColor},
  litDir{LightDirection*(1./std::sqrt(LightDirection*LightDirection))},
  litcol{LightColor}
{}

Scene::~Scene() {
  for(auto p : scene) delete p;
  scene.clear();
}

void Scene::AddShape(Shape* s) {
  scene.push_back(s);
}

std::vector<Shape*> const & Scene::GetShapes() const {
  return scene;
}

double directed_distance(Ray const& r, Point const& p) {
  return (p-r.getPosition())*r.getDirection();
}

double Gaus(double x) {
  return std::exp(-0.5*x*x);
}

#define MAX_RAY_DEPTH 5

Color Scene::RenderRayColor(Ray const& r, double depth) const {
  Color c {0};
  if(depth>MAX_RAY_DEPTH) return c;
  Shape* sh = findClosestObject(r);
  if(sh==nullptr) return bgcol;
  Point hp = sh->getClosestPoint(r);    // hit point
  Vector np = sh->getNormal(r);        // normal
  double cosf = -(np*litDir);          // normal*light
  Ray r1 {hp,litDir*(-1.)};            // ray hit point -> light dir
  Shape* sh2 = findClosestObject(r1,sh); // object that shadows?
  double cosd = -(np*r.getDirection());  // normal*inray
  Vector ref = np*(2.*cosd) + r.getDirection();  // reflected ray
  double reflc = sh->getReflectivity();  // reflection coefficient
  if(cosf<0. && ( sh2==nullptr
                  && directed_distance(r1,sh2->getClosestPoint(r1))<0.0 )) {
    c += (sh->getColor()*cosf);        // direct surface reflection
  }
  Ray r2 {hp,ref};                   // ray (hit point -> reflect dir
  Color cr = RenderRayColor(r2, ++depth); // reflect ray render
  c += (1.-reflc);                   // direct surface reflection
  c += (cr*reflc);
  if(cosf>0. && ( sh2==nullptr
                  && directed_distance(r1,sh2->getClosestPoint(r1))<0.0 )) {
    double cosrf = -(ref*litDir);    // reflect*light
    c += litcol*(Gaus((cosrf-1.)/(0.005*(1.-reflc)))*(5.*reflc)); // head light
  }
  return c;
}

Shape* Scene::findClosestObject(Ray const& r, Shape *s) const {
  Shape *shp = nullptr;
  double dist = INFINITY, tmpdist;
  for(auto p : scene) {
    if(p!=s && p->Hit(r)) {
      Point objP = p->getClosestPoint(r);
      tmpdist = directed_distance(r,objP);
      if(tmpdist<0. && tmpdist<dist) {
        dist=tmpdist;
        shp=p;
      }
    }
  }
  return shp;
}
APPENDIX H. PROPOSALS FOR SOLUTIONS FOR THE HANDS-ON EXERCISES OF PART 5

```c++
return sthp;
}

// main.cxx ----------------------------------------- ----------------------
#include "Camera.h"
#include "Sphere.h"
#include "Scene.h"
int main() {
  Color white = Color{255,255,255};
  Color red = Color{255,0,0};
  Color green = Color{0,255,0};
  Color blue = Color{0,0,255};
  Color magenta = Color{255,0,255};
  Scene sc {Color{20,10,50},Vector{1,1,-1}};
  sc.AddShape(new Sphere{Point{7,-1,-1},1,green,0.05});
  sc.AddShape(new Sphere{Point{10,0,0},3,red});
  sc.AddShape(new Sphere{Point{5,1,1},1,blue,0.6});
  sc.AddShape(new Sphere{Point{7.5,-2.7,2.},0.7,Color{200,200,155}});
  sc.AddShape(new Sphere{Point{8.5,-3.5,0},1.5,magenta});
  sc.AddShape(new Sphere{Point{0,0,-1.e6},1.e6-4.,white,0.001});
  Camera ca = Camera {Point{-1.5,0,0}};
  ca.RenderToFile(sc,"test.ppm");
}
```

And the CMakeLists.txt.

```cmake
cmake_minimum_required (VERSION 3.5)
project (RayTracer)
include_directories("${PROJECT_SOURCE_DIR}/include")
add_executable (raytracer src/main.cxx src/XYimage.cxx src/Color.cxx
src/Camera.cxx src/Point.cxx src/Scene.cxx
src/Shape.cxx src/Sphere.cxx src/Ray.cxx)
```

Disclaimer: This solution is certainly not the only or best one! You are invited to provide better ones! Btw. before you try to hack all the above code into your computer, please ask us for the source code! This is only meant here for an overview!

H.2 Algebra

H.2.1 Templatated Matrices and Vectors

Matrix.h looks as follows (for instance).

```c++
#ifndef MATRIX_
#define MATRIX_
#include <array>
#include <algorithm>
#include <array>
#include <iostream>
#include <stdexcept>
#include <string>
template<typename T, int Ni, int Nj>
class Matrix {
  using MatForm = std::array<std::array<T,Nj>,Ni>;
public:
  Matrix();
  Matrix(const Matrix& a) = default;
  std::array<T,Nj>& operator[](int i);
  std::array<T,Nj> operator[](int i) const;
  Matrix& operator+=(Matrix const& Q);
```
Matrix& operator=(double val);
Matrix<T,Nj,Ni> operator~() const;
private:
    std::array<std::array<T,Nj>,Ni> d;
};
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj>::Matrix() {}
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj>::Matrix(std::array<std::array<T,Nj>,Ni> a)
    : d(a) {}
template<typename T, int Ni, int Nj>
std::array<T,Nj>& Matrix<T,Ni,Nj>::operator[](int i) {
    return d[i];
}
template<typename T, int Ni, int Nj>
std::array<T,Nj> Matrix<T,Ni,Nj>::operator[](int i) const {
    return d[i];
}
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj>& Matrix<T,Ni,Nj>::operator+=Matrix<T,Ni,Nj> const& Q) {
    for(int i=0; i<Ni; ++i)
        for(int j=0; j<Nj; ++j)
            d[i][j] += Q.d[i][j];
    return *this;
}
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj>& Matrix<T,Ni,Nj>::operator+=(double val) {
    for(int i=0; i<Ni; ++i)
        for(int j=0; j<Nj; ++j)
            d[i][j] += val;
    return *this;
}
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj> Matrix<T,Ni,Nj>::operator~() const {
    Matrix<T,Nj,Ni> R;
    for(int i=0; i<Ni; ++i)
        for(int j=0; j<Nj; ++j)
            R[j][i] = d[i][j];
    return R;
}

//-----------------------------------------------
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj> operator+(Matrix<T,Ni,Nj> P, Matrix<T,Ni,Nj> const& Q) {
    return P+=Q;
}
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj> operator*(Matrix<T,Ni,Nj> Q, double val) {
    return Q*=val;
}
template<typename T, int Ni, int Nj>
Matrix<T,Ni,Nj> operator*(double val, Matrix<T,Ni,Nj> Q) {
    return Q*=val;
}
template<typename T, int Ni, int Nj, int Nk>
Matrix<T,Ni,Nk> operator*(Matrix<T,Ni,Nj> P, Matrix<T,Nj,Nk> const& Q) {
    Matrix<T,Ni,Nk> R;
    for(int i=0; i<Ni; ++i)
        for(int j=0; j<Nk; ++j)
            double res=0.;
            for(int k=0; k<Nj; ++k) res+=P[i][k]*Q[k][j];
        R[i][j] = res;
    return R;
}
template<typename T, int Ni, int Nj>
std::ostream& operator<<(std::ostream& ofs, Matrix<T,Ni,Nj> const& Q) {
    for(int i=0; i<Ni; ++i) {
The corresponding `main.cxx`, where we use the matrices, looks as follows.

```cpp
#include "Matrix.h"
#include <iostream>
using Matrix3 = Matrix<double,3,3>;
using Vector3 = Matrix<double,3,1>;
int main() {
    Matrix3 M {{{{1,2,3},
                  {4,5,6},
                  {7,8,9}}}};
    std::cout << "M = " << M << std::endl;
    Matrix3 P = (~M);
    std::cout << "P = " << P << std::endl;
    Matrix3 S = 0.5*(M + P);
    std::cout << "S = P+M =" << S << std::endl;
    Matrix3 Q = M * P;
    std::cout << "Q = M*P =" << Q << std::endl;
    Vector3 v {{1,2,3}};
    std::cout << "v = " << v << std::endl;
    Vector3 w {2*v};
    std::cout << "w = " << w << std::endl;
    Vector3 x = v+w;
    std::cout << "x = v+w = " << x << std::endl;
    std::cout << "x*v = " << (~x)*v << std::endl;
    Vector3 y = M*v;
    std::cout << "y = M*v = " << y << std::endl;
    double val = ((~v)*M*v)[0][0];
    std::cout << "v*M*v = " << (~v)*M*v << " = " << val << std::endl;
}
```

So, at least from the point of usage, not much has changed! But the header is, of course, crowded with template nomenclature. One can hardly change this!

On the pro side is that we now can handle vectors and matrices on the same footing – it’s just one class (template) definition. We can even further define any shape matrix (square or non-square), and exercise operations on them.

By using `operator~`, also the transpose operation is included. For matrix and inner products, it is now needed.

We cannot execute all the operations anymore. Non-square matrices usually have no inverse, in general, for instance. That’s at the downside. Before, we also managed that a bilinear form and an inner product of vectors returns `double` value. Now it's a `std::array<std::array<double,1>,1>`.

For output, that’s not critical. But we have to perform a conversion (picking the [0][0] element), if we assign it to a `double` variable. There are certainly ways around that issue. We leave this to the reader to accomplish.

Although we have gained a lot of flexibility, you should be aware about that we created actually many matrix types at the same moment! And everything happens during compile-time. There is no chance to perform a run-time change of type. That’s inherent to templates.

This is actually also the reason, why we included the implementation of the operators and class member functions into the header. You cannot separately compile and use them, because template resolution happens during compile-time. So using a template immediately means that the template implementation must be accessible during compile-time. Otherwise, you will obtain a linker error. Please try! It is a good exercise in understanding!
H.2.2 Solution: Eigen Library (Tutorial)

Playing on this higher level of solvers and preconditioners is optimization on the algorithmic level. The same concerns the optimization with respect to the time step and other parameters, which influence the speed of the simulation. This is rather different from the optimization on code level, in the previous part, which we would call lower-level optimization. But if the algorithm is badly chosen, almost any lower-level optimization is hopeless. This should give you an indication to possibly think about your problem at hand before starting to implement.

Concerning the boundary conditions: Fixed values of $\psi$ at the boundaries are put to the right-hand side. The discretized equation of $\psi$ at a point next to a boundary will just contain $\psi$ at this boundary point. As it does not change, it will not be part of the system matrix. Derivatives of $\psi$ at the boundary are more difficult to handle. As those boundary conditions also must be discretized, you will dynamically express $\psi$ on a boundary point by $\psi$-values of inner (i.e. non-boundary) points. This will reduce the system matrix by the row and column for that boundary point. Additionally, the other non-boundary point matrix entries (for those points effected by the BC) will experience a change in the system matrix. This might also – by means of the BCs – make the system matrix non-symmetric (if it was not anyway before). And non-symmetric matrices require different kinds of iterative solvers, which will most probably be more inefficient than those for symmetric matrices!

H.3 Solution: Template Meta-Programming and Type Traits (Introduction)

There are lot of books like e.g. [1, 43], and also more web pages (see e.g. Wikibooks: Template Meta-Programming) that describe in more detail practical applications of TMP and Type Traits.

For the algorithm, a supposedly $O(N)$ implementation is to start with one iterator at the beginning, and the other at the end. If the sum of those two elements is larger than the compare value (8), shift the right iterator to smaller values. Otherwise, if it is smaller, shift the left iterator to the right. Stop the loop if either a match is found, or latest if both iterators are equal.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

template<class Iter, class CompType, 
        class BinaryOperation = std::plus<CompType>>
bool AnyOf_SumEqualVal(Iter beg, Iter end, CompType val, 
        BinaryOperation op = std::plus<CompType>{} )
{
    end--; 
    while(beg!=end) {
        CompType tmp = op(*beg, *end);
        if(tmp==val) return true;
        if(tmp>val) end--;
        else beg++;
    }
    return false;
}

int main() {
    std::vector<int> v = {1,3,3,4,7,9,11};
    if(AnyOf_SumEqualVal(v.begin(), v.end(), 8))
        std::cout << "Found Pair\n";
    else
        std::cout << "Found No Pair\n";
}
```

Obviously, this makes sense only for linear (i.e. consecutive in memory) containers, which are sorted. Conceivable are also lists and also sets. Important is that a forward and a backward (reverse) iterator is present. For the type CompType, it must be fulfilled that it is operator== and operator> comparable.

1The lowest are the optimizations the compiler can do. But he needs maybe help by the programmer.
and that the BinaryOperation is defined. As we wrote it, it does not need to be \texttt{operator+} exclusively (which we but chose as default)!

As we designed this algorithm, and thus also the choice of name, we obtain only a statement whether two such elements exist. It conceivable to also return the pair of numbers first found. How would you realize this?

**H.4 Solution: Named Parameters without Boost (Tutorial)**

Nothing to solve!
Bibliography


