Christian Engwer, Simon Praetorius

Scientific Programming

Advanced Concepts

Scientific Programming with C++

Advanced Concepts

Christian Engwer, Simon Praetorius

Summer term 2021

Prof. Dr. Christian Engwer
Westfälische Wilhelms-Universität Münster
Orleansring 10
48149 Münster

Dr. Simon Praetorius
Technische Universität Dresden
Willersbau B219
Zellescher Weg 12–14
01069 Dresden

These lecture notes are based on lectures held at TU Dresden and WWU Münster.

It is continuously developed since the initial lecture in So2014. This is *work in progress* and may contain typos and errors that need to be corrected in future releases of the notes.

You are welcome to submit any correction.

Preface

The focus of this module lies on aspects of software development like programming on high-performance computers, object-oriented software design, generic (template-based) programming, and the efficient implementation of numerical algorithms. Additionally experience in analysis, application and extension of software and software libraries is developed. This module in the summer term especially focuses on software development with the programming language C++.

Three main learning goals can be formulated:

1. You know how to program with modern C++, using generic programming and advanced techniques, like meta programming, expression templates, and concepts.
2. You know how to use programming tools and you can apply these tools to debug, benchmark, and manage your code. The list of tools include compilers, build systems, version control, debuggers, and profilers.
3. You can read, understand, and utilize (scientific) software libraries, like BLAS (Basic Linear Algebra Subroutines), LAPACK (Linear Algebra Package), STL (Standard template library), Dune (framework for the discretization of partial differential equations), MTL4 (Matrix Template Library), Boost (portable C++ library).

There are exercises every week to practice the C++ programming. During the semester programming projects in groups are assigned. Important parts of the exercises are reviews of C++ code. You will get reviews for your own written code and have to perform reviews of the code of others in the course.

Prerequisites Basic experience with at least one programming language (e.g. Fortran, python, C, Java, ...), No prior knowledge about C++ is required. Basics knowledge of numerical mathematics.

Target group This course is for master and PhD students of Mathematics, Technomathematics, and Mathematics in Business and Economics as well as for students in the masters programs “Computational Modeling and Simulation” (CMS) especially for the specialization in Computational Mathematics.

Language The course will be taught in English.

Teaching concept The lecture will be taught remotely via BigBlueButton, in the (a)synchronous format 2+2 (2h lectures + 2h tutorial). Lectures are recorded, while the tutorials are more interactive and not recorded. There will be weekly exercises. Some of those are marked as homework projects that get reviewed, others are group projects to be worked on during the tutorials. A third type of exercises is peer-review of the code submissions (this will be explained in the tutorial).

Contents

1 Introduction

"It would be nice if every kind of numeric software could be written in C++ without loss of efficiency, but unless something can be found that achieves this without compromising the C++ type system it may be preferable to rely on Fortran, Assembler or architecture-specific extensions."

— Bjarne Stroustrup (1994) [1]

Scientific programming is an old discipline in computer science. The first applications on computers were indeed computations. In the early decades, ALGOL was a relatively popular programming language, competing with FORTRAN. FORTRAN 77 became a standard in scientific programming because of its efficiency and portability. Other computer languages were developed in computer science but not frequently used in scientific computing: C, Ada, Java, C++. They were merely used in universities and labs for research purposes.

C++ was not a reliable computer language in the 90s: code was not portable, object code not efficient and had a large size. This made C++ unpopular in scientific computing. This changed at the end of the nineties: the compilers produced more efficient code, and the standard was more and more supported by compilers. Especially the ability to inline small functions and the introduction of complex numbers in the C99 standard made C++ more attractive to scientific programmers.

Together with the development of compilers, numerical libraries are being developed in C++ that offer great flexibility together with efficiency. This work is still ongoing and more and more software is being written in C++. Currently, other languages used for numerics are FORTRAN 77 (even new codes!), Fortran 95, and Matlab. More and more becoming popular is Python. The nice thing about Python is that it is relatively easy to link C++ functions and classes into Python scripts. Writing such interfaces is not a subject of this course, though.

The quote of Bjarne Stroustrup, the developer of the C++ programming language, is from the mid 90s. Since then a lot has changed.

"C++ is now ready to compete with the performance of Fortran. Its performance problems have been solved by a combination of better optimizing compilers and template techniques. It's possible that C++ will be faster than Fortran for some applications."

— Todd L. Veldhuizen (2000) [2]

The goal of this course is to introduce students to the exciting world of C++ programming for scientific applications. The course does not offer a deep study of the programming language itself, but rather focuses on those aspects that make C++ suitable for scientific programming. Language concepts are introduced and applied to numerical programming, together with the STL and Boost.

It is not our goal to explain all C++ features in a well-balanced manner. We rather aim for an application-driven illustration of features that are valuable for writing **well-structured**, **readable**, **maintainable**, **extensible**, **type-safe**, **reliable**, **portable**, and last but not least **highly performing** software. In order to achieve this goal we not only want to explain a programming language, but also discuss best practice in coding and project management of scientific software.

1.1 Motivating example

While in classical computer science courses the famous *hello world* example is shown at the beginning, in the scientific programming lecture we switch to a more numerics oriented initial example: matrix-vector multiplication.

Let $A \in \mathbb{R}^{n\times n}$ be a real (dense) matrix of size $n \times n$, with $n > 0$ a positive integer, and $\mathbf{x} \in \mathbb{R}^n$ a corresponding real vector of size n. We do not assume any special properties of A, like symmetry, bandedness, or triangular shape. As will be explained later in the lecture and in some exercises, we have two classes, `DenseMatrix` and `DenseVector` that can represent our real-valued matrix A and vector $\mathbf{x}$:

```
DenseMatrix A(100, 100);
// initialize A
DenseVector x(100);
// initialize x
```

We want to solve the task $\mathbf{y} = \alpha A\mathbf{x} + \beta\mathbf{y}$ (accumulated matrix-vector multiplication) with $\mathbf{y} \in \mathbb{R}^n$ initialized to zero and $\alpha = \beta = 1$ for simplicity.

```
DenseVector y(n, 0.0);
```

First, we are using the library CBLAS:

```
int n = A.rows();         // number of rows of the matrix A
                          //  == number of columns of the matrix A
int lda = std::max(1,n);  // Specifies the leading dimension of array
↪  storing the value of A
int incx = 1;             // Specifies the increment for the elements of x
int incy = 1;             // Specifies the increment for the elements of y
double alpha = 1.0, beta = 1.0;

cblas_dgemv(CblasRowMajor, CblasNoTrans, n, n, alpha, &A[0][0], lda,
            &x[0], incx, beta, &y[0], incy);
```

Second, we solve the same task, using some techniques we will develop during this lecture:

```
y += A*x; // or more generally: y = alpha*A*x + beta*y;
```

The C++ version more clearly states what it computes, has no arguments that you could mix up, and (this might be surprising) is not necessarily slower than the first version and even can call all variants of the `clas_***mv` methods from CBLAS simply by analyzing the arguments passed to the multiplication operator `*`.

Remark 1. *If you test the initial exercise implementation, you will find that it is indeed slower than the CBLAS version. During the semester we will learn some more advanced techniques and ways to come close the performance of an optimized blas. An example of a high-level C++ implementation beating the BLAS call is MTL4 by Peter Gottschling.*

Remark 2. *In BLAS there are multiple different versions of the* `dgemv` *function. The function name encodes different constellations, i. e., the first letter* `d` *means* `double` *precision, the letters* `ge` *means* general, `m` *mean* matrix *and* `v` *means vector. So, it implements the matrix-vector product of a general matrix in double precision. Other precision types are* {`s`*: single precision,* `d`*: double precision,* `c`*: complex single precision, and* `z`*: complex double precision*}*, for the matrix type, we have* {general matrix, general band matrix, hermitian/symmetric matrix, triangular matrix, ...}*, and the matrix can be transposed, or conjugate transposed, and can be in different storage format.*

Thus, BLAS provides many combinations of these different properties, but all functions have *a different name, and different arguments. It is not so easy to get it right and is hard to debug and to find errors. Even the documentation of just 3 different BLAS Level 2 functions in all its variants is a 24 pages long document.*

On the other hand, in C++ you can encode several of these properties in the data-type of the matrix. This allows to generate for the ***same*** *function call different specialized implementations by the compiler. So, there is no overhead of dispatching by the various matrix properties. The MTL4 library implements this and we will also see in this lecture how to switch an algorithm by inspecting type properties.*

This lecture goes beyond the basics of the C++ programming language and tries to teach how to write, debug, and manage code, how to analyze performance and how to test and optimize your implementation. Classical design patterns of object-oriented programming is more the focus of the twin lecture SCPROG – Fortgeschrittene Konzepte des Wissenschaftlichen Programmierens: OOP mit Java – by Prof. W. Walter.

1.2 History of C++

The programming language has a long history, that is described in the books by Bjarne Stroustrup: *The Design and Evolution of C++, 1994*, `http://www.stroustrup.com/hopl2.pdf` (*A History of C++: 1979–1991*) and `http://www.stroustrup.com/hopl-almost-final.pdf` (*Evolving a language in and for the real world: C++ 1991–2006*). A summary of this can be found at `http://en.cppreference.com/w/cpp/language/history`. The following story is mainly based on the book from 1994.

1.2.1 Early C++

The history of C++ goes back to the year **1979** where Bjarne Stroustrup worked in his doctoral thesis on topics about the programming language *Simula*. This language is named on of the first object-oriented programming languages, developed in the 1960s by Ole-Johan Dahl and Kristen Nygaard. Bjarne recognized the advantage of this programming technique in the area of software development, but also has seen the disadvantage of Simula, its very low performance for real-world projects.

Thus, he decided to design a new language including ideas of Simula. The first development in this direction was done in the years **1979–1983**. He extended the language C by the concept of classes – a combination of data and functions acting on this data $\to$ *C with classes*. A first compiler based on a C compiler was build (*Cfront*). The feature set of this new programming language can be summarized as: Classes, Derived class (no virtual functions), public/private access control, constructors/destructors, Call and return functions (later removed), friend classes, type checking and conversion of function arguments, inline functions, default arguments and assignment-operator overloading.

When it came to more and more features, the concept C with classes was buried and designed from scratch. Also the compiler based on a C compiler was abandoned. In the year **1983** a new language with a new name was born: *C++*, derived from the increment operator in C, where the `++` could be understood as *next*, *successor*, or *increment* of the C language. New features, compared to *C with classes* introduced in C++ are: virtual functions, function name and operator overloading, references, constants, user-controlled free-storage memory control, improved type checking. A long list a more feature was added in the Release 2.0 of the language around **1986-1987**.

A first reference manual, *The C++ programming language*, was written in the year **1985**, and later in **1990** the book *The Annotated C++ Reference Manual*. The language was not yet standardized. So, the books are kind of the standard reference manuals. During these days other compiler vendors published a C++ compiler, like the *Borland C++ Compiler* and in **1987** C++ was added to the GNU Compiler Collection GCC.

1.2.2 Standard C++

The first ISO standard of C++ was published in the year **1998**, also called `C++98` standard. From that release on, the C++ versions are named after the publication year of the standard document. Around **1979** an accompanying library was developed, providing standard containers and algorithms. This library, known as the standard library, later became part of the C++ standard. During the year **1999** the collection *Boost* was founded by some standard committee members, to push the development of standard library components.

A first correction of the `C++98` standard was published in **2003**. It resolved several smaller or larger errors and problems in the 1998 standard and was named again after the publication year, `C++03` standard. In **2005** extensions of the 2003 standard were collected and summarized in the technical report TR1. Those developments were the bases for a new standard that was intended to be published in the early 2000s. This can be seen on the working title of this standard `C++0x`. But it needed more time and couldn't be finished within the same decade. Finally in 2011 the standard `C++11` was finalized and published. It contains a long list of new features (including lambda functions, `constexpr`, auto type deduction, `decltype` specifier, several library extensions like tuples, arrays, type-traits and a lot more) and was the biggest new release since the initial version from 1998.

Since version 4.8.1. of GCC the `C++11` version is completely implemented. In Clang since version 3.3., in Intel ICC Compiler since version 15.0, and in Microsoft Visual Studio Compiler since version 19.0. See `https://en.cppreference.com/w/cpp/compiler_support` for an overview of the compiler support of C++ features.

Although the `C++11` standard brought so many new features, some could not be completed in the standard, others needed minor fixes. This is the reason why in **2014** the next *minor* revision of the standard, called `C++14`, was published. But it is more than a revision, it is a completion of the `C++11` standard. The working title of the standard was `C++1y` and the Compiler vendors finished their implementation of the standard with version 5 of GCC, version 3.4 of clang, version 17.0 if Intel ICC, and version 19.10 of MSVC. Due to some bugs in the implementation, the recommendation for GCC is version 6.1 for `C++14`, though.

`C++14` is not the most recent standard. Since the release of the `C++11` standard in 2011, it was decided to have a fixed release cycle of 3 years, where each new feature that is not completely ready until a some feature freeze date has to wait for the next release and not the release for the feature. The current published C++ release is `C++17` and the development for `C++20` (working title `C++2a`) is in its final standardizing phase. Both standards are only partially implemented by the compiler vendors, but at least the core language features of `C++17` are now finished. It needs approximately 3 years after a standard is published until all feature and library components are implemented by the compilers GCC, clang, ICC, and MSVC.

Proposals for new features and some discussion of the future development of C++ can be found at the ISO standard working group 21 page `http://www.open-std.org/jtc1/sc22/wg21/`.

2 Language Basics

2.1 Introductory example

Introductory example

```cpp
#include <iostream>
#include <boost/numeric/mtl/mtl.hpp>

using namespace mtl;

int main(int argc, char** argv)
{
  int const size = 40, N = size * size;
  using matrix_t = compressed2D<double>;

  // Set up a matrix 1,600 x 1,600 with
  // 5-point-stencil
  matrix_t A{N, N};
  mat::laplacian_setup(A, size, size);

  // Compute b = A*x with x == 1
  dense_vector<double> x{N, 1.0}, b;
  b = A * x;

  std::cout << two_norm(b) << std::endl;
}
```

Some comments about this example:

- Main program: The entry point for the program is the function `main`. It must be available in all executables and returns an integer indicating an error code of the program (`0` means no error).
- Typically, there are just two variants of the `main` function allowed, without arguments or with two arguments containing command-line parameters to the program. Thereby, `int argc` indicates the number of command-line arguments and `char** argv` or `char* argv[]` a sequence of null-terminated character sequences (strings) representing the actual command-line arguments. Especially, the zero-th argument `argv[0]` corresponds to the name of the executable.

  Remark: Some compilers allow more than those two arguments, that may contain environmental variables.
- Input- and Output is not part of the C++ core language, but is implemented in libraries, like the C++ standard library. Those libraries must be included explicitly.
- Include-files are any regular files that can be found by the compiler. Typically, those includes have the file ending `.h` (for header file) and contain specifications of the interface of functions or even implementations of those functions. Include in C++ means: the

text of the file is copied to the include directive `#include` into the code.

Remark 1: Header-files of the C++ standard library do not have a file extension. This is, in order to avoid conflicts with the C standard library (those files have the extension `.h`).

Remark 2: There are two variants for the include directive: `#include <Dateiname>` or `#include "Dateiname"`. In the first variant the include files are searched in the compiler include paths and system paths only while in the second variant it is searched also in the current source directory. This is why the standard library include directives are typically written with angular brackets `<...>`.

- The functions of the standard library are grouped in the namespace `std`. Again, this is done in order to avoid conflicts with functions from other libraries and your own code. In order to call functions from the standard library, you have to add the prefix (name resolution operator) `::`, e. g., `std::sqrt`.
- In addition to the main program, we have a class (structure) `compressed2D<double>` and (free) functions `two_norm`, and `mat::laplacian_setup`. A free function is a function not part of a class. But, there is also a function bound to the class.
- Finally, the output of the result to the screen is by an output-stream object `std::cout` (part of the standard library). It provides a way to assign new output data to the output device, using the "shift" operator `<<`. The expression `std::endl` thereby indicates a line-break (the end-of-line symbol, and it flushes the output). One could also write the character `'\n'` directly.
- Brackets `{ }` in C++ inclose a local code block (scope). Variables declared inside a scope can only be accessed from within that scope or a sub-scope. The brackets are also used during the initialization of an object.
- Single line comments are introduced by `//` and multi-line comments are introduced by `/* ... */`

Some questions to think about:

- What happens if you add another `main(...)` function to the code? Is it possible to add both main functions, `main()`, `main(int, char**)` at the same time?
- There is not just text pushed to the output stream, but also values (numbers). How are numbers printed / converted to strings? How to modify this behavior?

2.2 Compiling C++ code

Compared to scripting or interpreted languages, like Python, Matlab, or JavaScript, C++ code must be translated into machine-readable, executable instructions. This process of translation is called **Compiling**. More generally, one could understand compiling as a transformation of code from one (high-level) language to another (low-level) language.

Remark 3. *During the compilation of C++ code, you might even print out intermediate states of its transformation process, like preprocessor output, or assembler output. We will look at these intermediate code in the lecture or exercises to understand better what the compiler is doing with our code.*

- The process of compiling is performed by a program, called the **compiler**. Typical examples of compilers are *g++*, *clang*, *Intel ICC*, *MSVC*, and others.
- The compiler gets as input a **translation unit**, typically a text file containing the C++

code — the definition of functions and classes. A program typically consists of many translation units that are combined.

- The output of the compiler is a collection of **object files**, one for each translation unit.
- To generate an executable (or a library) from these object files, the **linker** combines all the objects to a single file.

The process of compiling can be split into several stages:

pre-processing (performed by the **preprocessor**) The content of include files is copied to the include directives, macros and preprocessor constants are evaluated.

linguistic analysis Check of syntax rules.

assembling Translation of the language constructs into CPU instructions, e. g., in form of assembler code.

code output Transformation of internal code (assembler code) into machine-readable binary code. Collection of symbols into a symbol table with jump references.

On many linux distributions the C++ compiler of the GNU Compiler Collection (GCC) or the clang compiler of LLVM are preinstalled. Assume that the code from the introductory example is stored in a text file `distance.cc`. This can be compiled into an executable by

```
c++ distance.cpp
```

where `c++` is an alias (often a symbolic link) to the actual compiler.

Remark 4. *The version and name of the compiler can be obtained by* `c++ --version`*.*

The result of the compilation is a binary file, named `a.out`. This is the default executable name, that can be changed by providing the argument `-o <name>`, e. g.,

```
g++ distance.cpp -o distance
```

Later in the lecture we get to know different C++ language features available in a specific version of the C++ standard (see history of C++). The standard can be selected explicitly by the additional argument `-std=<version>`, e. g., for `C++11`:

```
g++ -std=c++11 distance.cpp -o distance
```

where the `<version>` follows the naming given in the chapter *History of C++*.

If you have multiple files to compile, e. g., one file provides the implementation of the functions and classes, and the other file just the `main()` function, we say that we have multiple translation units. Those can be compiled individually and then linked together:

```
g++ -c file1.cpp
g++ -c file2.cpp
g++ file1.o file.o -o program
```

The output name of the compiled translation units follows the pattern `<basename>.o`. The compiler allows to combine the compiler and linker call in one line, by listing all the files to compiler one after the other:

```
g++ file1.cpp file2.cpp -o program
```

If a source file depends on some include files (in the top of the file you find the lines `#include <...>` or `#include "..."`), the compiler has to search for these *header*-files. It automatically searches in default system paths, but for everything else the compiler has to be pointed to the location of the include files. This can be done by the additional argument `-I<path-to-files>`, e. g.,

```
g++ -I/usr/local/library/include/ file1.cpp file2.cpp -o program
```

and if the program depends not only on include files, but also **symbols** (compiled implementations) of library functions, a list of additional libraries to link the executable with has to be appended. Therefore, two arguments are allowed for the compiler: `-L<path-to-library>` and `-l<libname>`, where `<libname>` contains the part of the file name of the library between the prefix `lib` and the file extension `.so` or `.a`. (This might be different on different operating systems, like MacOS or MS Windows).

```
g++ -I/usr/local/library/include/ file1.cpp file2.cpp -o program \
    -L/usr/local/library/lib -llibrary
```

If your project depends on multiple libraries that itself depend on other libraries it gets more and more complicated to put everything correctly into the compile command. To simplify this, there are multiple different **build systems** developed that collect and analyze dependencies and generate compiler commands for you. A classical one is a **Makefile**, that defines various targets that can depend on each other and some way to construct from these targets a sequence of commands to execute in order to compile (build) the executable. Another example is **CMake** (more precisely it is a build system generator).

Remark 5. *As you may have noticed, source files that are compiled by the compiler are typically named with a file extension `.cc`, `.cpp`, or `.cxx`. This differs from the include (header) files with file extension `.h`, `.hh`, `.hpp`, or `.hxx`. Here the first file extension comes from C and is just a abbreviation for header. Later in the lecture, we will see source (implementation) files, that are not compiled, but are typically included at the end of the corresponding header file. This is related to template implementations. Sometimes these files are name `.tpp`, or `.txx`, but more ofter just `.impl.hh`, or `.inc.hh` (with any of the header file extensions from above).*

While file extensions and naming of files in general is arbitrary, it is recommended to name source and its corresponding header file with the same base name and matching file extensions, e. g., `linear_algebra.hh` and `linear_algebra.cc`. Use the standard file extensions also to get automatic syntax highlighting in your code editor of choice.

2.3 Basic structure of a C++ program

Each C++ code resulting in an executable must contain exactly one `main(...)` function, while both variants

```
int main();
int main(int argc, char* argv[]); // or. int main(int argc, char** argv);
```

are allowed. The arguments `argc, argv` are filled when running the executable with command-line arguments. Thereby, the argument `argc` represents the number of command-line arguments and `argv` represents and *array* of *strings* (character sequences) representing each individual command-line argument. The fist entry in this array, `argv[0]`, contains the name of the executed program.

Splitting in multiple source files Code can (and should) be split into multiple translation units representing different components of the program. This splitting means multiple header and source files, where each source file can be translated into an object file without the knowledge of the other source files.

Typically, in header files the functions and classes are just **declared**, while in the source file those entities are **defined**.

Example 1: A header file contains the **prototype** (interface description) of a function and a class definition.

example.hh

```cpp
#ifndef EXAMPLE_HH
#define EXAMPLE_HH

// declaration and definition of a class
struct Point
{
  double x, y;

  // declaration of a member function
  Point subtract(Point const& other) const;
};

// declaration of a function
double distance(Point const& a, Point const& b);

// declaration of a template function
template <class T> void foo();

#include "example.impl.hh"
#endif // EXAMPLE_HH
```

Example 2: The definition of a template function (included at the end of the header file)

example.impl.hh

```cpp
#pragma once
// definition of the function foo()
template <class T>
void foo() { /*...*/ }
```

Example 3: The source file, includes the header file and defines the functions

example.cc

```cpp
#include "example.hh" // include the declaration
#include <cmath>      // include additional function (declarations)

// definition of a member function
Point Point::subtract(Point const& other) const
{
  return {this->x - other.x, this->y - other.y};
}

// definition of the function distance()
double distance(Point const& a, Point const& b)
```

```
{
  Point ab = a.subtract(b);
  return std::sqrt(ab.x * ab.x + ab.y * ab.y);
}

int main(int argc, char** argv)
{
  Point a{ 1.0, 2.0 }, b{ 7.0,-1.5 };
  distance(a,b);
  return 0;
}
```

Some remarks to the examples above:

- The triplet `#ifndef NAME`, `#define NAME` and `#endif` builds a so called **include guard**. It prevents the header file to be included multiple times in the same translation unit. This is not allowed, since the C++ standard imposes a **one definition rule**, meaning: No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.

  Another way of enforcing that a file is included only once, is by using the (non-standard) preprocessor directive `#pragma once` in the top of the include file. This directive is supported by all major compilers and can be used without any problems.

- If you want to (or have to) provide an implementation of a function or class method in a header file, it must be included together with the corresponding declaration. Often this is done by an include statement at the end of the header file. Or the definition is provided together with the declaration.

2.4 Variables and Datatypes

C++ is a statically typed language (in contrast to dynamically typed languages like e. g., PHP), meaning: each identifier and expression in a C++ program has assigned a type that is already known to the compiler and this type cannot be changed.

Examples:

```
float x;          // x is a single precision floating point number
int y = 3+4;      // y is an integer variable with initial value 7
float f(int);     // f is a function with one integer argument and float
↪   return type
```

Here, the variable `y` is initialized with an expression on the right-hand side of the assignment operator `=`. This expression `3+4` also has a type. Since `3` and `4` are integer numbers and the result of the addition of two integers is defined to be also an integer, the expression is of type `int`.

An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects.

— C++-Standard (N4835) §7.1 (1)

Remark 6. *That the expression 3+4 has the type `int` is not as trivial as you might think. In some languages it might be a type that could be larger than `int` but that can hold the value of the addition of these two integers.*

2.4.1 Automatic type deduction

When the compiler parses an expression, it internally determines its type. In order to create a variable of just that type, the language has introduced the meta-type `auto`. It is not an actual data-type, but is a placeholder for the type determined by the expression of the initialization:

```
auto x = 3+4;       // deduce the type from the expression: x <-- int
auto y = long{3+4}; // explicitly committing to a type: y <-- long
auto z{3+4};        // same as variable x: z <-- int [C++17]
```

2.4.2 Literals

A literal is a token directly representing a constant value of a concrete type.

Examples:

```
42u       // unsigned integer literal
108.87e-1 // floating point literal
true      // boolean literal
"Hello"   // string literal
```

The type of the literal is often determined by a literal suffix (like in `42u` the `u`). All integer literals are just a sequence of digits that has no period or exponent part, while floating point literals must contain a period and/or an exponent part. Character literals are introduced with a single quote, e.g., `'c'`, and string literals with the double quotes, e.g., `"Hello"`. There are some more literals, we will see in the exercise.

Remark 7. *Since `C++11` one can define own literals of the form `built-in literal + _ + suffix`. This allows, for example, to create numbers with units.* [C++11]

Example:

```
101000101_b  // binary representation
63_s         // seconds
123.45_km    // kilometer
33_cent      // cent
```

where the implementer is responsible for giving those literals a meaning.

Remark 8. *A literal is a primary expression. Its type depends on its form (see above). A string literal is an* ***lvalue****; all other literals are* ***prvalues*** *(see chapter about value categories).*

2.4.3 Declaration – Definition – Initialization

Declaration A declaration may introduce one or more names into a translation unit or redeclare names introduced by previous declarations.

Definition A *declaration* that provides the implementation details of that entity, or in case of variables, reserves memory for the entity.

A *declaration* of a class (`struct`, `class`, `enum`, `union`), function, or method is a definition if the declaration is followed by curly braces containing the implementation body.

Variable declarations are always *definitions* unless prefixed with the keyword `extern`.

Initialization A *definition* with explicit value assignment.

Examples:

```
class Test;               // declaration of a class
class Test {};            // definition of that class

int func();               // declaration of a function
int func() { return 7;} // definition of that function

extern int i=func();    // definition and initialization of a variable
extern int j;             // declaration of a variable
int k;                    // definition of a variable

int obj();                // !!! declaration of a function
```

A fundamental rule is that you are not allowed to define an object twice. While it may be allowed to declare exactly the same object multiple times, even after the definition.

> ***One-definition rule:*** *No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.*
>
> — C++-Standard (N4835) §6.3 (1)

> *Declare variables as late as possible, usually right before using them the first time and whenever possible not before you can initialize them.*
>
> — **Principle**

2.4.4 Fundamental Types

We have seen already some types in the examples above, like integer types and floating-point types. There are more fundamental data-types available in C++. A summary can be found at `http://en.cppreference.com/w/cpp/language/types`.

Basic types in C++ are categorized into three groups: integral types, floating-point types, and `void`. Integral types represent integer numbers, while floating-point types might represent fractions.

The type `void` represents the empty set of values. No variable can be declared of type `void`. Thus, `void` is an *incomplete type*. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type `void`.

Integral numbers

The group of integral types contains

- The boolean type `bool` with values `true` and `false` (both are boolean literals). The size of that type is implementation defined and typically 1 Byte.
- Character types `char`, `signed char`, and `unsigned char` to represent a single character. These are distinct types and either a signed or unsigned integer type of size 1 Byte. There are also larger character types like `wchar_t`, `char16_t`, or `char32_t` to represent larger character sets.
- Standard (signed/unsigned) integer types include `short int`, `int`, `long int`, `long long int` possibly qualified with the type prefix `signed`, `unsigned`. No signed-ness qualification means signed integers. The postfix `int` may be omitted (except for `int` itself).

  The range of representable values for a signed integer type is -2^{N-1} to $2^{N-1}-1$ (inclusive), where N is called the width of the type. An unsigned integer type has the same width N as the corresponding signed integer type. The range of representable values for the unsigned type is 0 to 2^{N-1} (inclusive).

Arithmetic for the unsigned type is performed modulo 2^N. *Note:* Unsigned arithmetic does not overflow. Overflow for signed arithmetic yields **undefined behavior** (what this means is explained later).

In the C++ standard the sizes (widths) of the integer types are not specified explicitly, but a minimal size is given. Thus, one finds the relations

```
sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

where `sizeof` is a C++-operator returning the width of a data-type (or an expression) in Byte. On 32-Bit systems, typically the sizes 2, 4, 4, 8 Byte are used, on 64 Bit systems `long` is often of size 8 Bytes.

The type `long long` was introduced in `C++11` and was available as compiler specific extensions before. [C++11]

Remark 9. *As for `char` there are integer type with prescribed width, defined in the header file `<cstdint>`. Those are named `std::int16_t`, `std::uint32_t`, ...* [C++11]

Remark 10. *In the standard library a **type-alias** is introduced for an integer type often used for vector indices and vector sizes, named `std::size_t`. It is typically an `unsigned long int` type, but on some compilers it may be different. The type referenced by `std::size_t` can store the maximum size of a theoretically possible object of any type (including array).*

There are special *literals*, suffixes appended to numbers, to indicate explicitly a type: `U`, `u`, `L`, `l`, `LL`, `ll`, where there is no difference between lower and upper case suffixes. `u` means `unsigned`, `l` means `long`, and `ll` means `long long`. Additionally, a prefix can be put in front of the number to indicate a base for the number systems used: `0` (Null), `0x`, or `0b`. Those represent octal, hexadecimal, or binary numbers, respectively.

Remark 11. *The type of the integer literal is the first type $\geq$ in which the value can fit and with the right signed-ness.*

Example:

```
1234L, 9565ul, 012 == 10, 0x2a == 42
```

> *An integer literal is a sequence of digits that has no period or exponent part, with optional separating single quotes that are ignored when determining its value. An integer literal may have a prefix that specifies its base and a suffix that specifies its type.*
>
> — C++-Standard (N4835) §5.13.2 (1)

Floating-point types

Standard types for floating-point numbers are

```
float, double, long double
```

The range of possible values is defined in `<limits>` and the sizes may be compiler dependent, typically 4, 8, 10 Byte. The relation

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

holds for the floating-point types.

Literals, to indicate how to interpret a number, are `F,f,L,l` for `float` and `long double`.

Remark 12. *In GCC an extension is implemented to allow quad-precision arithmetics with the data-type `__float128`. The size is 16 Byte. Typically, this is implemented as a software library, e. g., by concatenating two `double` types. Only rarely there is hardware support for quad precision numbers (e. g., IBM POWER9 CPU). The arithmetic is defined in the standard document IEEE 754-2008.*

Remark 13. *To do arithmetic with arbitrary precision, there are multiple libraries available. Examples include GNU GMP (Gnu MultiPrecision Arithmetic Library) and Boost.Multiprecision library.*

An example of high precision calculation of the enclosed area of a circle with boost multiprecision is given below. Note, it uses templates to implement the actual algorithm.

multiprecision.cc

```
#include <iostream>
#include <boost/multiprecision/cpp_dec_float.hpp>
#include <boost/math/constants/constants.hpp>

// Type-alias for floating-point numbers with 50 decimal digits precision and
↪  int32_t to represent the exponent
using float_50 = boost::multiprecision::cpp_dec_float_50;

template <class T>
T area_of_a_circle(T r)
{
   using boost::math::constants::pi;
   return pi<T>() * r * r;
}

template <class T>
int digits() { return std::numeric_limits<T>::digits10; }

int main()
{
  float r_f = float(123) / 100;
  float a_f = area_of_a_circle(r_f);

  double r_d = double(123) / 100;
  double a_d = area_of_a_circle(r_d);

  float_50 r_mp = float_50(123) / 100;
  float_50 a_mp = area_of_a_circle(r_mp);

  // 4.75292
  std::cout << std::setprecision(digits<float>()) << a_f << std::endl;
  // 4.752915525616
  std::cout << std::setprecision(digits<double>()) << a_d << std::endl;
  // 4.7529155256159981904701331745635599135018975843146
  std::cout << std::setprecision(digits<float_50>()) << a_mp << std::endl;
}
```

Remark 14. *Arithmetic with floating-point numbers is not the same as arithmetic with real $\mathbb{R}$ numbers. There are effects of rounding, finite representation, cancellation, non-associativity, ... Details can be found in the standard document IEEE 754 and are explained in the lecture* Computer Arithmetics *by Prof. W. Walter.*

2.4.5 Number conversion

Whenever you initialize a variable with an expression, the value of that expression must be converted to the type of the variable.

Example:

```
int l = 1234567890123; // number will be narrowed to fit into integer int
```

Definition 1. We call an initialization of a value to a smaller type that cannot represent this value a *narrowing initialization* or *narrowing conversion.*

In the example above, the compiler will not give any error and compiles fine, although the value might be wrong. Maybe the compiler prints a warning, but not on all warning levels and this is not guaranteed.

> *Enable all warnings and stick to the C++ standard, i. e., use the compiler flags `-Wall -Wextra -pedantic`, optionally you may even set the flag `-Werror` to assert an error instead of warnings.*
>
> — **Principle**

With `C++11` the compiler added the *uniform initialization* using curly brackets, in order to raise an error instead of silently accepting the code, in case of narrowing conversion. This means, almost always use [C++11]

```
long l1{1234567890123}; // or
long l2 = {1234567890123};
```

Some examples of narrowing conversions:

```
int i1 = 3.14;     // initializes to 3, no error
int i2 = {3.14};   // Narrowing ERROR: fractional part lost

unsigned u1 = -3; // initializes to largest possible unsigned number
unsigned u2{-3};  // Narrowing ERROR: no negative values

float f1 = {3.14} // ok. initializes to float number closest to 3.14

double d = 3.14;
float f2 = {d};   // Narrowing ERROR. Possible lost of accuracy

unsigned u3 = {3};
int      i3 = {2};
unsigned u4 = {i3}; // Narrowing ERROR: no negative values
int      i4 = {u3}; // Narrowing ERROR: no all values
```

Remark 15. *The curly braces, i. e., uniform initialization, also works with the automatic type deduction `auto`. But be careful! The meaning of the curly braces has changed in `C++17` and also before results sometimes in a type different from what you would expect.* [C++17]

```
auto x1 = {42}; // C++14 x1 is of type std::initializer_list<int>
auto x2 = {42}; // C++17 x2 is of type std::initializer_list<int>

auto x3{42};    // C++14: x3 is of type std::initializer_list<int>
auto x4{42};    // C++17: x4 is of type int
```

Definition 2. For floating point values we call a conversion to a smaller data type (e. g., `double -> float`) a *floating-point conversion* (with possibly loss of precision) and otherwise a *floating-point promotion* (represent the value exactly with the larger type).

Remark 16. *Note, in floating-point conversion, if a value of* `T1 > T2` *is between two floating point number of* `T2`*, the rounding to a one of the both values is* implementation defined *and might be controlled with some intrinsic functions. If the value is out of range of* `T2` *the behavior is* undefined*.*

2.4.6 Constants

An important aspect of programming languages is to control the access to data. A data-type with the property `const` is called a *constant* and is immutable. The syntax to declare a constant is

`TYPE const VARNAME = VALUE;`

The `const` could also be on the left of the `TYPE`, but as a rule of thumb put the qualifier `const` on the right of what should be constant. The compiler will assert an error if you try to modify a constant object.

Example:

```
int n1 = 0;           // non-const object
int const n2 = 0;     // const object
const int n3 = 0;     // const object (same as n2)

n1 = 1;  // OK: mutable object
n2 = 2;  // ERROR: non-mutable object
```

Constants can be defined using automatic type deduction. Therefore, the keyword `const` simply qualifies the placeholder `auto`:

```
auto i1 = 7;          // mutable variable
auto const i2 = 8;    // const integer variable initialized with 8
const auto d1 = 2.0;  // const double variable initialized with 2.0
```

constexpr specifier

There is another qualifier that is stronger than `const`: The `constexpr` specifier declares that it is possible to evaluate the value of the variable at compile time. Such variables can then be used where only compile time constant expressions are allowed. A `constexpr` specifier used in an object declaration implies `const`.

A `constexpr` variable must satisfy the following requirements:

- its type must be a *LiteralType*.
- it must be immediately initialized
- the full-expression of its initialization, including all implicit conversions, constructors calls, etc, must be a constant expression

The category *LiteralType* cannot yet be fully explained, but especially the fundamental types discussed above are *LiteralTypes*.

Remark 17. *`constexpr` variables, expressions and functions are a powerful tool within C++, available since `C++11` and extended in `C++14` and `C++17`. In the chapter* Meta programming, *we will see how to use `constexpr` (functions) as a language within C++ to force the compiler to do computations for us.*

2.4.7 Scopes

Each name that appears in a C++ program is only valid in some possibly discontinuous portion of the source code called its scope. Thus, scopes determine the lifetime and visibility of (non-static) variables and constants. There are different types of scopes, global scope, function scope, class scope, block scope, function parameter scope, namespace scope, ... Typically, scopes are blocks of code surrounded by curly braces, except for the global scope that lives outside of functions and classes.

A local variable is declared within a block of a function. Its visibility and accessability is limited to within the { - }-enclosed block of its declaration. More precisely, the scope of the variable begins at the point of declaration and ends at the end of the block.

```
int main()
{                     // begin of the function block
  double pi = 3.14;   // begin of the variable scope
  std::cout << pi;
}                     // end of variable scope and function block
```

There might be nested blocks within other blocks that can limit the visibility of names declared inside this block:

```
int main()
{                     // begin of the function block
  {                   // begin of an inner block scope
    double pi = 3.14; // begin of the variable scope
  }                   // end of variable and block scope
  std::cout << pi;    // ERROR: pi is out of scope
}                     // end of variable scope and function block
```

Do not put variables in the global scope, i.e., do not use global variables!
— **Principle**

Hiding

In each scope a name can be defined only once (one-definition rule), but in another scope (even nested) the same name can be used to declare a new variable hiding the outer one with lifetime only in that nested scope.

scope.cc

```
int x = 11;   // (0) global variable x

void f() {    // function scope
  int x;      // (1) local x hiding global x
  x = 1;      // assignment to local x (1)
  {
      int x;    // (2) hides local x (1)
      x = 2;    // assignment to local x (2)
    }
    x = 3;      // assignment to local x (1)
  }

  void f2() {  // function scope
    int y = x; // use global x
    int x = 1; // (3) hides the global x
    ::x = 2;    // assignment to global x
    y = x;      // use local x (3)
    x = 2;      // assignment to local x (3)
  }
```

2.5 Library types

With the fundamental types and some language constructs we will learn later, you can already build powerful C++ programs. But it is possible to define your own (lass-)types for more complex data-structures, like vectors, tuples, lists, associative maps and sets, and so on. The standard library already defines some of these types thus allows to easily write more advanced programs.

Here, I just give an overview about some useful data-structures, later we will look more deeply into the standard library.

2.5.1 Strings

Character sequences were already mentioned at the beginning, when discussing the arguments of the function `main(int, char**)`. This functions accepts a low-level form of arrays of strings its as second argument. But these low-level strings are hard to use correctly. Thus, the standard library defines the type `std::string` instead:

```
#include <iostream> // for std::cout, std::endl
#include <string>
int main()
{
  std::string text = "This is a long text";
  std::cout << "length = " << text.size() << std::endl;
}
```

2.5.2 Container

Sequence Container

In the header `<vector>` the standard library provides a contiguous resizable vector container with flexible element types:

```
#include <cassert>
#include <vector>
int main()
{
    // vector of 3 doubles
    std::vector<double> x(3);
    x[0] = 0.0; x[1] = 1.0; x[2] = 2.0;

    // direct initialization with values
    std::vector<double> y = {1.0, 2.0, 3.0};
    std::vector y2 = {1.0, 2.0, 3.0}; // since c++17

    // add a new entry at the end of the vector
    x.push_back(3.0);

    // resize the vector to the specified size
    y.resize(4);

    assert(x.size() == y.size());
  }
```

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays (see below), because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function.

Remark 18. *The `main()` function needs two argument to represent an array of strings, the size and the address. Instead, one could store this in a vector of strings:*

```
  int main(int argc, char** argv) {
    std::vector<std::string> args(argv, argv + argc);
    // nr of arguments = args.size()
    // i-th argument = args[i]
  }
```

There is even a proposal for a future c++ standard to add a signature of the main function providing (something like) strings.

Associative Container

Apart from the *sequence container* `std::vector`, there is also the associative container `std::map<key_type, value_type>`, associating a value to a key:

```
  #include <map>
  int main()
  {
    std::map<int, double> m; // mapping int -> double
    m[17] = 42.0              // new association is created on access

    // test whether the key 42 is found
    assert(m.count(42) == 1);
  }
```

This can be combined with vector and string:

```
  #include <map>
  #include <string>
  #include <vector>
  int main()
```

```
{
  std::map<std::string, std::vector<int>> m; // mapping string ->
  ↪  vector<int>
  m["Hello"] = {1,2,3,4,5};
}
```

Remark 19. *Note that maps need more memory and the access is much slower than for a vector. So, if you have an integer key and know the min and max index, prefer a vector over a map, or just use the map as intermediate container to build up the container.*

2.5.3 Iterating over container

All standard containers can be traversed using *range-based for loops*

```
#include <map>
#include <string>
#include <vector>
int main()
{
  std::map<int, double> m; // fill up the map
  std::vector<double> v; // fill up the vector

  for (auto i : m)
    std::cout << i.first << ", " << i.second << std::endl; // (key, value)
    ↪  pair, see below

  for (double d : v)
    std::cout << d << std::endl;
}
```

If you don't know the type of the elements in traversal or it is complicated to write (like for `std::map`), use (qualified) `auto` instead.

2.5.4 Tuples

A vector represents a tuple of values of the same type. If the type should be different for each element, one could use a `std::tuple` instead. There, the types of all elements must be given explicitly:

```
std::tuple<int,double,float> t = {1, 2.0, 3.0f};
std::tuple t2 = {1, 2.0, 3.0f}; // c++17
```

To access an entry in a tuple, one cannot use the classical bracket operator `[]` as for vectors, but has to call a function instead:

```
double t1 = std::get<1>(t);
```

A special tuple is a pair. It consists of just two elements:

```
std::pair<int,double> p = {1, 2.0};
std::pair p2 = {1, 2.0}; // c++17
```

Here, the elements can be accessed again using `std::get`, but have also an explicit name:

```
int p0 = p.first;
double p1 = p.second;
```

Tuples can be used to return multiple values from a function (see below) and to assign multiple values to a set of variables:

```
// create a tuple from values
auto t = std::tuple{0, 1.0, 2.0f};

// assign the tuple entries to variables
int t0;
double t1;
float t2; // not used
std::tie(t0,t1,std::ignore) = t;
```

where `std::ignore` is an object of unspecified type such that any value can be assigned to it with no effect, and `std::tie` is a function that takes references to variables and assigns the value of a tuple to it.

Structured Binding

We have seen the effect of `std::tie` in the last section. There, we had to declare the variables before we can assign values to it and we have to know the types explicitly. With `C++17` this can be combined with `auto` to create new variables with types deduced from the tuple elements. This is called **structured binding**: [C++17]

```
auto [t0,t1,t2] = t;
```

Again, as for classical `auto` type deduction, the variable declaration can be extended by the const qualifiers:

```
auto const [t0,t1,t2] = t;
```

This structured binding does not only work for tuple-like structures, but also for structs:

```
std::pair<int, std::string> p {1, "pair"};
auto [first,second] = p;
assert(first == p.first && second == p.second);

Point point {0.0, 4.0};
auto [x,y] = point;
assert(x == point.x && y == point.y);
```

2.5.5 Iterating over associative containers

Recapitulate the iteration example:

```
#include <map>
#include <string>
int main()
{
  std::map<int, double> m; // fill up the map

  for (auto i : m)
    std::cout << i.first << ", " << i.second << std::endl; // (key, value)
    ↪  pair
}
```

In the iteration we get a pair `(key, value)`. This can be split up automatically using structured binding:

```
for (auto [key,value] i : m)
  std::cout << key << ", " << value << std::endl;
```

2.5.6 Iterating over tuples

Tuples (and pairs) cannot be traversed like other containers. The reason is that each element in a tuple has a different type and in a loop the elements must have the same type in each iteration. Currently the standard committee discusses and extended version of a loop, like `for...(auto t : tuple)` or `for constexpr(auto t : tuple)`. But this is not yet decided. We will see later in the chapter about meta-programming how to write a loop over tuples yourself. Then you get something like

```
forEach(tuple, [](auto t) {
  std::cout << t << std::endl;
});
```

that looks quite similar to a regular loop but works with tuples and pairs and many more.

In `C++17` some utility functions for tuples are introduced. An example is `std::apply` to apply an function to each entry of a tuple. This can be used to emulate a `forEach` loop:

```
std::apply([](auto... t) {
  ((std::cout << t << std::endl), ...);
}, tuple);
```

This uses advanced feature like lambda expressions, variadic templates and fold expressions.

2.6 Operators

We have see already in the initial example the usage of arithmetic and many other *operators*. Operators are special functions with typically prefix-notation (for unary operators) and infix notation (for binary operators) and are written with the classical mathematical symbols. In the initial example, we had `+, -, *, ::, ., <<, = , ",", ()`.

Definition 3. We call operators acting on one operand *unary operators* (like `-, *, &`), acting on two operands *binary operator* (like `+, -, *, /, =, +=, ...`) and even acting on three operands a *ternary operators* (there is only `?:`).

Operators are characterized by its properties: *associativity*, *precedence*, and whether they can be *overloaded*.

2.6.1 Associativity

Definition 4. Operator associativity means the property of an operator determining the order in which to evaluate multiple occurrences of this operator in an expression without changing the result of the evaluation.

Only in *associative compositions* the result depends on the associativity. In order to fix the meaning of expressions with multiple similar operators without brackets we introduce the convention

- A *left-associative operator* is evaluated from left to right.
- A *right-associative operator* is evaluated from right to left.

All (binary) operators are classified as left or right associative

Example 1. *Binary arithmetic operations are* left-associative*: (`+, -, *, /, %, <, >, &&, ||`)*

`a + b + c` is equivalent to `(a + b) + c`

Example 2. *Assignment operators are* right-associative*: (`=, +=, <<=, ...`), i. e., the expression*

`x= y= z` is evaluated from right to left and thus equivalent to: `x= (y= z)`

2.6.2 Precedence

Operators are ordered by its precedence. This defines a *partial ordering* of the operators and specified which operators is evaluated first. The precedence can be overridden using brackets.

From basic arithmetics you know the precedence of some arithmetic operators, especially `+, -, *, /`. This precedence is known in some countries as mnemonics:

- Germany: *Punktrechnung geht vor Strichrechnung*, meaning Multiplication/Division before Addition/Subtraction
- United States: **PEMDAS**: Parentheses, Exponents, Multiplication/Division, Addition/Subtraction. PEMDAS is often expanded to the mnemonic "*Please Excuse My Dear Aunt Sally*".
- Canada and New Zealand: **BEDMAS**: Brackets, Exponents, Division/Multiplication, Addition/Subtraction.
- UK, Pakistan, India, Bangladesh and Australia and some other English-speaking countries: **BODMAS**: Brackets, Order, Division/Multiplication, Addition/Subtraction or Brackets, Of/Division/Multiplication, Addition/Subtraction.

Example 3. *An example with classical mathematical operators on integers:*

$$x = 2 * 2 + 2\,/\,2 - 2 \quad\Rightarrow\quad x = (((2 * 2) + (2/2)) - 2) = 3$$
$$y = 8/2 * (2 + 2) \quad\Rightarrow\quad y = (8/2) * (2 + 2) = 16$$

Remark 20. *The operator `^` does not mean exponentiation or power, but the logical X-Or. In Matlab/Octave this operator has the expected meaning, but in C++ the operator has a different precedence than the power operator would have from a mathematical perspective.*

This means:

`a = b^2 + c`

is equivalent to

`a = b^(2 + c)`

BUT NOT TO

`a = (b^2) + c`

There is no power operator in C++! You have to call the function `std::pow` instead.

When in doubt or to clarify the expression: use parentheses.

— **Principle**

2.6.3 Examples of operators

In the table below, the operators are ordered by precedence and information about associativity is given. Here I give you a summary of most of the operators and its meaning. Most of the operators are defined for arithmetic types (integers or floating-point numbers), but some are also defined for library types, like `std::string`, `std::vector` and others.

Arithmetic operators

Operator	Action
`-`	Subtraction (unary minus)
`+`	Addition (unary plus)
`*`	Multiplication
`/`	Division
`%`	Modulo = Reminder of division, i. e., for integers: if `r = a % b`, then there exists `c`, such that `a=b*c`
`--`	Decrement (Pre- and Postfix), i. e., `--a` is equivalent to `a = a - 1`
`++`	Increment (Pre- and Postfix), i. e., `++a` is equivalent to `a = a + 1`

```cpp
int operator++(int& a, int) { // a++
  int r = a;
  a += 1;
  return r;
}
int& operator++(int& a) { // ++a
  a += 1
  return a;
}
```

Boolean operators

Logical operators and comparison operators

Operator	Action
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`==`	equal to
`!=`	unequal to
`&&`	AND
`||`	OR
`!`	NOT

Remark 21. *The result of a boolean expression is a* `bool` *value, e. g.,*

`bool out_of_bound = x < min_x || x > max_x`

[C++20] **Remark 22.** *With C++20 a new comparison operator is introduced, called* three-way comparison operator *or sometimes also* space-ship operator*. It is written as* `<=>` *and returns an object such that*

- *`(a <=> b) < 0` if `lhs < rhs`*
- *`(a <=> b) > 0` if `lhs > rhs`*

- *`(a <=> b) == 0` if `lhs` and `rhs` are equal/equivalent.*

and the returned object indicates the type of ordering (strong ordering, partial ordering, weak equality, ...).

Bitwise operators

Modify or test for the bits of integers

Operator	Action
&	AND
|	OR
^	exclusive OR
~	1-complement
>>	right-shift of the bits
<<	left-shift of the bits

Remark 23. *The logical operators << and >> are used in C++ often in a different context, namely to* shift *something into a* stream*. Streams are abstractions devices allowing input and output operations. The operator << is therefore called* insertion operator *and is used with output streams, whereas the operator >> is called* extraction operator *and is used with input streams.*

Example 4. *While a modern CPU efficiently implements arithmetic and logical operators on integers, one could implement those manually, by using bitwise operations and comparison with 0 only. The following pseudo code implements multiplication of two integers `a` and `b` with bit shifts, comparison and addition:*

```
c := 0
solange b <> 0
    falls (b und 1) <> 0
        c := c + a
    schiebe a um 1 nach links
    schiebe b um 1 nach rechts
return c
```

It implements kind of a manual multiplication in the binary base but in the uncommon order from right to left. [1]

As an exercise, you could implement this algorithm and compare the result with the classical multiplication.

Assignment operators

Compound-assignment operators like `+=`, `-=`, `*=`, `/=`, `%=` `>>=`, `<<=`, `&=`, `^=`, `|=` apply an operator to the left and right-side of an assignment, and store the result in the left operand.

Example:

```
a += b  // corresponds to  a = a + b
```

Remark 24. *Compared to the simple assignment, a compound-assignment does not need to create a temporary and maybe copy it to the left-hand side operand, but works with the operand directly.*

[1] See https://de.wikipedia.org/wiki/Bitweiser_Operator

```
struct A { double value; };

A operator+(A const& a, A const& b) {
  A c(a); // create local copy of a
  c += b;
  return c;
}
```

but

```
A& operator+=(A& a, A const& b) {
  a.value += b.value;
  return a;
}
```

Bracket-Access operators

The round brackets `()` and the square brackets `[]` also represent operators, namely *bracket-access operators*. While the square bracket is typically used as vector access and allows only one argument, e. g., the vector element index, the round brackets represent a function call and thus allow any number of arguments `>= 0`.

Sequence of evaluation and side effects

A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed.

A side effect is, for example, the change of a variable not directly involved in the computation.

[C++17] Logical and-then/or-else (`&&` and `||`) operators, ternary `?:` question mark operators, and commas `,` create sequence points; `+`, `-`, `<<` and so on do not! With `C++17` several more operations are now sequenced and are safe to include side effects. But in general, you should know that

> *When you use an expression with side effects multiple times in the absence of sequence points, the resulting behavior is* ***undefined*** *in C++. Any result is possible, including one that does not make logical sense.*
>
> — **Attention**

See also: Wikipedia, CppReference.com

For logical operators, at first the left operand is evaluated and based on its value it is decided whether to evaluate the right operand at all or not, i. e.,

```
A && B        // first evaluate A, if A == true evaluate B and return its
↪  value
A || B        // first evaluate A, if A == false, evaluate B and return its
↪  value
A ? B : C     // first evaluate A, if true, evaluate B, otherwise evaluate
↪  C
(A,B,C, ...)  // first evaluate A, then evaluate B, then evaluate C, ...
f() + g()     // it is not specified whether f is evaluated first or g
```

Never rely on a function call in a logical expression.

— **Principle**

Remark 25. *The precedence property is not related to the order of evaluation, but to the rank of the operator. For an arbitrary function `f(a,b,c,d)` it is not specified in which order the expressions for a, b, c, d are evaluated.*

Example 5. *The following expressions have different behavior:*

```
int foo(int a, int b) { return a + b; }

int x = 1;
foo(++x, x++); // Variant 1 (unspecified behavior)

x = 1;
int a = ++x, b = x++;
foo(a, b); // Variant 2 (behavior defined)
```

2.6.4 Operators as functions

In C++ nearly all operator can be written (and called) as a regular function. Let `o` be the symbol of the operator, e. g., `o` ∈{`+`,`*`,`()`,`+=`,`<`,`...`}, then there zero, one or both of the following implementations are available:

```
Result operator o(Arg1 a, Arg2 b, ...) // a o b
Result Arg1::operator o(Arg2 b, ...) // Arg1 a; a o b
```

The variant 1 implement the operator as a free function, whereas the variant 2 implements the operator as a member function of a class, where `Arg1` is the name of that class. Whether there is a function representation of the operator and whether it is allowed to *overload* that function is specified in the table below.

Overview

Precedence	Operator	Description	Associativity	Overload
1 = highest	`::`	Scope resolution	Left-to-right	—
2	`++ --`	Suffix/postfix increment and decrement	Left-to-right	✓ ✓
	`()`	Function call		(C)
	`[]`	Subscript		(C)
	`. ->`	Member access		— (C)
3	`++ --`	Prefix increment and decrement	Right-to-left	✓ ✓
	`+ -`	Unary plus and minus		✓ ✓
	`! ~`	Logical NOT and bitwise NOT		✓ ✓
	`*`	Indirection (dereference)		✓
	`&`	Address-of		✓
4	`.* ->*`	Pointer-to-member	Left-to-right	— ✓
5	`* / %`	Multiplication, division, and remainder		✓ ✓ ✓
6	`+ -`	Addition and subtraction		✓ ✓
7	`<< >>`	Bitwise left shift and right shift		✓ ✓
8	`<=>`	Three-way comparison operator (`C++20`)		✓
9	`< <=`	For relational operators $<$ and $\leq$ respectively		✓ ✓
	`> >=`	For relational operators $>$ and $\geq$ respectively		✓ ✓
10	`== !=`	For relational operators $=$ and $\neq$ respectively		✓ ✓
11	`&`	Bitwise AND		✓
12	`^`	Bitwise XOR (exclusive or)		✓
13	`|`	Bitwise OR (inclusive or)		✓
14	`&&`	Logical AND		✓
15	`||`	Logical OR		✓
16	`?:`	Ternary conditional	Right-to-left	—
	`=`	Direct assignment		(C)
	`#=`	Compound assignment operators [note 1]		✓
17 = lowest	`,`	Comma	Left-to-right	✓

[note 1]: $\# \in \{$`+, -, *, /, %, <<, >>, &, ^, |`$\}$

In the column *Overload* means (C), the operator can only be implemented as member function of a class, whereas ✓ allows to write the operator as member function or free function.

A more complete table can also be found at `http://en.wikipedia.org/w/index.php?title=C++_operators`, or `https://en.cppreference.com/w/cpp/language/operator_precedence`.

2.7 Functions

A function declaration consists (in the simplest case) of a return type, the function name and comma separated list of argument types (zero or more):

```
ReturnType function_name(Type1 arg1, Type2 arg2,...);
```

A declaration (or prototype) introduces this name into the current scope and tells the compiler that you intend to define it. The implementation of the body of the function makes a declaration to a definition. Both can be combined or separated. You can even provide the definition in a separate source file, or even link it dynamically at runtime in your program.

Functions that do not return a value have the return type `void` — a fundamental type representing the empty set of values.

Remark 26. *With `C++11` an alternate syntax for the function declaration is introduced, utilizing the `auto` keyword:* [C++11]

```
auto function_name(Type1 arg1, Type2 arg2,...) -> ReturnType;
```

It's called a function with trailing return type. Here the return type might directly depend on the arguments (argument types not values).

With `C++14` one can even omit the trailing type: [C++14]

```
auto function_name(Type1 arg1, Type2 arg2,...) { ... };
```

Here, the declaration must be a definition, since the actual return type is deduced from the `return` expression of the function.

2.7.1 Argument passing and return values

There are essentially two ways how to pass values/arguments to functions:

pass-by-value The passed values initialized a new local variable, that is destroyed at the end of the function body:

```
void foo(int i) {
  i = 5; // the local variable i is changed
}

int main() {
  int j = 4;
  foo(j);     // j is not changed
  return 0;
}
```

pass-by-reference Arguments are references to the passed variables and build a local alias of that:

```
void foo(int& i) {
  i = 5; // changes the alias i, thus changing the passed variable j
}
int main() {
  int j = 4;
  foo(j);     // j is changed by the function foo
  return 0;
}
```

pass-by-const-reference Similar to *pass-by-reference*, but does not allow to change the value of the alias:

```
void foo(int const& i) {
  /* i = 5; */ // a change of the local reference i is not allowed
}

// or
void foo2(const int& i) {...}

int main() {
  int j = 4;
  foo(j);     // j cannot be changed by foo
  return 0;
}
```

Remark 27. *For small types `T`, e. g., `int`, `double`, ..., it is recommended to simple pass-by-value. Larger objects should be passed by reference instead, to reduce the cost of copies:*

1) *If the function intends to change the argument as a side effect, take it* by non-const reference.
2) *If the function doesn't modify its argument and the argument is of primitive type, take it* by value. *("Primitive" basically means small data types that are a few bytes long and aren't polymorphic (iterators, function objects, etc...))*
3) *Otherwise take it* by const reference, *except in the following case:*
    - *If the function would then need to make a copy of the const reference anyway, take it* by value.

Example 6. *Assume, you have a type `A` that implements a `+=` operator. In order to implement the plus operator, one can use that. Two implementations are possible:*

```
// instead of
A operator+(A const& a, A const& b) {
  A tmp = a; // create a copy of a
  return tmp += b;
}

// write:
A operator+(A a, A const& b) {
  return a += b;
}
```

The plus operator can in most cases be implemented as `+=` and this should be the way to go! Don't repeat yourself.

> *Make all function arguments `const` by default, except when intending to change its value.*
>
> — **Principle**

Default values

Arguments can be given a default value, that is taken if the argument is not passed to the function. Since parameters are passed in order, default values must be set to arguments starting from the last one.

Example 7. *The following example defines a function with 3 arguments, where the last two are optional:*

```
void foo(int a, double b = 2.0, char c = 'c') { /* ... */ }

int main()
{
  foo(1); // OK
  foo(1, 5.0); // OK
  foo(1, 5.0, 'x'); // OK
}
```

Remark 28. *One cannot directly set a default value for the first argument and no default value for the second argument. If a default is set, for argument n, all following arguments also must have a default value.*

An alternative to these default values, are optional *values. This is a library extension in C++17, where the actual default value is set when you use it:* [C++17]

```
#include <optional>
void foo(std::optional<int> a_, double b, char c)
{
  int a = a_.value_or(1);
  /* ... */
}

int main()
{
  foo(1, 5.0, 'x'); // OK
  foo(1, 5.0); // ERROR, char c not passed to function
  foo(std::nullopt, 5.0, 'x'); // OK, a = 1
  foo({}, 5.0, 'x'); // OK, same as above
}
```

Return values

If a function has a return type unequal to `void`, it needs a `return` statement, unless it never returns. Void-functions can omit this statement, but are allowed to call an empty return:

```
int foo() {
  return 42;
}

void bar() {
  return;
}
```

It is allowed to have multiple return statements, but there is only one return type that is fixed by the function declaration. Thus all return statements must return the same type (or at least all return values are implicitly cast to the return type).

> *Make sure, that a function has a call to* `return` *in every execution path.*
> — **Principle**

For the return types it is similar to the argument passing. You can return by value, or return by reference (where for the latter case there are some pitfalls, see below). But, you can also return a value by providing an additional *output argument*, e. g.,

```
void function_name(Type1 arg1, Type2 arg, ..., OutputType& output, ...) { ...
↪  }
```

where the order of the function arguments does not matter, i. e., you could also start with the output argument. The important aspect is that output arguments must be passed by (non-const) reference. The intend is that you create the output variable outside of the function and pass it by reference to the function where it is filled with values.

Remark 29. *There is no only-output parameter type in C++, you can always read from the output variable. This is a source of errors so that some people prefer not to use output* *arguments, or use pointer arguments exclusively for this purpose, so that access to the value must be done explicitly.*

A common mistake is to return a reference to a local variable of the function:

```
std::vector<int>& foo(int input) {
  std::vector<int> temp; // local variable
  // ...
  return temp; // PROBLEM!!!
}
```

At the end of the function scope, the local variable is destroyed and thus the reference is referring to some destroyed memory. This is also called a *dangling reference* and results in an error that is very hard to detect. So, don't do that!

If you return by reference, make sure the object you return outlives the lifetime of the function.

Positive examples:

```
int& foo1(int& input) {
  return input; // input references an existing object living outside of the
  ↪   function
}
int& foo2() {
  static int i = 5; // a static variable is not destroyed at the end of the
  ↪   function scope
  return i;
}
```

Remark 30. *Using* `auto` *return type might lead to unexpected behavior if combined with references. The type that is returned is the raw type of the expression in the return statement, removing all top-level const and reference qualifiers:*

```
auto           f0(int i) { return i; }  // -> int
auto           f1(int& i){ return i; }  // -> int
auto&          f2(int& i){ return i; }  // -> int&
```

[C++14] *The new language keyword* `decltype(auto)` *is introduced, to overcome the problem that returning a reference does not return a reference:*

```
decltype(auto) f3(int i) { return i; }  // -> int
decltype(auto) f4(int& i){ return i; }  // -> int&
```

It is a placeholder type, as `auto`*, but does not remove top-level const and references.*

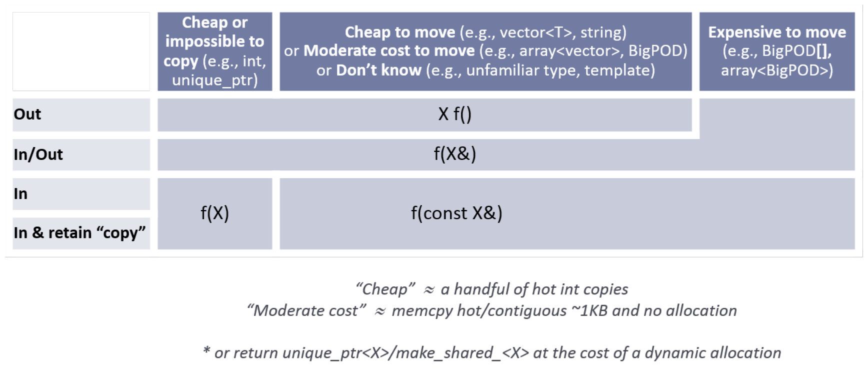
Guidelines

The CppCoreGuidelines provide a guideline how to pass parameters and how to return values, depending on the type of the arguments:

Return value optimization / Copy elision

Returning a local value created inside the function by value means naively the same as in argument passing: when the returned value is assigned to a new variable, this is created as copy of that return value.

This is not completely true. Instead of the procedure 1. create / allocate space for the local variable, 2. create the target variable, 3. copy the function return from the local variable



	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

* or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

Figure 2.1: From CppCoreGuidelines secion F.15

to the target variable, 4. destroy the local variable, the compiler is allowed to use the fact that the local variable is destroyed at the end of the function, so not used any more, and the target variable is created only to hold the returned function value. Instead of allocating two variables, it allocates only one and works in both cases with this variable. With `C++17` this behavior is guaranteed by the standard. Before 2017 standard, it was a compiler optimization. [`C++17`]

Example 1:

```
double f() { return 42.0; }
int main()
{
  double x = f();
  // translates to
  double x = 42.0;
}
```

Example 2:

```
double f()
{
  double y = 42;
  return y; // copy is not guaranteed to be elided, but a common
  ↪   optimization
}

int main()
{
  double x = f();
  // may translate to (named return value optimization)
  double x = 42.0;
}
```

Some people prefer the naming *deferred temporary materialization*, i. e., the materialization of the value 42 happens in the `main()` when assigned to the variable `x`.

This optimization or guarantee is especially important if you want to return large objects created inside of the function. It is thus guaranteed that no expensive copy operation must be performed (not even a move operation). The created objects simply materializes outside of the function in the target variable.

2.7.2 Signature

The signature of the function is used to select a function to call. For simple functions, the signature consists of

- function name,
- parameter-type-list,
- enclosing namespace (if any), and
- trailing requires-clause (if any) [C++20]

where the *parameter-type-list* specifies the number and type of each individual parameter. Also qualifiers like `const` and `&` are important in the to distinguish types in that list. It is also allowed to have an *ellipsis*, i. e., `...` at the end of the parameter list, to indicate a *variadic* function.

The *enclosing namespace* defines the named scope in which the function is defined in and the *trailing requires-clause* is related to concepts that the parameter types and function must fulfill. (we will discuss this only briefly in a later chapter of the lecture).

The signature + return type defines the type of the function.

Remark 31. *An empty parameter list is equivalent to a single parameter of type* `void`*. (This is typical in some C programs).*

2.7.3 Function overloading

In C++ it is allowed to have multiple functions with the same name (in the same scope). At least if it is possible to distinguish those function by their function signatures! The introduction of the same name multiple times is called *overloading*. Function overloading is used to implement the same algorithm for different argument types with the same function name.

Example 8. *Let's consider an example from the beginning:*

```
struct Point { double x, y; };

double distance(Point const& a, Point const& b)
{
  double dx = a.x - b.x;
  double dy = a.y - b.y;
  return std::sqrt(dx * dx + dy * dy);
}
```

The function `distance` *is defined for exactly one parameter-type-list, i. e., two arguments of type* `Point const&`*. But the only requirement we need in order to calculate the distance with this algorithm is that there are two data members* `.x` *and* `.y` *that are of some number type.*

If you want to calculate the distance of some other point type, the algorithm looks pretty much the same, so it would be logical to use the same function name for that algorithm.

```
struct DPoint { double x, y; };
struct FPoint { float x, y; };

double distance(DPoint const& a, DPoint const& b)
{
  double dx = a.x - b.x;
  double dy = a.y - b.y;
  return std::sqrt(dx * dx + dy * dy);
}

float distance(FPoint const& a, FPoint const& b)
{
  float dx = a.x - b.x;
  float dy = a.y - b.y;
  return std::sqrt(dx * dx + dy * dy);
}

int main()
{
  FPoint a, b;
  distance(a,b);
}
```

Remark 32. *In the example above there even the whole body of the function is very similar, except for the used data types. In the chapter* Generic Programming *we will look into this and instead of writing explicit functions, we will write a function template, parameterized with the value type.*

> *Functions implementing the same algorithm on different types should be named equal.*
> — **Principle**

In order to resolve a function call, i. e., to find the right function to evaluate, the compiler has to follow some common steps:

1. Look for all functions with the given function name (*name lookup*). All visible function (and names introduced by a `using` declaration/directive) are considered. Thereby the function namespace is important. If the argument types are defined in some namespace also this is considered a viable scope for lookup (*Argument Dependent Lookup, ADL*).
2. If there is more than one viable function found in the name lookup, the compiler tries to resolve the function overload to select a function to call (*overload resolution*). Therefore it is considered which parameter-type-list matches best the passed arguments.
3. All non-matching functions are eliminated from the list of viable candidates (e. g., wrong number of arguments or non-matching types). All remaining functions are ordered by the property which one is a "better" match for the arguments. At first position in that list is the exact match (if it exists).
4. Each pair of viable functions F_1 and F_2 is ranked by the implicit casts of its arguments, to decide the order. Thereby we have the rule: F_1 is determined to be a better function than F_2 if implicit conversions for all arguments of F_1 are not worse than the implicit conversions for all arguments of F_2, and there is at least one argument of F_1 whose implicit conversion is better than the corresponding implicit conversion for that argument of F_2. Therefore we have the three conversion ranks
   a) exact match: no conversion required, lvalue-to-rvalue conversion, qualification conversion (e. g., add `const`)
   b) Type promotion: integral promotion, floating-point promotion
   c) Type conversion: integral conversion, floating-point conversion, floating-integral conversion

Example 9. *In the following example the function* `foo()` *is overloaded 4 times:*

```
#include <iostream>
struct A { double x = 1.0; };

int foo(A a) { return 0; }                // (0)
int foo(A& a) { return 1; }               // (1)
int foo(A const& a) { return 2; }         // (2)
int foo(A const& a, int b) { return 3; } // (3)

int main() {
  A a;
  std::cout << foo(a) << "\n";
}
```

This overload-set generates a compiler error, since (0) is equivalent to (1) or (2), i. e., all three are exact matches. Removing (0) resolves this error. Then (1) is called, since `a` *is a mutable object and thus (1) represents an exact match and no const-qualification is needed. (3) is ignored because of the wrong number of arguments.*

Example 10. *The following example illustrates what it means to be a better candidate:*

```
void foo(int const&, short); // overload #1
void foo(int&, int);         // overload #2

int main() {
  int i;
  int const ic = 0;
  short s = 0;

  foo(i, 1L);  // 1st argument: i -> int& is better than i -> int const&
               // 2nd argument: 1L -> short and 1L -> int are equivalent
               // calls foo(int&, int)

  foo(i,'c');  // 1st argument: i -> int& is better than i -> int const&
               // 2nd argument: 'c' -> int is better than 'c' -> short
               // calls foo(int&, int)

  foo(i, s);   // 1st argument: i -> int& is better than i -> int const&
               // 2nd argument: s -> short is better than s -> int
               // no winner, compilation error

  foo(ic, 'c'); // 1st argument: ic -> int& not allowed, but ic -> int
  ↪   const& exact match
                // ---> #1 removed from list of viable functions
                // 2nd argument: 'c' -> short is allowed
                // calls foo(int const&, short);
}
```

Remark 33. *There are many more rules for the selection of the best matching function. A detailed explanation can be found on*

- https://accu.org/index.php/journals/268
- http://en.cppreference.com/w/cpp/language/overload_resolution

2.7.4 Inline Functions

An *inline*-function is a function qualified with the keyword `inline`. This has the following consequences:

- It is allowed to have multiple definitions of the same (inline)-functions in the same translation unit, as long as all definitions are identical. (Otherwise the behavior is undefined)
- The definition must be visible in the translation unit (declaration without definition is not allowed)

Further reading: http://en.cppreference.com/w/cpp/language/inline

Do not mix up the keyword `inline` with the compiler optimization *inline expansion/replacement*. There, the code of the called function is expanded at the position of the call instead of performing a jump to the separately defined function. For this to work, the compiler must see the definition of the function. Advantage of function inlining is the higher locality and the lack of a jump (and jump back). But this also may result in larger output code. The compiler decides automatically which function to inline. The keyword `inline` has only very little effect on this decision.

2.8 References and Pointers

Although used already, the references need a revisit. References can be understood as aliases to (existing) objects. Compared to classical pointers, they do not represent the address of the referenced objects, but the data of the objects directly. They can be seen as dereferenced pointers.

A reference variable can be declared like a regular variable, using additionally the reference qualifier `&`:

```
TYPE &  var1  = aliased_object; // (1)
TYPE && var2 = aliased_object; // (2) ... since C++11
```

Where (1) is called a lvalue-reference and (2) a rvalue-reference . A reference must be initialized directly as it is not possible to change which object the reference refers to. This is a major difference to pointers.

```
int i = 0;
int& r = i; // r references i
r = 1;      // changes the value of i => i == 1
i = 2;      // also r == 2
```

References itself are not objects and do not need own memory. That is why there are no arrays of references and no references of references!

```
int& a[3]; // error
int& &r;   // error
```

2.8.1 lvalues and rvalues / value categories

See https://en.cppreference.com/w/cpp/language/value_category:

Each C++ expression (an operator with its operands, a literal, a variable name, etc.) is characterized by two independent properties: a *type* and a *value category*. Each expression has some non-reference type, and each expression belongs to exactly one of the three primary value categories: *prvalue*, *xvalue*, and *lvalue*.

- a *glvalue* (“generalized” lvalue) is an expression whose evaluation determines the identity of an object, bit-field, or function;
- a *prvalue* (“pure” rvalue) is an expression whose evaluation either
  - computes the value of the operand of an operator (such prvalue has no result object), or
  - initializes an object or a bit-field (such prvalue is said to have a result object). All class and array prvalues have a result object even if it is discarded. In certain contexts, temporary materialization occurs to create a temporary as the result object;
- an *xvalue* (an “eXpiring” value) is a *glvalue* that denotes an object or bit-field whose resources can be reused;
- an *lvalue* (so-called, historically, because lvalues could appear on the left-hand side of an assignment expression) is a *glvalue* that is not an *xvalue*;
- an *rvalue* (so-called, historically, because rvalues could appear on the right-hand side of an assignment expression) is a *prvalue* or an *xvalue*.

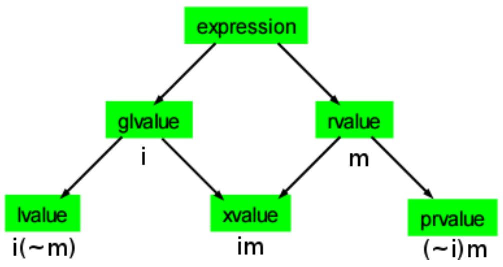


Figure 2.2: Diagram of value categories.

This categorization results from the idea that every expression can be characterized by two orthogonal properties:

i **has identity**: Object that has an address, a pointer, the user can determine whether 2 copies are identical

m **can be moved from**: We are allowed to leave the source of a “copy” in some indetermined, but valid, state.

∼ has not the property.

The combination `(∼i)(∼m)` does not really exists and thus, we have three leaf categories, see Figure 2.2.

Examples for lvalues, rvalues, and prvalues:

```
int i = 3;
int j = 4;

i;                // the name of a variable is an lvalue
123; bool; 'c'; // literals (except for string literals) are prvalues
```

```
// a + b, a % b, a & b, a << b, and all other built-in arithmetic expressions
↪   are prvalues

int c = i * j; // OK, (p)rvalue on the right-hand side of an assignment
i * j = 42;    // ERROR, rvalue not allowed on the left-hand side of an
↪   assignment

// more on lvalues:
i = 43;           // OK, i is an lvalue
int* p = &i;      // OK, i is an lvalue

int& foo();       // a function call, whose return type is lvalue reference
↪   is an lvalue
foo() = 42;       // OK, foo() is an lvalue
int* p1 = &foo(); // OK, foo() is an lvalue

// more on rvalues:
int foobar();        // a function call, whose return type is non-reference
↪   is an prvalue
int j = 0;
j = foobar();        // OK, foobar() is an rvalue
int* p2 = &foobar(); // ERROR, cannot take the address of an rvalue
j = 42;              // OK, 42 is an rvalue
```

2.8.2 lvalue- and rvalue references

We distinguish references to lvalues and rvalues as lvalue-references and rvalue-references, denoted by one or two `&`, respectively. So, lvalue-references refer to existing objects with an identity, i. e., lvalues, whereas rvalues typically refer to something that goes out of scope, whose lifetime ends. Rvalue references can be used to extend the lifetimes of temporary objects:

```
int f() { return 42; }
int& g(int& a) { return a;
↪  }

// f() is prvalue expr.
int i = f(); // OK
f()   = 7;   // ERROR

// g() is lvalue expr.
int j = g(i); // OK
g(i)  = 3;   // OK => i ==
↪  3
```

```
int&  k1 = f(); // ERROR
int&& k2 = f(); // OK
k2 += 1;        // OK, rvalue-Reference extends
↪  lifetime

int&  l1 = g(i);      // OK
int&& l2 = g(i);      // ERROR, cannot bind lvalue
↪  to
                      // rvalue-References.

// const-References
int const& k3 = f();  // OK
k3 += 1;              // ERROR, const-ref. is
↪  non-mutable
int const& l3 = g(i); // OK
```

In the example, the function `f()` only returns a temporary object (a number), that is not yet assigned to a variable. Rvalue-references can now be used to refer to this value with a variable before it is finally destroyed.

Example 11. *Example from Stackoverflow:*

```
T&  lvalue();
T   prvalue();
T&& xvalue();

T&& does_not_compile = lvalue();
T&& well_behaved = prvalue();
T&& problematic = xvalue();
```

Remark 34. *The ultimate usage of rvalue-references is, that you know that the object refereed to is going out of scope or is deleted afterwards, or is a temporary at all, thus one can directly take the ownership of that object, without copying its content. This is called a move operation.*

2.8.3 Pointers

Referencing something in memory could be understood as just storing the address of that memory. An address is something like an integer indicating the position in memory. But, we need more than just the address, we need the type of the data that we are referring to, so that we can give the memory a meaning. This is called a pointer:

```
TYPE * pointer = &OBJECT;
```

where `OBJECT` is any lvalue-reference type. We have used the `&` symbol before, to qualify/indicate a reference. In the context of pointers, this symbol is an (unary) operator, the address-of operator, returning a pointer to the object in memory. A pointer type is indicated by the `*` qualifier.

The *dual* operator to the address-of operator is the de-reference operator, giving a reference to the object, a pointer points to:

```
TYPE & reference = *POINTER;
```

where `POINTER` must be any object of pointer type. So, the dereferenced pointer is again an lvalue.

Pointers can be combined with `const` qualifiers, to either indicate that the addressed data is immutable or that the address value is constant. This behaviour is determined by the position of the `const` qualifier, i. e.,

```
int data = 42, data2 = 1234;

int const* i1 = &data;
*i1 = 7;                  // ERROR: cannot modify constant data
i1 = &data2;              // OK: i1 now points to data2

int* const i2 = &data;
*i2 = 7;                  // OK: data is not const
i2 = &data2;              // ERROR: cannot change address of const pointer

int const* const i3 = &data; // completely imutable
```

Since references are not objects, one cannot declare a pointer to a reference (something like `int&*`), but a pointer is a regular type representing an address, it can be referenced, i. e.,

```
int data = 42, data2 = 1234;
int* p = &data;
int*& r = p;  // reference to pointer p
```

```
*r = 7;       // changes value of data;
r = &data2;   // changes stored address of p
```

Don't use pointers to pass data to functions, only if you have a strong reason to do so. Some style-guides use pointers as output arguments in functions to not mix it up with input arguments. This is because you have to explicitly dereference the pointer in order to use it.

— **Guideline**

2.9 Arrays and Dynamic Memory

2.9.1 Static Arrays

So far, we had only data-types with 1 entry per variable (except for the library types `vector`, `pair` and `tuple`). Arrays are compound data-types with a fixed number of entries of the same type:

`TYPE variable[ SIZE ];`

where `SIZE` is an *integral constant expression*.

Examples:

```
int n[5];
double f[ 10 ];
constexpr int len = 32;
char str[len];
```

Those arrays can then be accessed by square-brackets, similar to `std::vector`, e. g.,

`n[2] = 7;`

Arrays can be pre-initialized. In that case, the array size can be omitted, i. e., is detected automatically by the compiler:

```
int n1[5]  = {1, 2, 3, 4, 5};
int n2[]  = {10, 9, 8}; // automatically size 3
```

Multi-dimensional arrays are written with multiple square brackets:

```
int A[2][3]; // a 2 × 3 matrix
double B[5][4][3]; // an 5 × 4 × 3 tensor
```

and can be understood as arrays of arrays (a matrix is and arrays of rows).

The initialization of multi-dimensional arrays is by lists of initializer-lists, e. g.,

```
double A[2][3] =
{
  {1.0, 2.0, 3.0},
  {4.0, 5.0, 6.0}
};
```

Remark 35. *An array can be pre-initialized with less elements than the given size. In that case the remaining entries are default-initialized, i. e., number types are initialized to zero and class-types using the default-constructor.*

Example:

```
int foo[5] {1,2,3}; // [1,2,3,0,0]
```

(The assignment operator "=" is here replaced by Universal Initialization.)

A multi-dimensional array can be written with a one-dimensional initializer-list. In that case, the initialization happens arrays-wise.

```
int foo[][2] {1,2,3,4}; // [1,2; 3,4]
```

For multi-dimensional arrays only one size entry can be omitted. The shape must be deducible by the number of entries.

Size of an array

While the syntax for arrays is simple and clean, it lacks the direct possibility to get the size after the declaration. In C, you typically use macros to deduce the size from the byte-size of the array, i. e.,

```
#define SIZE(a) (sizeof(a) / sizeof((a)[0]))
```

Note the usage of additional brackets, here.

Example:

```
  int vec[5] {1,2,3};
  static_assert(SIZE(vec) == 5);

  int mat[][2] {1,2,3,4};
  static_assert(SIZE(mat) == 4);
```

Arrays as function arguments

Arrays can be (implicitly) converted to a pointer, i. e., to a pointer to the first element of the array:

```
int a[3] = {1, 2, 3};
int* p = a;  // pointer to the first element of the array
```

This allows to pass arrays to functions, accepting pointers as arguments, i. e.,

```
  void f1(int* p) { /* sizeof(p) != sizeof(a) */ }
  void f2(int p[]) { /* sizeof(p) != sizeof(a) */ }

  int a[3] = {1, 2, 3};
  f1(a); // OK: a is converted to a pointer
  f2(a); // OK: equivalent to f1
```

But, then we loose any information about the array. It is just a pointer to the first element of that array.

Nevertheless, pointers to array data allow direct element access, like for the regular arrays:

```
int a1 = p[1];
```

The second form of passing arrays to functions is by array reference

```
TYPE (&reference)[SIZE] = ARRAY;
```

Example:

```
  void g(int (&b)[3]) { /* sizeof(b) == sizeof(a) */ }
```

```
int a[3] = {1, 2, 3};
g(a);
```

The problem here, we have to give the size in the function argument declaration, since the size of the arguments must be known at compile-time and are fixed to exactly one size. It is not allowed to pass arrays of different size to that function.

Remark 36. *Outlook: Using* templates*, we can write functions of arbitrary array size and can deduce that size automatically, i. e.,*

```
template <class T, int N>
int array_size(T (&)[N]) { return N; }

template <class T, int N1, int N2>
std::pair<int,int> mult_array_size(T (&)[N1][N2]) { return {N1, N2}; }
```

2.9.2 Standard Arrays

The problem with the size of the arrays, the argument passing to functions (and some more restriction) motivates to introduce a library type that overcomes those problems. With C++11 [C++11] the type `std::array<T, Size>` was added to the standard library:

```
#include <array>
// ...
std::array<int, 3> vec1 = { 1.0, 2.0, 3.0 };
std::array vec2 = {1, 2, 3}; // with C++17

std::cout << vec1[1];
```

The difference to the other library type `std::vector` is, that an array must have a fixed size, and the data is allocated to the "stack" and not dynamic memory allocated on the "heap" as in the vector implementation (see below).

Prefer the library type `std::array` over classical arrays.

— **Principle**

2.9.3 Dynamic memory allocation

We have seen that arrays can be implicitly converted to pointers and pointers can be used similar to arrays. Can we allocate an array directly as a pointer? Is it possible to have an array of dynamic size? One needs to initialize a pointer with the address of a memory block large enough to hold the wanted array. This is accomplished by *dynamic memory management*: memory of arbitrary size can be allocated and deallocated at runtime!

In C++ this is done with the operator `new` and `new[]` to allocate memory and construct objects in the newly allocated memory, and `delete` and `delete[]` to destruct the objects and deallocate memory.

For a single element:

```
TYPE * p = new TYPE;  // allocation of one element of type TYPE
delete p;             // deallocation of one element of type TYPE
```

and for more than one element:

```
TYPE * p = new TYPE[ SIZE ];
delete[] p;
```

Note, the `SIZE` parameter to `new[]` does not need to be a constant expression!

Examples:

```
char* s = new char[100];    // dynamic array of 100 elements
int n = 1024;
double* v = new double[n];  // dynamic size
float* f = new float;       // single element

for (int i = 0; i < n; ++i)
  v[i] = i*i;

*f = 1.41421356237;         // dereference the pointer to get a reference

delete[] v;   // new[] => delete[]
delete[] s;   // new[] => delete[]
delete f;     // new   => delete
```

One problem with dynamic arrays is that there is no way to get the size of that array, and there is no way for the compiler to perform a boundary check. Access of a memory location outside of the allocated array size may lead to a *segmentation fault* or may simply overwrite the data stored at the specific memory location → hard to find errors.

> *`delete` (or `delete[]`) can be applied to pointers created by the corresponding operator `new` (or `new[]`). On other pointers it may lead to undefined behavior (and often produces a runtime-error of the form "invalid pointer"). Only call `delete` once on a pointer!*
>
> — **Rule**

Another problem with dynamic arrays is that you cannot even see whether it is an allocated array, or just an uninitialized pointer variable. So, here again the general rule applies: initialize variables directly on declaration. If you do not yet have an address to initialize the pointer with, set it to zero, i. e.,

```
TYPE * pointer = nullptr;
```

where `nullptr` is a special object just for this case of initialization of pointers. After `delete`, reset the pointer to `nullptr` again.

Example:

```
int* p = nullptr;
p = new int[1234];
// ...
delete[] p;
p = nullptr;

if (p) {  // test for p != nullptr
  /* work with p */
}
```

Who owns the memory of a pointer, i. e., who is responsible for deleting that pointer? It is the one that has created the array, so do not forget to delete! Otherwise you have a *memory leak*. The aftermath of a pointer related bug, e. g., array boundary violation or accessing deleted memory, may show up much later than the actual position of the error.

Pointers are dangerous *and require careful programming. So, prefer library types like* ***std::vector*** *or* ***std::array*** *instead of (dynamic/static) arrays.*

— **Principle**

Pointer arithmetics

Pointers used as addresses of arrays, i. e., the address to the first element of the array, have their one arithmetic. Since addresses are like integers, they can be incremented and decremented, meaning a movement of the pointer in memory.

Pointers are associated to a specific element type (underlying type of the data). Incrementing and decrementing thus moves the address by multiples of the size of the underlying data. This means that `pointer + 1` points to the next element in an array, where `pointer` points to.

Examples:

```
int* vec = new int[10]; // vec points to the first element of the array
vec = vec + 1;          // vec now contains the address (vec + sizeof(int)),
↪   that is:
                        // vec points to the second element of the array
vec++;                  // vec points to the third element of the array...
delete[] vec;           // ERROR: Address of vec not allocated by a direct
↪   call to new[]
```

Pointers can be incremented (`+, ++`), or decremented (`-, --`), pointers can be compared, comparing the underlying address. Pointers can be dereferenced, giving a reference to the data it points to. Additionally, pointers have the subscript-operator `[n]` that is equivalent to `*(pointer + n)`.

Requirement for using pointer arithmetic is that the data has a defined size (complete types). The “empty-set” type `void` can be used as a pointer type, meaning just the address without any reference to the data-type of the stored data. Since `void` does not have a size, pointers to `void` cannot be used in pointer arithmetics.

2.9.4 Storage duration

All objects in a program have one of the following storage durations:

- **automatic** storage duration. The storage for the object is allocated at the beginning of the enclosing code block and deallocated at the end. All local objects have this storage duration, except those declared `static`, `extern` or `thread_local`.
- **dynamic** storage duration. The storage for the object is allocated and deallocated per request by using dynamic memory allocation functions (`new` and `delete`).
- **static** storage duration. The storage for the object is allocated when the program begins and deallocated when the program ends. Only one instance of the object exists. All objects declared at namespace scope (including global namespace) have this storage duration, plus those declared with `static` or `extern`.
- **thread** storage duration. The storage for the object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object. Only objects declared `thread_local` have this storage duration.

The memory for *dynamic arrays* is allocated “on the heap”, meaning dynamic arrays have *dynamic storage duration*. Dynamic arrays are not destroyed at the end of a function or block scope. They are destroyed only by a call to `delete` or `delete[]`.

2.9.5 Smart pointers

We have (static) arrays and dynamic arrays. The difference is that static arrays have a constant size, but automatic storage duration, whereas dynamic arrays have a dynamic size but also dynamic storage duration. Both can be represented by pointers. What if we want to have a dynamic size, but an automatic storage duration?

This is possible using the special pointer type `std::unique_ptr` or `std::shared_ptr`, sometimes referred to as “smart pointers”. Both are library types defined in the header `<memory>`.

Example:

```
#include <memory>
//...
std::shared_ptr<int> ip{ new int };
std::unique_ptr<int[]> vec{ new int[10] };
// or
std::shared_ptr<int> ip2 = std::make_shared<int>();         // calls `new`
↪   internally
std::unique_ptr<int[]> vec2 = std::make_unique<int[]>(10);

std::cout << *ip << vec[5];
```

So, the definition and usage is very close to classical pointers. But you don’t need to call `delete` to free the memory.

The difference between both types is the criterion when to deallocate the memory:

- `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several `shared_ptr` objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:
  - the last remaining `shared_ptr` owning the object is destroyed;
  - the last remaining `shared_ptr` owning the object is assigned another pointer via `operator=` or `reset()`.
- `std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope. The object is disposed of using the associated deleter when either of the following happens:
  - the managing `unique_ptr` object is destroyed
  - the managing `unique_ptr` object is assigned another pointer via `operator=` or `reset()`.

You can have multiple pointers pointing to the same object, i. e., to the same memory location. Since a pointer is a simple type, it can directly be copied to create a new pointer with the same address:

```
int data = 7;

int* p = &data;
int* q = &data; // second pointer pointing to the same object

int* p2 = p;     // copy of the first pointer creates a new pointer
                 // pointing to the same object
```

The same can be done with `shared_ptr`. They share the same resource, i. e., can point to the same object. Whenever you copy a `shared_ptr`, an internal counter is increased that counts how many pointers are pointing to the same object. Only if this counter is zero, the `shared_ptr` is destroyed at the end of a scope:

```
{
  std::shared_ptr<int> p{ new int(7) };
  std::shared_ptr<int> q{ new int(7) }; // independent new pointer to another
  ↪  memory location

  {
    std::shared_ptr<int> p2 = p; // copy of the first pointer creates a new
    ↪  pointer
                                 // pointing to the same object, the internal
                                 ↪  counter
                                 // is increased

  } // at the end of scope, the second pointer is released -> decrease of the
  ↪  internal
    // counter

  std::shared_ptr<int> p3 = p; // another copy of the first pointer,
  ↪  increases the counter

} // at the end of scope, p and p3 are released, decreasing the counter by 2
↪  -> 0. The
  // memory is finally deallocated.
```

> *Don't use explicit* `new`*,* `delete`*, and owning * pointers, except in rare cases encapsulated inside the implementation of a low-level data structure. (Herb Sutter)*
>
> — **Guideline**

2.10 Namespaces

Namespaces allow the programmer to resolve name conflicts. Defining the same name in the same scope results in an error. Namespaces define named (or unnamed) global scopes.

```
namespace NAME { ... }
```

Symbols declared inside the `namespace` block are visible only in this block. The `NAME` of the namespace can be used to explicitly qualify a function-, class, or variable name inside this namespace. A classical example, that we have seen before, is the standard namespace `std::`. The symbol `::` thereby is the *name-resolution operator* and may be chained to address multiple nested namespaces:

```
namespace Q {
  namespace V { // original-namespace-definition for V
    void f(); // declaration of Q::V::f
  }
  void V::f() {} // OK
  void V::g() {} // Error: g() is not yet a member of V
  namespace V { // extension-namespace-definition for V
    void g(); // declaration of Q::V::g
    }
  }
  namespace R { // not a enclosing namespace for Q
     void Q::V::g() {} // Error: cannot define Q::V::g inside R
  }
  void Q::V::g() {} // OK: global namespace encloses Q
```

One can separate declaration and definition of symbols, but only symbols from sub-namespaces can be defined within a namespace. Not just the definition, but also the access is by the name-resolution operator `::`.

Namespaces can contain functions, classes, global variables. The outer most surrounding scope is called the *global namespace* and is addressed with an empty name before the name-resolution operator.

2.10.1 `using` Directive and Declaration

The statement `using` allows to import names from another namespace. So, those names can then be used without any namespace qualification, without the name-resolution operator.

Either, you can import all names from a namespace, by the using-directive `using namespace NAMESPACE`:

```
namespace scprog {
  void foo() { /*...*/ }
}
int main() {
  using namespace scprog;

  foo(); // Calls foo() from the namespace scprog without scprog::
}
```

This is allowed in namespace scope and block scope only. Every name from the import namespace is visible as if it were declared in the nearest enclosing namespace.

This does not import the functions or classes into the current namespace, but just makes the names visible. This is important for name-resolution, e. g., in function calls.

Note the following:

- If there is already a local name in the scope declared, the name from the using namespace is hidden by the local name
- A name from the imported namespace hides a same name in an enclosing namespace of the scope
- Importing the same name from multiple namespaces into a scope results in a compiler error
- Importing a name that is the same in the global namespace, results in a compiler error

```
#include <iostream>
int foo() { return 1; }       // (1)

namespace scprog1 {
  int foo() { return 2; }     // (2)
}
namespace scprog2 {
  int foo() { return 3; }     // (3)
  int bar() {
    using namespace scprog1;
    return foo();            // OK: calls (3)
  }
}
int main() {
  using namespace scprog2;
  using namespace scprog1;

  std::cout << bar();
  std::cout << foo(); // error: call of overloaded 'foo()' is ambiguous,
                      // candidates: (1), (2) or (3)
}
```

Namenspace alias

If you have a complicated or multiple enclosed namespaces, you can introduce a new name (an alias) for this namespace in the local scope.

```
namespace Q {
  namespace V {  // sub-namespace of Q
    void f() {}; // declaration of Q::V::f
  }
}
void g() {
  using R = Q::V; // alias for the namespace Q::V
  R::f();
}
```

Import of some names from a namespace

Instead of importing all namespace from a namespace (with `using namespace N`) one could just declare a local name for a specific name in the namespace: This allows to not only make a name visible in the current scope, but to actually import that name as if declared in the scope.

```
#include <cmath> // std::sqrt
int main() {
  using std::sqrt;

  sqrt(3.0); // call of std::sqrt without namespace qualification
}
```

It is important to understand the difference to the `using namespace` directive. The `using` declaration is an actual declaration. So, you can declare the exact same name twice, but if a name was already declared in the scope and you introduce that same name again (with a different meaning), the compiler throws an error.

Importing the same name of a function with the same signature from multiple namespace may result in an ambiguous function call.

```
  namespace B {
    void f(int);
    void f(double);
  }
  namespace C {
    void f(int);
    void f(double);
    void f(char);
  }
  void h() {
    using B::f; // introduces B::f(int), B::f(double)
    using C::f; // introduces C::f(int), C::f(double), and C::f(char)
    f('h');      // calls C::f(char)
    f(1);        // error: B::f(int) or C::f(int)?
    void f(int); // error: f(int) conflicts with C::f(int) and B::f(int)
  }
```

2.10.2 Argument dependent lookup

In function call name-lookup, all visible names are inspected. With a using declaration and using directive this list of visible name is extended. But there is a second way of increasing the list of possible candidates for the overload resolution, by *argument dependent lookup*, or *Koenigs lookup*.

There, the names from another namespace are included in the list of visible functions, of a function parameter is declared in that namespace. So, the compiler looks also in the namespace of the function arguments.

```
namespace scprog {
  struct A { double x = 1.0; };

  void set(A& a) { a.x = 2.0; }

  A operator+(A, A const&);
}

struct B { double x = 1.0; };

int main()
{
  scprog::A a, b;
  scprog::set(a); // OK: explicit namespace qulification
  set(b);         // OK: ADL

  scprog::operator+(a,b); // OK: explicit namespace qulification
  a + b;                  // OK: ADL, like operator+(.,.) function call

  B x;
  set(x); // ERROR
}
```

Example 12. *For example, to compile* `std::cout << std::endl;`*, the compiler performs:*

1. *unqualified name lookup for the name* `std`*, which finds the declaration of namespace* `std` *in the header* `<iostream>`
2. *qualified name lookup for the name* `cout`*, which finds a variable declaration in the names-*

pace `std`

3. *qualified name lookup for the name* `endl`*, which finds a function template declaration in the namespace* `std`
4. *argument-dependent lookup for the name* `operator<<`*, which finds multiple function template declarations in the namespace* `std`

3 Classes

Object types or user-defined types are called *classes* in C++ and are introduced with the keywords `class` or `struct` (or `union`). A class defines a new data type and is a collection of data (attributes, member variables), functions (member functions and class functions), and *associated* types. The individual parts of a class are called *members*. All members can be qualified with an access restriction (`public`, `protected`, or `private`).

Example 13. *The following class generalizes the point/vector struct we have seen in the first lecture. It provides functions for the initialization and for the access and some data:*

```
class MyPoint
{
  using value_type = double;      // Associated type

public:
  MyPoint() {};                   // Default-Constructor

  MyPoint(double x, double y)     // Constructor
    : x_(x), y_(y)                // Initializer-List
  { }

  void set(double x, double y);   // Member function
  double norm() const;            // (constant) member function

  static int dim() { return dim_; }  // Class-function, or static member
  ↪  function

private:
  double x_ = 0.0, y_ = 0.0;      // Member variables (with default
  ↪  initialization)
  static const int dim_ = 2;      // Static member variable
};
```

3.1 Objects

Objects are instances of classes, i. e., concrete realizations of a class at runtime. Via an object one gets access to the individual (public) members (variables, functions) of a class. Each new object allocates new memory for all the class members and fill the initial values by using a constructor. So, after construction of a new instance of a class, the object is in a valid state.

The following statements declare a new object `p` or `q` as an instance of the class `MyPoint`:

```
MyPoint p{1.0, 2.0}; // or
MyPoint q(1.0, 2.0);
```

This declaration is also a definition, since memory is allocated. The initialization (construction) of that object is done by the constructor of the class. In this example, the constructor gets two doubles, and sets the values of the member variables `_x` and `_y`.

The definition as above creates an object with automatic storage duration (i. e., the object is created on the "stack"). An analogy to fundamental types, the object can also be allocated on the "heap":

```
MyPoint* p = new MyPoint{1.0, 2.0};
// ...
delete p;
```

The operator `new` returns a pointer to the allocated memory. At first the memory is allocated for all member variables, then the constructor initializes the variables. Similar to all other dynamic memory allocation, the object must be deleted manually.

Objects created with the `new`-operator have to be destroyed with the `delete`-operator to release the memory. At destruction, whether at the end of a scope for objects with automatic storage duration, or when calling `delete`, the *destructor* of the class is called (see below). It is responsible for any cleanup to do.

Remark 37. *Similar to fundamental types and arrays you should not work with raw pointers. Instead, use* smart-pointers *like `std::unique_ptr` or `std::shared_ptr`!*

Remark 38. *When constructing an object using a constructor without arguments, one can either use empty curly braces, i. e.,*

`MyPoint p{};`

or omit any braces, i. e.,

`MyPoint p;`

It is, unfortunately, not allowed to use round empty braces `MyPoint p()`. This would declare a function `p` returning a `MyPoint`, instead.

3.2 Access to class members

To access the data of a constructed class object, one can either use the *dot*-access operator `a.b`, or the *pointer*-access operator `a->b` that is equivalent to `(*a).b`. The former can be used on expressions of *class type* and the latter or *pointer to class type*. Static members or properties of the class can be accessed using the name-resolution operator `A::B`.

Examples:

```
MyPoint  p1{1.0, 2.0};
MyPoint* p2 = new MyPoint(1.0, 2.0);
std::unique_ptr<MyPoint> p3 = std::make_unique<MyPoint>(1.0, 2.0);

double n = p1.norm();   // access on class type
p2->set(2.0, 1.0);      // access on pointer to class type
double n2 = p3->norm(); // access on pointer to class type

int dimension = MyPoint::dim(); // access of static members

delete p2;              // raw pointers must be deleted
```

Remark 39. *The access functions `.`, `->`, and `::` are operators in C++ and thus have a precedence and an associativity.*

3.3 Class Data

Attributes of a class are declared in a class definition and include *methods*, *variables*, *types*, and even sub-classes or *types*.

Each instance of a class gets its own copy of all member variables, except for static variables. Those exist exactly once for each class type and are bound to the type and not to the instance.

3.3.1 Member variables

Variables, declared in a class, can be used to provide data that is distinct in each instance of a class. If a variable is qualified with `const` or `&`, it must be initialized directly in each constructor, since const and reference require direct initialization.

```
class MyPoint
{
public:
  MyPoint(...) : ..., dim(2) {}
//...
  double x;
  double y;
  bool const valid = true;
  int const dim;
};
```

If member variables are supposed to be initialized with a default value in all constructors if not specified otherwise, the initial value can be prescribed directly on declaration, like `bool const value = true;`

3.3.2 Methods

Methods or member functions of a class are functions defined in the scope of the class that have access to all member variables of the instance of that class. Member functions can be defined inline or outside of the class:

```
// in header file:
class MyPoint
{
//...
  void set(double x, double y) // inline definition
  {
    x_ = x;
    y_ = y;
  }

  double norm() const;  // just declaration
};

// in .cpp-file
double MyPoint::norm() const
{
  return std::sqrt(x_*x_ + y_*y_);
}
```

If you intend to include the header file with the class definition multiple times, you need to define all functions in a source file, i. e., a separate translation unit. This prevents from multiple definitions of the same function. Functions defined directly in the class automatically are classified as `inline` functions and thus can exist multiple times.

Within a class method, one gets direct access to all class members (functions, variables, types...). Additionally, there is a special pointer, `this`, pointing to the current instance of that class in memory. This can be used to access the members as well (see pointer-access operator):

```
double MyPoint::norm() const
{
  return std::sqrt(x_*x_ + y_*y_);
}
// or
double MyPoint::norm() const
{
  return std::sqrt(this->x_*this->x_ + this->y_*this->y_);
}
```

The keyword `const` in the member function declaration means, that 1. no member of the class is changed and 2. only other `const` functions are called in that function. If you have a `const` class object or a `const&` to a class object, only `const` member functions can be called:

```
MyPoint const& q = p;
q.set(3.0, 4.0);                // ERROR
std::cout << q.norm() << "\n"; // OK, norm is const-function
```

3.3.3 Static Members

Static variables (class variables) or static function (class functions) are shared by all instances of that class and are introduced with the keyword `static`. Static variables must be defined outside of the class. Static members cannot be initialized in a constructor, since they are independent of any instance.

```
class MyPoint
{
public:
  static int nPoints;
};
...
int MyPoint::nPoints = 0;
```

[C++17] Since C++17, static variables can be marked `inline` that allows to initialize them inside the class, like regular member variables, e. g., `inline static int nPoints = 42;`

Static variables can have the qualifier `const` or `constexpr`. In this way one can define constant properties of a class. If the static constant is additionally an integral type it can always be defined in the class directly:

```
class MyPoint
{
public:
  static const int dim_ = 2;
};
```

Static members can be accessed by the class name and the name resolution operator `::`.

Within any other class member function or static function in the class this can be omitted.

Remark 40. *The definition of a static class member variable must be put into a source file (not in a header file). Otherwise it would be defined multiple times.*

Also functions can have the `static` keyword, indicating that they are bound to the class type and not to any instance of the class. Within static function, only static member variables of the class can be accessed:

```
// in header file
class MyPoint
{
public:
  static int dim() { return dim_; }
  static int dim_;
};
...
// in source file
int MyPoint::dim_ = 2;
...
int main() {
  std::cout << MyPoint::dim();
}
```

3.3.4 Class member types / associated types

Within a class, local typedefs or type aliases can be introduced. This allows to have a central place in the class definition where to fix types used somewhere in the class. And it allows to give that information to the user of the class.

We call those types *associated types* since they are associated with the class type (and thus shared by all instances of that class). Types are introduced using the keyword `typedef` or `using`:

```
class MyPoint
{
public:
  typedef int     size_type; // old syntax
  using value_type = double; // C++11 syntax
};
```

(where I would prefer the new syntax).

In containers of the standard library (and of many other libraries) some generic type names are used:

- `size_type` An integer type that can be used for the size of a container (and for indices)
- `value_type` The type of the elements stored in a container
- `reference`, `pointer`, `iterator`, `const_iterator` Types used to defined the return type of access methods and iteration methods.

```
class MyPoint
{
public:
  using size_type  = int;
```

```
    using value_type = double;

    value_type& operator[](size_type i) { return i == 0 ? x : y; }

  private:
    value_type x, y;
  };
```

Types can be accessed like other static properties, using the name resolution operator: `MyPoint::value_type`.

3.4 Constructors

Constructors are special member functions with their name equal to the class name and no return type. Constructors initialize an object of a class type and create the workspace for all other members. Each instantiation of a class (directly, or with `new`), with or without arguments, calls a constructor. If no constructor is defined at all, the compiler tries to generate one automatically with default behavior.

Example 14. *A constructor could open a file, where other member function tend to write into. At the end of the lifetime of the class this file must be closed.*

It is allowed to declare multiple constructors, while following the rules of regular function overloading, i. e., the constructors must have different signatures.

The construction of a class happens in three stages: 1. memory for all member variables (and all base-classes) is allocated, 2. all members (and base-classes) are initialized by the constructor initializer-lists, 3. the constructor body is executed to finalize any initialization.

3.4.1 Initializer Lists

Constructors are responsible for initializing all member variables. The initialization may invoke the corresponding constructors of those variables. Since all members should be initialized with their corresponding constructors, a special syntax is reserved for this initialization: constructor *initializer lists*.

Directly after the constructor argument list (after the closing round bracket), a colon `:` can introduce the initializer-list. It is a comma-separated list of initializer expressions (constructor calls) for each member variable, base-class, or a call to another constructor (*constructor chaining*, or *delegation*).

After the initializer list, the regular constructor function body must be follow.

Example:

```
class MyPoint
{
public:
  MyPoint(double x, double y) : x_{x}, y_(y) { }  // variable initializers
  MyPoint() : MyPoint(0.0, 0.0) { }               // delegating constructor
  double x_,y_;
};

class MyPoint2
{
```

```
public:
  MyPoint2(double x, double y) : point_{x,y} { }

  MyPoint point_;
}
```

In the first class `MyPoint`, two member variables `x_` and `y_` are initializes with the values of the constructor arguments. Thereby, one can use the initialization brackets similar to regular variables, i. e. curly or round brackets. The second constructor delegates the initialization to the first constructor, by calling the class-constructor with some fixed arguments. The second class `MyPoint2` has a variable of user-defined type that is also initialized by a constructor call passing two arguments.

Note: The order of the initialization should follow the order of the declaration of the member variables in the class (and is independent of the order of the constructor arguments).

Note: If some member variables have default initializers, like `int dim = 2;` those initializations are appended to all constructor initializer lists if no other value for that variable is prescribed.

3.4.2 Some special constructors

Some constructors are special, either because they could be generated automatically by the compiler, or because of the way those constructors are invoked. We will discuss here the default constructor, copy constructor, move constructor, and the destructor (that is not directly a constructor, but kind of the opposite).

Default constructor

A constructor that is defined with an empty parameter list is called the *default constructor*. It is used to bring the class into a default valid state by (typically) initializing all member variables with its default values.

To use the default constructor when instantiating a class there are two variants:

```
// without any argumentlist
MyPoint p1;
MyPoint* q1 = new MyPoint;
// with "universal" initialization
MyPoint p2{};
MyPoint* q2 = new MyPoint{};
```

If there is no other constructor defined, the compiler tries to generate a default constructor with default behavior, i. e., that calls the default constructor on each member variable (and base-class). If it is not generate, if it is not possible, this is no error. For integral and floating-point types the default initializer value is `0` if not specified otherwise by putting the value in the variable definition, i. e., `int dim = 2;`.

If there is any other constructor defined, one can force the compiler to generate the default constructor explicitly, by adding the keyword `= default` to the declaration of the constructor:

```
class MyPoint
{
public:
  MyPoint() = default; // initializes x,y with default values
  MyPoint(double x_, double y_) { ... } // another user-defined constructor
  double x,y;
};
```

Copy constructor

A *copy constructor* is used to create a new instance as a copy of another object. This constructors expects exactly one argument, a const lvalue-reference to the class-type itself:

```
class MyPoint
{
public:
  MyPoint(MyPoint const& that) // copy constructor
    : x(that.x)
    , y(that.y)
  { }
  double x,y;
};
```

In this constructor, it is recommended to also use the constructor initializer lists.

Copy constructors are used/invoked whenever:

- An object is passed by value to a function:

  ```
  double norm(MyPoint p);
  ```

- An object is returned by value from a function:

  ```
  MyPoint generate_point();
  ```

  (Maybe return value optimization or copy-elision kicks in)

- An object of class-type is passed as single argument to another constructor:

  ```
    MyPoint p(1.0, 2.0);
    MyPoint p2(p);
    MyPoint p3 = p;
  ```

- In initializers lists:

  ```
    class A {
      A(MyPoint const& p_) : p(p_) {}
      MyPoint p;
    };
  ```

The default behavior of a copy constructor is, that each member is copied (using its copy constructor). This default behavior is automatically generated by the compiler if the compiler can be sure that this is the expected behavior. This is not the case if

- any member (or base-class) is non-copyable
- a user-defined move-constructor is provided
- a user-defined move assignment operator is provided

If the default behavior at least would be well defined (i. e., all member can be copied), one can force the compiler to generate the default copy constructor by using the keyword `= default`, as above:

```
  class MyPoint
  {
  public:
    MyPoint(MyPoint const& that) = default;
```

```
    double x,y;
  };
```

Also, one could explicitly mark the copy constructor as *deleted* to enforce that a class cannot be copied. This can be accomplished by using the keyword `= delete`, similar to the default.

> *While the arguments to the copy constructor might be given as const or non-const lvalue reference, it is not allowed to pass the argument by value, e. g.,* `MyPoint(MyPoint p)`*. Why not?*
>
> — **Attention**

The cousin of the copy-constructor is the *copy assignment operator*. It is used to copy the content of an object to another already existing object. The argument to the assignment operator thereby is the same as for the copy constructor, but it has to return a reference to the class type:

```
  class MyPoint
  {
  public:
    MyPoint(MyPoint const& that) = default; // copy constructor

    MyPoint& operator=(MyPoint const& that)
    {
      x = that.x;
      y = that.y;
      return *this;
    }
    double x,y;
  };
```

Note: You cannot use the initializer-lists anywhere outside of constructors, thus especially not in assignment operators. Therefore, the copy has to happen in the function body.

> *If you define a copy-constructor you should also define a copy assignment operator.*
>
> — **Principle**

Move constructor

A *move constructor* is used to create a new instance of an object, by "stealing" the data from another object, leaving it in an undetermined but valid state. This requires that the moved-from object is not used anymore. The concept of such a state is a temporary, or an rvalue. So, move constructors expect as single argument an rvalue-reference to the class-type: `MyPoint(MyPoint&& that)`. [`C++11`]

Move constructors are introduced in `C++11` (together with rvalue reference) to allow the fast construction of copies of objects, that are destroyed anyway. Thus, one just use the data of the temporary object, without copying it. This can be motivated by a class storing a pointer to a large dynamic array that was allocated in some constructor and would be destroyed in a destructor. Instead of copying all the elements of that array, a move-constructor could just copy the pointer to the array and set it to `nullptr` inside the moved-from object to forbid the deallocation:

```
class MyPoint
{
```

```
public:
  MyPoint()                   // default constructor
    : big_data(new double[1000000])
  {}

  ~MyPoint()                  // destructor (see below)
  {
    delete [] big_data;       // deallocate the data (if not nullptr)
  }

  MyPoint(MyPoint&& that)     // move constructor
    : big_data(that.big_data) // copy just the pointer
  {
    that.big_data = nullptr;  // unset the point in `that`
  }

  double* big_data;
};
```

In contrast to the copy constructor, the move constructor expects a mutable rvalue-reference, so that fields can be made invalid.

Move constructors are used/invoked when:

- initialization: `T a = std::move(b);` or `T a{std::move(b)};`, where `b` is of type `T`;
- function argument passing: `f(std::move(a));`, where `a` is of type `T` and `f` is `void f(T t);`
- function return: `return a;` inside a function such as `T f()`, where `a` is of type `T` which has a move constructor. (This operation might be removed by the compiler due to return value optimization or copy elision)

The default behavior of a move-constructor is to move construct all its members from the members of the passed argument. This default move constructor can be generated by the compiler automatically. This does not always work, especially if you have data managed dynamically. Thus, a move constructor is implicitly declared only of

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there are no user-declared destructors;

and is implicitly deleted if some data members (or base-classes) cannnot be moved.

In case the default behavior would make sense but another constructor or destructor of the list above is defined, one can force the compiler to generate the move-constructor automatically, using the keyword `= default`, as above.

Also, one could explicitly mark the move constructor as *deleted* to enforce that a class cannot be moved. This can be accomplished by using the keyword `= delete`, similar to the default.

Similar to the copy constructor, there is a cousin of the move constructor, the move-assignment operator: This should be defined whenever a move constructor is provided:

```
MyPoint& operator=(MyPoint&& that)
```

3.4.3 Destructor

A destructor is kind of the opposite to a constructer. It is responsible for releasing all resources allocated in any other constructor of that class, e. g., to free all memory or to close opened files. A destructor is called whenever the life-time of a class end, either at the end of a scope or explicitly via the `delete` operator.

Destructors are indicated using a ˜ in front of the class name. They have an empty parameter list and no return type:

```
class MyPoint
{
public:
  MyPoint(int num = 99999) : big_data(new double[num]) { }
  ~MyPoint() { delete[] big_data; }   // destructor, frees the memory
private:
  double* big_data = nullptr;
};
```

At the end of a user-defined destructor, the destructors of all member variables are called automatically.

The default destructor simply destroys all members using their destructors. This default behavior is automatically generated if no user-defined destructor is provided. If any member cannot be destructed, the implicitly generated destructor is deleted.

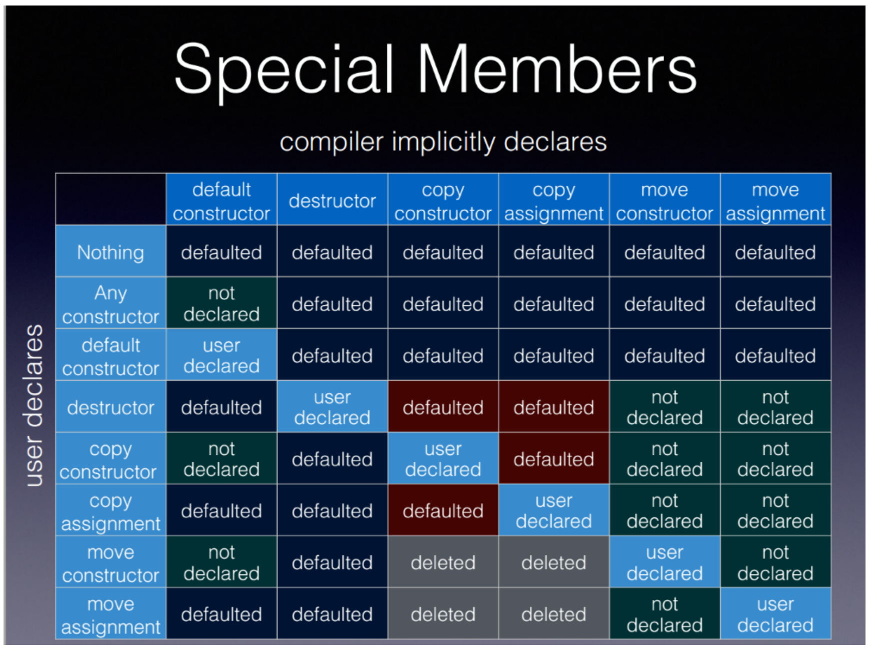
3.5 Rule of five / Rule of zero

Since the compiler sometimes generates constructors (and destructor) automatically and in some combinations not, there is a rule that is known as *rule of five* stating that if you write any of copy constructor, copy assignment operator, move constructor, move assignment operator, or destructor, you have to implement them all.

Especially for copy or move assignment operators, you have to be careful not to assign an object to yourself. Or you have to handle this case manually. The destructor needs to be aware of the move operations and you need to be careful when an error happens somewhere in the process of construction or destruction.

This is often very complicated and thus a common guideline is, to write your class in such a way that the compiler always generates everything for you and you don't need to write any of the special constructors (including the default constructor). This pattern is called *rule of zero* and means, especially, that you should not do dynamic memory management using `new` and `delete` inside of your classes (or just in one class that is responsible just for this), but use library types that are good replacements for your manual management, like `std::shared_ptr`, `std::unique_ptr`, `std::vector`, `std::array`, or `std::string`.

> *Use Data-members that already have an own copy/move constructors and assignment operators and destructor for their data-management. So, you are not responsible for the release of allocated memory and don't need to implement an own constructor and destructor yourself.*
>
> — **Rule of Zero**



Special Members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Figure 3.1: Howard Hinnant: *Everything You Ever Wanted To Know About Move Semantics (and then some)* (ACCU 2014)

3.6 Access restriction

In C++ it is possible to restrict in classes who can access the members of that class. There are two access groups:

- `public`: Everyone can access the members.
- `private`: Only other class members are allowed to access this attribute.

Additionally, there is the access restriction `protected` that we will see later.

Example 15. *For the illustration of access restrictions, consider the following example:*

```
class Point {
public:
  double x,y;
  void foo() {
    a = 2.0;  // OK: in a member function of the class we can access
    ↪  private data
    bar();    // OK: in a member function of the class we can call private
    ↪  methods
  }
private:
  double a;
  void bar();
};
// ...
Point p{1.0, 2.0}; // Initialization
```

```
p.x;   // OK: x is a public member variable
p.a;   // ERROR: a ist a private member and cannot be accessed from outside
↪  the class

p.foo(); // OK: foo is a public member function
p.bar(); // ERROR: bar is a private member function
```

The keywords `public`, `private` (and `protected`) followed by a colon `:` introduces a block in which all following members have this access restriction.

Remark 41. *In classes introduced with* `class`*, the default access restriction is* `private`*, whereas in* `struct`*, the default access restriction is* `public`*.*

Remark 42. *Also constructors can be restricted. This allows to instantiate a class (or copy-/move a class) in a specific way only. If there are no public constructors available, the class cannot be instantiated directly and one has to provide a "factory"-function instead, i. e., a static public member function that constructs and returns the instantiated class.*

3.7 Inheritance

(The chapter inheritance is touched here only slightly, see other lectures about object oriented design for more details)

Classes characterize properties and methods of a specific space of objects. An example is a class representing a vector space. Instances of that class are vectors. But, there are other spaces, with more or less properties, like a normed vector space that additionally has a norm function, or a scalar-product space that additionally has a scalar-product. So, we can find a hierarchy of classes with a general class and more and more specialized classes. The more specialized ones can inherit properties / members of the more general class, like a normed vector space inherits all methods from a vector space. This principle is called inheritance in C++.

In the head of a class definition, one can list other classes we want to inherit properties and members from. Thereby, we can specify how those inherited members are visible (can be accessed) (Either they are just properties we want to use internally: `private`, or all properties are made visible to the outside: `public`).

```
class VectorSpace {                     // Base class
public:
  VectorSpace();
  VectorSpace operator+(VectorSpace const& rhs) const;
  VectorSpace operator-(VectorSpace const& rhs) const;
  // ...

  int dimension_;                       // is public to the outside and
  ↪  derived classes

protected:
  double component(int i);              // is accessible from derived
  ↪  classes, but not public

private:
  std::vector<double> components_;      // is private to this class
};

class NormedVectorSpace : public VectorSpace
{
public:
  double norm() const
  {
    double n = 0.0;
    for (int i = 0; i < dimension_; ++i)  // we can access public members of
     ↪  base class
      n += component(i)*component(i);     // we can access protected
       ↪  functions of base class
    return std::sqrt(n);
  }
};
// ...
VectorSpace vec1;
vec1.dimension_     // OK: public member
vec1.component()    // ERROR: protected members not accessible
vec1.components_    // ERROR: private members not accessible
vec1.norm()         // ERROR: VectorSpace has no method norm()

NormedVectorSpace vec2;
vec2.dimension_     // OK: public derived member
vec2.component()    // ERROR: protected member of VectorSpace
vec2.norm()         // OK: public function
```

The derived class can call protected and public members of the base class, and can even call its constructors.

Remark 43. *Again, the difference between `class` and `struct` is that for class the inheritance is `private` by default and for struct it is `public` by default.*

4 Generic programming

If you have an algorithm or function that can be expressed independent of the representation details of the arguments and without complicated case-by-case variations, it should be written in a generic form. Thereby, instead of implementing a concrete function (or class), just a template for the function (or class) is provided that is later filled with concrete realizations of types. This is called *generic programming*.

> *Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.*
> — Musser, David R.; Stepanov, Alexander A., Generic Programming (1989)

4.1 Templates

Generic programming in C++ allows to parametrize functions and classes not just with values (function parameters, or constructor arguments), but with types. A *template parameter* thereby is a special kind of parameter passed to the function, class, or even variable. Inside of the realization of the function or class, this type parameter, represented by its name, can be used as any other regular type, e. g., to declare variables, to specify function parameter lists or return types, or to parametrize other function or class templates.

An introductory example illustrates the usage of function and class templates:

```cpp
#include <cmath>

// class template
template <typename T> // T is type for coordinates, e.g. T=double
struct MyPoint
{
  using value_type = T; // introduction of a type alias
  T x, y;               // type T used here to declare a variable
                        // alternatively: value_type x,y;
};

// function template
template <typename P>   // P is type for the vectors to calculate the
↪  distance of
auto distance(P const& a, P const& b) // type P in the argumentlist, both
↪  arguments
{                                     // should have the same type.
  using T = typename P::value_type;   // access to a class type
  T dx = a.x - b.x;
  T dy = a.y - b.y;                   // Or: auto dy = a.y - b.y
  return std::sqrt(dx * dx + dy * dy);
}
```

The call to the (generic) function `distance`, i. e., the *instantiation* of that function with a concrete type, is initiated by passing concrete arguments to that function, or by specifying the type explicitly

```
int main()
{
  // Type for MyPoint explicitly specified
  MyPoint<double> a{1.0, 2.0}, b{7.0,-1.5};

  // Or implicit deduction of the type (C++17)
  MyPoint c{2.0, 3.0};   // double coordinates
  MyPoint d{3.0f, 4.0f}; // float coordinates

  // Type for the arguments of distance( , ) automatically deduced
  std::cout << "Distance of a and b (double) = "
            << distance(a,b) << std::endl;

  // Similarly, by explicit specification:
  std::cout << "Distance of a and b (double) = "
            << distance<MyPoint<double>>(a,b) << std::endl;

  // Call the function with different type
  MyPoint<float> a2{1.0f, 2.0f}, b2{7.0f,-1.5f};
  std::cout << "Distance of a and b (float) = "
            << distance(a2,b2) << std::endl;

  // invalid argument types
  distance(1.0, 2.0); // ERROR: `double` has no `value_type` and access to
                      // the members x and y not possible

  // incompatible argument types MyPoint<double> and MyPoint<float>
  distance(c, d);  // ERROR
}
```

4.2 Generic functions

Generic functions extend the concept of function overloading. The generic functions are called *function templates* (and sometimes also *templated functions*).

The basic idea is that instead of a function with concrete types, we parametrize the function with a type parameter that can be used instead of the concrete type.

The syntax to declare a function template:

```
template <typename NAME1, typename NAME2, (...)> FUNCTION_DECLARATION;
template <class NAME, (...)> FUNCTION_DECLARATION;
```

It simply starts with the keyword `template` and in angular brackets `<...>` a comma separated list of template parameters.

The argument `NAMEx` is called *template parameter* and must be a valid name in C++. The argument is introduced with either the additional keyword `typename` or `class` for type-parameters (there is no difference of these two keywords).

Remark 44. *Also, `NAME` can be a* nontype-*parameter, a* template-*parameter, or a* parameterpack. *We will focus here on type parameters and later introduce nontype parameters.*

Remark 45. *The number of template parameters is restricted only by compiler-specific limits.*

The call of a function template is a two-phase procedure:

1. Template instantiation → concrete function
2. Invocation of that concrete function

Template instantiation is the process of filling the placeholder template parameters with concrete types (or values, see below). This can be achieved in two ways. 1. Explicit specification of the types:

```
FUNCTION_NAME<TYPES...>(PARAMETERS);
```

where `TYPES` must be replaced with concrete types in the order of the template parameter declaration (without the keywords `typename` or `class`).

Or the template instantiation can be obtained by implicit deduction of the template parameter types from the function arguments (*argument type deduction* (ATD)):

```
FUNCTION_NAME(PARAMETERS);
```

Thereby, the function parameter must be dependent on all the template parameter types.

Example 16. *We have the function template* `abs_difference` *defined as follow:*

```
template <typename T>
T abs_difference(T value1, T value2)
{
  return std::abs(value1 - value2);
}
```

Explicit instantiation of this function template:

```
abs_difference<double>(1.0, 2.0);
abs_difference<float>(1.0f, 2.0f);
```

and implicit instantiation by argument type deduction:

```
abs_difference(1.0, 2.0); // -> T = double
abs_difference(3, 4);     // -> T = int
```

The type deduction thereby is very similar to the `auto` type deduction from the auto-placeholder syntax introduced before. Actually, in C++ first the template parameter deduction was invented and later the auto-type deduction was added with `C++11`.

Remark 46. *Argument type deduction only works if all the types can be deduced uniquely from the function parameters. In the example above, we have declared a function template with two parameters and have specified both with the same template parameter* `T`*. Thus, we have to pass two arguments of the same type to the function template in order to allow the template parameters to be deduced from the function parameters:*

```
double a = 1.0;
float b = 2.0f;
abs_difference(a,b); // ERROR: cannot deduce template parameters
```

The parameter `T` *could be either deduced to* `double` *or to* `float`*.*

But, if the template is instantiated with an explicit type, no automatic deduction needs to be done and thus we have a regular function that can be called with different argument that are automatically converted:

```
    abs_difference<double>(a, b); // OK => generates function
    ↪   'double(double,double)' and
                                  // calls this function with parameters
                                  ↪   double and float
                                  // -> implicit conversion of float to
                                  ↪   double
```

Since function templates can be "called" like regular functions, name and overload resolution must be considered. This will be discussed in detail later. Here just an example:

```
  // non-template function
  double abs_difference(double a, double b); // (a)

  // function template
  template <class T>
  T abs_difference(T a, T b);                // (b)

  double a=1.0, b=2.0;
  abs_difference(a,b);          // (1) calls (a)
  abs_difference<double>(a,b); // (2) calls (b)
  abs_difference<>(a,b);        // (3) calls (b)
```

If there is an exact match of a non-templated function, this is always preferred (1). If the explicit instantiation with empty template parameter list, it asks for argument type deduction, but enforces a call to the instantiated template, instead of the non-template (3).

4.2.1 Templates as placeholders

Similar to the `auto` placeholder, template parameter can be equipped with qualifiers, like `const`, reference (or pointer). But additionally, template parameters can be used just like regular types, as parameters to other templates, like class templates:

```
  // accept all std::vector arguments only
  template <class T>
  double norm1(std::vector<T> const& vec); // (a)

  // accept any type, and pass it by reference
  template <class T>
  double norm2(T const& vec);          // (b)

  // accept any type, and pass it by value
  template <class T>
  double norm3(T vec);                 // (c)
```

The rules how to deduce a type `T` from the arguments is just like the deduction rules of `auto`, plus a pattern matching in case the parameter is used in another template.

```
std::vector<double> vec = {...};

norm1(vec); // T = double
norm2(vec); // T = std::vector<double>
norm3(vec); // T = std::vector<double>
```

4.3 Generic Classes

A generic class, or *class template* (or templated class) is a user-defined type, parameterized with other types (and values). This allows to specify the internal data-type during the *instantiation* of the outer class type. Important examples of class templates are containers, where the type of the elements that are stored in the container are passed as template parameters to the class template (like `std::vector<double>`).

The syntax is similar to that of generic functions:

```
template <typename NAME1, typename NAME2, (...)> CLASS_DECLARATION;
template <class NAME1, (...)> CLASS_DECLARATION;
```

We call this a *template declaration* (or *template definition* in case of class definition) with `NAMEx` the *template parameters*. `CLASS_DECLARATION` is thereby either `struct CLASS_NAME` or `class CLASS_NAME` (similar to regular classes). Again, there is no difference between `typename` and `class` of the template parameter declaration, for type parameters.

Remark 47. *In analogy to function parameters, template parameters can have default "values" (or default types), specified starting from the right-most template parameter:*

```
template <class T1, class T2, class T3 = int> class CLASS_NAME { ... };
```

The *template instantiation* of a class template, i. e., the replacement of a template parameter with a concrete type (or value), follows the syntax

```
CLASS_NAME<TYPES...> variable(CONSTRUCTOR_ARGUMENTS); // or...
CLASS_NAME<TYPES...> variable{CONSTRUCTOR_ARGUMENTS}; // Uniform
↪   initialization, or
                                                       // initializer lists
```

where `TYPES...` is a list of concrete types (without the keyword `typename` or `class`).

Remark 48. *With C++17 the class template parameters can be automatically deduced from the class constructor arguments. This is called* class template argument deduction. *Similar to argument type deduction for function templates the template parameters are deduced from the arguments passed to a constructor of the class template. This works if all class template parameters are related to a constructor parameter:* [C++17]

```
std::pair p(2, 4.5);        // deduces to std::pair<int, double> p(2, 4.5);
std::tuple t(4, 3, 2.5);    // same as auto t = std::make_tuple(4, 3, 2.5);
std::vector v{1,2,3,4,5};   // deduces to std::vector<int> v{1,2,3,4,5};
```

but

```
std::vector v2(7);        // ERROR: type cannot be deduced. The constructor
↪   argument is
                          // just the size of the vector, no value.
```

See also cppreference:class_template_argument_deduction.

4.4 Non-type parameters

Apart from type parameters, template parameters can represent integral values. Instead of specifying the template parameter with `typename` or `class`, it is declared with any integral value type, e. g., `int`.

The following example defines a class template of a container with fixed number of elements that is given as non-type template parameter:

```
template <class T, int N>
struct fixed_size_array
{
  T& operator[](int i) { return data[i]; }
  const T& operator[](int i) const { return data[i]; }

private:
  T data[N];  // size is fixed by template parameter
};

int main()
{
  fixed_size_array<double, 3> point3D;
  point3D[2] = 7.0;
}
```

Remark 49. *Not all types are allowed as non-type template parameters. It must be possible to uniquely evaluate those types at compile-type. Therefore the C++ standard has restricted non-type template parameters to integral types, enums, characters, and pointer types (until C++17) and floating-point types and some literal class types (since C++20).*

Similar to type parameters, default values are possible:

```
template <class T, int N = 3>
struct fixed_size_array { ... };
```

Since non-type parameters must be specified at compile-time to instantiate that template, it must be a `constexpr`. Runtime values are not allowed:

```
constexpr int n = 7;
fixed_size_array<double, n> point7D; // OK

int m = 5;
fixed_size_array<double, m> point5D; // ERROR: m is runtime value
```

4.5 Comparison of templates to concrete types/functions

A class generated from a class template is like any other regular / concrete class. It is just a user-defined data-type with size and alignment. One can instantiate that class and create type aliases to that class. Especially,

- there is no additional runtime overhead compared to a hand-written (concrete) class,
- there is no space overhead in the instantiated template compared to a hand-written class.

Each class template is translated into a regular class at template instantiation. Those instantiations get internally a unique name tepending on the instantiated template parameters, e. g., `mypoint<double>` ⇒ `[...]mypointId[...]` (in g++)

After instantiation of a template, all its internal types, member functions, member variables and static constants are concrete, have a concrete type or (in case of static constants) have a concrete value, that can be evaluated and asked for.

For function templates it is similar. After function template instantiation, the template becomes a regular function, that participates in overload resolution like any other function. It can be passed as a function parameter to other functions, or can be stored in a function reference variable, or a `std::function` object. And, similar to class templates,

- there is no additional runtime overhead compared to a hand-written (concrete) function,
- there is no space overhead in the instantiated template compared to a hand-written function.

5 Functors

A Functor is an abstraction of a classical function, but implemented as class type. It can be used (called) like a regular function, but has additional advantage to use all the flexibility of a class, like local member variables, or other member functions that can be called.

The basic idea is, that a class that implements the `operator()` can be evaluated like a function. The call to that special member function has the same syntax as a regular function invocation.

Example:

```
struct Square {
  // Implementation of the bracket operator
  double operator()(double x) const {
    return x*x;
  }
};

int main() {
  // Instantiation of the class
  Square square{};

  // Use the instance of the class, like a regular function
  double y = square(3.0); // = 9
}
```

Implement functors as classes with an overloaded `operator()` as `public` and (often) `const` member-function.

— **Guideline**

5.1 Functors as function parameters

Since functors are regular user-defined types, they can be passed to any other function as function parameter. This allows to write algorithms without specifying the concrete operation explicitly in code, but expecting this operation to be passed as argument. Examples for this type of algorithms are sorting algorithms, where your want to specify the comparison operator, e. g., `std::less`, as argument so that you can easily switch between increasing and decreasing order of the elements.

Another example is the function `for_each`, that applies to each element in a container a function (e. g., to change the value of the element):

```
void for_each(std::vector<double>& vec, Square const& f) {
  for (auto& v_i : vec)
    v_i = f(v_i);
}
```

```
int main() {
  std::vector<double> v{1.0, 2.0, 3.0, 4.0, 5.0};

  // Instantiation of the class
  Square square{};

  // Pass the functor to the function
  for_each(v, square);

  // -> {1.0, 4.0, 9.0, 16.0, 25.0}
}
```

We could even instantiate the functor directly in the call of the `for_each` function, i. e.,

```
for_each(v, Square{});
```

The problem with this approach is, that we might want to pass different functors to the same function / algorithm. This is a perfect example of a function template. Instead of explicitly specifying the type of the functor that we can pass, we specify a template parameter for the functor argument:

```
template <class F>
void for_each(std::vector<double>& vec, F const& f) {
  for (auto& v_i : vec)
    v_i = f(v_i);
}
```

This allows to call the function `for_each` the same way as before (thanks to ATD), but we can also pass another functor without reimplementing the algorithm.

Remark 50. *A more generic implementation of the `for_each` function uses a second template parameter for the container type, i. e.,*

```
template <class Range, class F>
void for_each(Range& range, F const& f) {
  for (auto& element : range)
    element = f(element);
}
```

This allows to use that function with `std::vector<...>` of any type supported by the functor, or `std::list`, `std::array`, or any other type that can be iterated over using range-based for-loops.

5.2 Parametrized Functors

We can not only parametrize the functions that use the functor, but also the functor itself. The allows to specify the argument or return types of the `operator()` implemented inside the functor, or any other useful type that we need to implement the functor operation.

Example 17. *The square functor from above could be applied to any number type that supports multiplication. Thus, we could make the implementation more generic by parametrizing the functor:*

```
template <typename domain_type, typename range_type = domain_type>
struct Square
{
    range_type operator()(domain_type x) const {
      return x*x;
    }
  };
```

In the instantiation of the class, we now have to instantiate the functor class template as well:

```
for_each(v, Square<double>{});
```

Often, the functor class itself needs no parametrization, but the member function `operator()` depends on template parameters. So, one could directly parametrize just the member function, instead of the whole class:

Example:

```
  struct Square
  {
    template <typename domain_type>
    auto operator()(domain_type x) const {
      return x*x;
    }
  };
```

For simplicity, I have used the return type `auto` to let the compiler automatically deduce the return type from the expression in the function. The `auto` return type gets very useful when combined with input template parameters.

The call to `for_each` is now as simple as before:

```
for_each(v, Square{});
```

and we do not need to know the type of the argument in advance.

5.3 Lambda expressions

In `C++11` the definition of functors was simplified a lot. Instead of writing a class or class template first, then instantiating that class and passing it to an algorithm, one can define an "anonymous" functor directly in the line of usage. The compiler then generates automatically a corresponding class with a unique names, instantiates that class and passes it to the function. [`C++11`]

Those anonymous functors are called *lambda expressions* in C++ and can be declared as follows:

```
  [captures...] (parameters...) -> ReturnType { function body };
  /* or shorted in case of empty parameter list */
  [captures...] { function body };
```

Without the *capture* clause, it looks like a regular function definition.

- The *captures* is a list of zero or more variables or references that can be used inside the function.
- *parameters* is a list of function parameters including its types and qualifiers (like in regular function declarations)
- *ReturnType* is the (optional) (trailing) return type of the function, that can depend on the types of the function parameters It can be omitted, resulting in the automatic return type deduction, like in functions returning `auto`.
- The function body may contain any sequence of C++ statements and (optionally) a

return statement.

Example 18. *The following example illustrates the usage of lambda expressions, compared to classical functors:*

```
// functor with internal member variable to store data.
struct Scale {
  double factor_;

  Scale(double factor = 1.0)
    : factor_(factor)
  {}

  double operator()(double value) const
  {
    return value * factor_;
  }
};

int main() {
  std::vector<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};

  double factor = 3.0;

  // Instantiate a regular functor with constructor argument
  for_each(v, Scale{factor});

  // Use a lambda expression
  for_each(v, [factor](double value) { return factor * value; });
}
```

The variable `factor` *is captured, and can thus be used inside the lambda expression.*

The capture list is very similar to a list of constructor argument of the functor. All variables in that list introduce a local member variable, that can be used inside the lambda expressions function body. Any other variable cannot be used there, since it generates an unnamed functor class with an `operator()` member function that is called on evaluation.

Using the capture list, a connection to the surrounding scope can be established. The following values are possible in the capture list:

```
[]    // no capture
[a]   // copies the variable a into a local member variable of the functor
[&a]  // stores a reference to the variable a
[=]   // copies all variables of the surrounding scope that are used inside
↪   the function body
[&]   // references all variables of the surrounding scope ...
```

5.3.1 Generic lambdas

Like functions and classes, also lambda expressions can be made generic. In the example `Square` above, we have parametrized the inner member function `operator()` with the type of the arguments. A similar templated bracket operator for lambda expressions can be obtained, by using the placeholder `auto` inside the function parameter list:

```
for_each(v, [factor](auto value) { return factor * value; });
```

This introduces a member-function template with an unnamed template parameter, i. e., it is similar to the functor definition

```
template <class T>
struct Scale {
  T factor_;

  Scale(T factor = T(1))
    : factor_(factor)
  {}

  template <typename S>
  auto operator()(S value) const
  {
    return value * factor_;
  }
};
```

but, we do not have a name for the type of the parameter like `S` in the explicit functor definition.

Remark 51. *The type of the function parameters in generic lambdas can be obtained by using the* `decltype` *operator.*

Similar to classical templates and the `auto` placeholder in variable declarations, the function parameters in generic lambdas can be qualified with `const`, reference and pointer. But they cannot be used as placeholders in other templated types, e. g., `std::vector<auto>` is not allowed as function parameter in generic lambdas.

Remark 52. *With C++20 one can introduce named template parameters in generic lambdas. Therefore, one adds a declaration of template parameters after the capture block in the lambda definition:* [C++20]

```
[...] <class P1, class P2, (...)> (P1 param1, P2 param2...) -> ReturnType {
↪  function body };
```

5.4 Abbreviated function template

Since C++20, a similar syntax as used for generic lambda can be used for regular function. When placeholder types `auto` appear in the parameter list of a function declaration or of a function template declaration, the declaration declares a function template, and one invented template parameter for each placeholder is appended to the template parameter list. [C++20]

```
void f1(auto arg); // same as template<class T> void f1(T arg)
void f2(auto const& arg); // same as template<class T> void f2(T const&
↪  arg)
```

This is especially useful if the actual type of the argument is irrelevant but the algorithm allows to generically pass arguments of different type.

6 Iterators

A central layer of abstraction introduced in the C++ standard library are *iterators*. Frankly, iterators are objects allowing the programmer to traverse the elements of an arbitrary container, without knowing the specific implementation, the data storage or the access pattern to individual elements of that data-structure. While vectors and arrays may have a direct element access, using the `operator[]`, lists and trees do not have such a method, or it might be very expensive to retrieve a specific element in the sequence rather than traversing linearly through the container.

An abstraction that you find in C and that is often that case of an implementation of a data-structure are pointers, pointing to the individual elements or to the next and previous element in a sequence. But it is not required that containers are implemented with a chain of pointers and constructing the pointer to an element could be an additional operation. Also, pointers do not give me any information about how a sequence might be traversed and where to find the next element.

The abstraction that we want to discuss here are *iterators*. They provide similar semantics as regular pointers, but incrementing (or decrementing) an iterator means pointing to the next (previous) element in the sequence rather than the next position in memory. To allow iterators to be used similar to pointers, they provide at least the following operation:

- Dereferencing with `operator*`, to access the element the iterator is pointing to
- Going to the next element in the sequence, be pre or post increment `operator++`
- Test for equality and inequality by `operator==` and `operator!=`.

Additionally, iterators are light-weight objects, meaning: they can be copied and returned/-passed by value without much overhead. They have a size similar to a regular pointer.

6.1 Iterators of standard library containers

In the standard library, all containers (e. g., `std::vector`, `std::list`, `std::set`) provide at least two functions, `begin()` and `end()` returning an iterator object. Thereby, `begin()` points to the first element in the sequence and `end()` is a so called *sentinel* – an abstraction of a past-the-end iterator.

Using these two iterators, a traversal over the elements of a container can be realized by

```
#include <iostream>
#include <vector> // or <list> or <set> or ...
int main() {
  std::vector<double> v = {...};
  for (auto it = v.begin(); it != v.end(); ++it) {
    // comparison of iterators, increment of the iterator

    std::cout << *it << std::endl;
    // dereferencing of the iterator to get the value
  }
}
```

In standard library containers, the type of iterators is given as a typedef/alias type in the container class, accessible by `::iterator` or `::const_iterator`. A `const_iterator` can be seen as an abstraction of a pointer to `const`, whereas an `iterator` as a pointer to non-const elements. Thus, one could write out the explicit type in the loop by

```
for (std::vector<double>::const_iterator it = v.begin(); it != v.end();
 ↪  ++it)
  std::cout << *it << std::endl;
```

[C++17] The end-iterator is not necessarily of the same type as the begin-iterator.

6.1.1 Range-based for-Loops

Since iterating over a container is essentially always the same, take begin, compare to end, increment; a loop structure that does essentially those steps is added in C++11, called *range-for* loop.

The syntax follows the general pattern

```
for (range_declaration : range_expression) loop_statement
```

Example 19. *Consider the following code:*

```
std::vector<double> v = ...;
for (auto const& v_i : v) { std::cout << v_i << std::endl; }
```

Here, the declaration of of the variable `auto const& v_i` *is the* range_declaration*, the vector* `v` *is a simple* range_expression *(could be anything that generates a range) and* `{ std::cout << v_i << std::endl; }` *is the* loop_statement*.*

The three parts can be defined as

range_declaration a declaration of a named variable, whose type is the type of the element of the sequence represented by range_expression, or a reference to that type. Often uses the auto specifier for automatic type deduction

range_expression any expression that represents a suitable sequence (either an array or an object for which begin and end member functions or free functions are defined, see below) or a braced-init-list.

loop_statement any statement, typically a compound statement, which is the body of the loop

The generic implementation, i. e., what the compiler make out of the range-for loop statement above, is the following code:

```
{
  auto && __range = range_expression;
  auto __begin = begin_expr;
  auto __end = end_expr;
  for ( ; __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

where `begin_expr` (and `end_expr`) represent the evaluation of `begin(c)` or `c.begin()` (and `end(c)` or `c.end()`).

6.2 Iterators in generic functions

In the same way all containers of the standard library can be iterated through as seen above. This allows to formulate a general programming methodology, implemented in the standard library:

Decouple the Implementation of Data Structures and Algorithms.
— **Fundamental Methodology**

Each container provides iterators for traversing its content. Each algorithm in the standard library, one the other hand, is implemented in terms of iterators. In a classical programming style where an algorithm is implemented for a specific data-structure, to implement m algorithms for n containers one needs

$$m \cdot n \text{ implementations}$$

whereas for algorithms based on iterators, one needs just

$$m + n \text{ implementations}$$

In order to allow iterators to be passed to functions, one needs function templates.

Example 20. *Lets have a look at an algorithm of the standard library:* accumulation *of the values in a container.*

```
template <typename InputIterator, typename T, typename BinaryFunction>
T accumulate(InputIterator it, InputIterator end, T init, BinaryFunction
↪  op)
{
  T sum(init);
  for (; it != end; ++it)
    sum = op(sum, *it);
  return sum;
}
```

One could think of summing up the values. Then, the `BinaryFunction` *is the function returning the sum of two passed arguments and the* `init` *value is typically, the neutral value for the plus operator (0), i. e., where to start the sum from.*

Additionally we get two iterators, describing the range of value we want to sum up. `it` *is thereby the first element, and* `end` *the position after the last element in the range, so that is it never equal to an iterator until the iteration is finished.*

A corresponding `main()` *function could look like*

```
#include <set>        // std::set
#include <functional> // std::plus

int main() {
  std::set<double> v = ...;
  double s = accumulate( v.begin(), v.end(), 0.0, std::plus<double>{} );
}
```

So, we can sum up the values of a set, even if they are not stored contiguously in memory.

6.2.1 Iterator categories

Not all iterators support iteration forward and backward. Some iterators not even support multiple iterations through the same sequence (after the first traversal, elements are invalidated), but other containers might even support jumping to an arbitrary position in the sequence.

To distinguish the different functionality of iterators, the standard library has introduced *iterator categories*:

Input iterator Supports only forward iteration at least once through the sequence, and allows to read from the iterators, i. e., const dereferencing.

- `operator++()`
- `operator++(int)`
- `operator*() const`
- `operator->() const`

Forward Iterator Same as *Input Iterator* but supports multiple iterations through the same sequence.

Bidirectional Iterator Same as *Forward Iterator* but additionally supports backward iteration:

- `operator--()`
- `operator--(int)`

Random Access Iterator Same as *Bidirectional Iterator* but additionally supports arbitrary forward and backward jumps in the sequence:

- `operator+=(int)`
- `operator-=(int)`
- `operator+(int)`
- `operator-(int)`
- `operator[](int)` increment (or decrement) by arbitrary distance and direct dereferencing
- `operator-(iterator)` distance between two iterators

and also comparison operators `<, >, <=, >=`.

Contiguous Iterator Same as *Random Access Iterator* but additionally guarantees contiguous storage of the elements in the underlying container.

Output Iterator Supports only forward iteration at least once through the sequence, and allows to write to the elements the iterator is pointing to, i. e., mutable dereferencing.

- `operator++()`
- `operator++(int)`
- `operator*()`
- `operator->()`

The categories introduced above formally describe what an iterator can do and where it can be applied. Some algorithms might require random access to the elements in the container, like some sorting algorithms, other algorithms are fine with just forward (or even input iterators). So, we need a way to distinguish different iterator types by its categories.

6.2.2 Iterator traits

Traits are class templates that just provide typedefs/alias types or static values, depending on the template parameters. The special template *Iterator Traits*, parametrized with an iterator type, thereby provides access to some properties the iterators. A container can provide its own iterator traits (specialization) to give information about its iterators.

A default implementation of the standard library iterator traits could look like the following:

```
template <typename Iter>
struct iterator_traits {
  using iterator_category = typename Iter::iterator_category;
  using value_type        = typename Iter::value_type;
  using difference_type   = typename Iter::difference_type;
  using pointer           = typename Iter::pointer;
  using reference         = typename Iter::reference;
};
```

So, it defines a type representing the *Iterator Category* and some more types, useful for generic algorithms.

Thereby, the type `pointer` describes the return type of the `operator->()` of an iterator, the type `reference` the return type of `operator*()`, `difference_type` is a type that can be used to describe the distance between iterator (typically a signed integer type) and that is used in the random access increments (not just `int` as written above).

The category is represented by empty classes, forming a hierarchy that represents the connection between the categories above:

```
struct input_iterator_tag { };
struct output_iterator_tag { };

struct forward_iterator_tag
  : public input_iterator_tag { };

struct bidirectional_iterator_tag
  : public forward_iterator_tag { };

struct random_access_iterator_tag
  : public bidirectional_iterator_tag { };

struct contiguous_iterator_tag
  : public random_access_iterator_tag { };
```

Example 21. *The iterator traits can be used to write more generic algorithms. Consider the* accumulate *function from above: The return type might be different from the init type, e. g., it could be the type of the elements of the sequence:*

```
template <typename InputIterator, typename T, typename BinaryFunction>
auto accumulate(InputIterator it, InputIterator end,
                T init,
                BinaryFunction op)
{
  // extract a type from the iterator traits
  using result_type = typename iterator_traits<InputIterator>::value_type;

  result_type sum(init);
```

```
    for (; it != end; ++it)
      sum = op(sum, *it);
    return sum;
  }
```

Remark 53. *(Dependent types) Is an associated type (alias type in a class) extracted from a class template instantiated with a template parameter from the surrounding scope, we call it a dependent type – the type depends on template parameters. The additional keyword* `typename` *is necessary to tell the compiler it is actually a type we are asking for and not a static value or function.*

Example 22. *Some algorithms can be improved by knowledge about the iterators. An example is the function* `std::distance` *returning the distance between two iterator, i. e., how many increments are necessary to reach from begin the end iterator.*

```
template <class InputIterator>
typename std::iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

The return type is the `distance_type` *associated to the iterator. Two algorithms could be implemented:*

1. *Iterate from* `first` *to* `last` *and count the steps.*
2. *Directly take the difference between* `first` *and* `last` *it the iterator is a* Random Access Iterator

Since the difference is only possible if the type supports this operation, we cannot simply use an `if - else` *branching. we have to implement two function, differing only in the iterator category:*

```
template <class InputIterator>
typename std::iterator_traits<InputIterator>::difference_type
distance_impl(InputIterator first, InputIterator last,
↪  std::input_iterator_tag)
{
  typename std::iterator_traits<InputIterator>::difference_type d = 0;
  for (; first != last; ++first, ++d) /* no operation */ ;
  return d;
}

template <class InputIterator>
typename std::iterator_traits<InputIterator>::difference_type
distance_impl(InputIterator first, InputIterator last,
↪  std::random_access_iterator_tag)
{
  return last - first;
}

template <class InputIterator>
typename std::iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last)
{
  return distance_impl(first, last,
    typename std::iterator_traits<InputIterator>::iterator_category{});
}
```

Note, that also forward and bidirectional iterators can be used with these two functions, since they are both also input iterators, but not random-access iterators. This can be seen at the class hierarchy of the iterator categories.

When specializing generic algorithms, based on properties of the types, tag dispatching *can be used. Thereby, an additional function parameter is introduced that gets passed a simple, typically empty, class, called* tag *that represents the specialization.*
If tags are put into a class hierarchy, automatic type conversion to base types can be used together with function overloading to allow to specialize for several similar properties.

— **Technique**

6.3 Algorithms

The C++ standard library implements data-structures and algorithms that can be combined in a flexible way, by using iterators. All container implementations at least have a method `begin()` and `end()` returning iterators to the first and one-past-the-end element. Algorithms on the other hand are written in terms of iterators for the input and output. Thus, data-structures and algorithms are separated.

Some algorithms of the standard library are introduced here. More can be found in cppreference.com in the `<algorithm>`[1] and `<numeric>`[2] headers.

[1] http://en.cppreference.com/w/cpp/algorithm
[2] http://en.cppreference.com/w/cpp/numeric

6.3.1 Copy

Copies the elements of a range[3] [first, last) into another range described by just one output-iterator. Two variants are described here: std::copy and std::copy_if. The latter only copies those elements fulfilling some condition.

```
template <class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first)
{
  while (first != last) {
    *d_first++ = *first++;
  }
  return d_first;
}
```

The parameters first and last are iterators of type InputIt and should fulfill the concepts of an input-iterator. The third parameter d_first is an out-iterator pointing to the first element in the output range. It is of type OutputIt. The types InputIt and OutputIt are different, allowing to copy from one container in a different one.

The return value of the copy function is again of type OutputIt and thus an iterator into the output range. It points to one-past the element copied last in the algorithm. Thus, the iterator pair [d_first, return_value) defines a range that has identical elements as the input range [first, last).

Since the output range is directly filled using the output iterators, it is expected that this range can store enough elements, i. e., the output iterator can be increment at least as often as the input iterator. It is, on the other hand, not required, that the returned iterator is identical to the end-iterator of the output range. It is fine to copy less elements that the range could store.

Example 23. *Copy elements from a list into a vector:*

```
#include <algorithm>
#include <cassert>
#include <list>
#include <vector>

int main()
{
  std::list<int> l = {0, 1, 2, 3, 4, 5};
  std::vector<int> v(l.size());
  auto it = std::copy(l.begin(), l.end(), v.begin());

  assert( it == v.end() ); // here, the return iterator points to the end
  ↪  of the vector
}
```

[3]A *range* denotes a data-structure that can be traversed by a pair of iterators

The second variant of copy, `std::copy_if`, expects an additional function parameter representing a *unary predicate*, i. e., function or functor taking one parameter and returning a `bool`:

```
template <class InputIt, class OutputIt, class UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last, OutputIt d_first,
 ↪  UnaryPredicate pred)
{
  while (first != last) {
    if (pred(*first))
        *d_first++ = *first;
    first++;
  }
  return d_first;
}
```

The output-iterator is incremented only if the predicate returns `true`.

Example 24. *In the second example, we copy only elements with even values. We use a lambda expression to define the predicate.*

```
#include <algorithm>
#include <cassert>
#include <list>
#include <vector>

int main()
{
  std::list<int> l = {0, 1, 2, 3, 4, 5};
  std::vector<int> v(l.size());
  auto it = std::copy_if(l.begin(), l.end(), v.begin(),
    [](int i) { return i % 2 == 0; }); // copy only even elements
}
```

Back-inserter

The problem with the copy algorithm is that we need to resize the output container before we can copy into it. In case of `copy_if` we even do not know how many elements are copied. It would be better if the algorithm itself could “resize” the container to the required size. But we do not get a container as input, but an iterator. This iterator has no functionality to resize the underlying container. It often does not even has a reference to the container it points into.

A workaround for this problem is to use a different output-iterator. Not an iterator directly pointing to elements of the output-range, but a special iterator wrapper that performs on assignment to the dereferenced iterator an `insert()` or `push_back()` operation. The corresponding wrapper type is called `std::back_insert_iterator` and is created using the function `std::back_inserter`.

```
#include <algorithm>
#include <cassert>
#include <iterator> // std::back_inserter
#include <list>
#include <vector>
```

```
int main()
{
  std::list<int> l = {0, 1, 2, 3, 4, 5};
  std::vector<int> v; // an empty vector

  // add new elements to the back of the vector on copy
  auto it = std::copy(l.begin(), l.end(), std::back_inserter(v));

  assert( v.size() == l.size() );
}
```

The class-type `std::back_insert_iterator` implements the concept of an `OutputIterator`. Thus, it can be incremented and dereferenced. But, both methods have a different effect as expected, essentially they do nothing, but, in case of the dereference operator, just return `*this`, i. e., a reference to itself. Additionally, this iterator implements an assignment `operator=`, calling `push_back()` on the wrapped container with the value given to the assignment operator. (see also the exercise)

Similarly, there is `std::front_insert_iterator` adding elements to the front of the container by `push_front()`.

Ostream-Iterator

Also some very special output-iterators can be used in combination with `std::copy()`. An example is the `std::ostream_iterator`. Instead of inserting the copied elements into a container, this iterator pushes the element into an output-stream, like `std::cout`, using the `operator<<`.

This allows to write all the elements of any container to the output in one line:

```
#include <algorithm>
#include <iostream>
#include <list>

int main()
{
  std::list<int> l = {0, 1, 2, 3, 4, 5};

  // print all elements of the list to the screen
  auto it = std::copy(l.begin(), l.end(),
  ↪  std::ostream_iterator<int>(std::cout, " "));
  std::cout << '\n';
}
```

The `std::ostream_iterator` is a class-template parametrized with the type of the element it has to print to the output-stream. It is constructed from an output-stream object, like `std::cout`, `std::cerr` or `std::ofstream`, and optionally as second argument a delimiter separating the printed elements.

The `std::ostream_iterator` is a model of a *single-pass* `OutputIterator`, thus can be "traversed" only once. In a second traversal the state of the output-range is already changed and we cannot go back.

6.3.2 Fill, generate, and transform

The following three algorithm applied to a container can change the value of the elements.

The first one, `std::fill`, sets all elements to the same value:

```
template <class ForwardIt, class T>
void fill(ForwardIt first, ForwardIt last, const T& value)
{
  for (; first != last; ++first) {
    *first = value;
  }
}
```

The algorithm `std::generate` fills a range by values generated by a nullary functor, called generator:

```
template <class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last, Generator g)
{
  while (first != last) {
    *first++ = g();
  }
}
```

The third algorithm, `std::transform`, applies a unary functor to the elements of a range and assigns the result to an output-iterator:

```
template <class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first,
↪   UnaryOperation unary_op)
{
  while (first1 != last1) {
    *d_first++ = unary_op(*first1++);
  }
  return d_first;
}
```

(There is a second version of `std::transform` taking two input ranges and one output range by iterators. With this, you could implement element-wise binary operations, e. g., sum of two vectors.)

Example 25. *The following example illustrates the interaction of multiple algorithms and data-structures:*

```
#include <algorithm>
#include <cmath>
#include <list>
#include <vector>

int main()
{
  std::list<int> l(10);
  std::vector<int> v(10);

  // generate input data
  std::fill(l.begin(), l.end(), 2);
```

```
    std::generate(v.begin(), v.end(), [](){ return std::rand() % 10; });

    // an empty output sequence
    std::list<int> erg;

    // sum the elements of the input ranges element-wise
    std::transform(l.begin(), l.end(), v.begin(), std::back_inserter(erg),
                   [](int a, int b) { return a+b; });
  }
```

6.3.3 Iota, accumulate, and inner_product

The following three algorithms are taken from the standard library `<numeric>`.

The first algorithm, `std::iota`, fills a sequence with increasing values, starting from an initial value:

```
  template <class ForwardIt, class T>
  void iota(ForwardIt first, ForwardIt last, T value)
  {
    while(first != last) {
      *first++ = value;
      ++value;
    }
  }
```

The first two parameter define the output-range, while the third parameter gives the start value for the increasing value sequence.

The second algorithm, `std::accumulate`, we have seen before. It is a reduction algorithm, returning one value as combination of all values in the range. Thus, it reduces a range to a single value. The default behavior of the accumulate algorithm is to sum up the values of the range, but it is possible to pass an additional binary functor argument, that defines the actual reduction operation to perform:

```
  template <class InputIt, class T, class BinaryOperation>
  T accumulate(InputIt first, InputIt last, T init, BinaryOperation op)
  {
    for (; first != last; ++first) {
      init = op(init, *first);
    }
    return init;
  }
```

The third algorithm in this chapter takes two ranges, given by its iterators, ane reduces the combined elements of the ranges to a single output values. The default behavior of this algorithms is to perform a classical Euclidean inner-product, i. e., to sum up the products of the elements of the two ranges.

Instead of this default operation, one can pass two functors to the algorithm. The first one is the reduction functor (plus by default) and the second one is the element-combination functor (multiplies by default):

```
template <class InputIt1, class InputIt2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T value,
                BinaryOperation1 op1, BinaryOperation2 op2)
{
  while (first1 != last1) {
    value = op1(value, op2(*first1, *first2));
    ++first1;
    ++first2;
  }
  return value;
}
```

Example 26. *The following example first generates sequences of increasing values and then calculates the two-norm and the inf-norm as well as an inner-product:*

```
#include <numeric>
#include <cmath>
#include <list>
#include <vector>

int main()
{
  std::list<int> l(10);
  std::vector<int> v(10);

  // generate input data
  std::iota(l.begin(), l.end(), 0); // 0,1,2,3...
  std::iota(v.begin(), v.end(), 1); // 1,2,3,4...

  auto two_norm = std::sqrt( std::accumulate(l.begin(), l.end(), 0,
    [](int res, int a) { return res + a*a; }) );

  auto inf_norm = std::accumulate(v.begin(), v.end(), 0,
    [](int res, int a) { return std::max(res, a); });

  auto dot = std::inner_product(l.begin(), l.end(), v.begin(), 0,
    [](int a, int b) { return a+b; },
    [](int a, int b) { return a*b; });
}
```

Index

Bibliography

[1] B. Stroustrup. *The Design and Evolution of C++*. Pearson Education, 1994.

[2] T. L. Veldhuizen and M. E. Jernigan. Will c++ be faster than fortran? Technical report, Department of System Design Engineering, University of Waterloo, 2000.