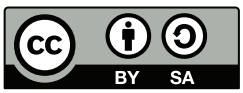
Modern C++ Programming Techniques for Scientific Computing

Innovations from C++11 to C++20

Dr. Ole Klein
Interdisciplinary Center for Scientific Computing (IWR)
Heidelberg University
email: ole.klein@iwr.uni-heidelberg.de

Winter Semester 2020/21

License and Copyright



If you would like to reference these lecture notes in your own scientific works, please cite as "Klein, O., 2021: Modern C++ Programming Techniques for Scientific Computing, Lecture Notes, Heidelberg University" including the link under which this document can be found online:

https://conan.iwr.uni-heidelberg.de/people/oklein/

Contents

1. Introduction **3**
1.1. Preface . . . 3
1.2. Acknowledgments . . . 4
1.3. Why C++? . . . 4

2. Fundamental Concepts of C++ **10**
2.1. Variables and Data Types . . . 10
2.2. Pointers and References . . . 13
2.3. Control Structures . . . 15
2.4. Functions . . . 17
2.5. Templates . . . 22
2.6. Classes . . . 26
2.7. Inheritance . . . 34
2.8. Code Structure . . . 39

3. The Standard Library **42**
3.1. Input / Output . . . 42
3.2. Container Classes . . . 44
3.3. Iterators . . . 48
3.4. Algorithms . . . 49
3.5. Companion Classes . . . 50
3.6. Exceptions . . . 50

4. Advanced Topics **52**
4.1. Template Specialization . . . 52
4.2. Templates and Inheritance . . . 56
4.3. RAII: Resource Acquisition is Initialization . . . 59
4.4. Template Metaprogramming . . . 62
4.5. Dynamic Polymorphism . . . 64
4.6. Static Polymorphism . . . 71
4.7. SFINAE: Substitution Failure is not an Error . . . 73

5. C++11 Features **79**
5.1. Automatic Type Deduction . . . 79
5.2. Compile-Time Evaluation . . . 81
5.3. Move Semantics . . . 82
5.4. Smart Pointers . . . 83
5.5. Lambda Expressions (Closures) . . . 90
5.6. Variadic Templates . . . 92
5.7. Concurrency with Threads . . . 94
5.8. Assorted New Features and Libraries . . . 103

6. C++14 Features **112**
6.1. Improvements to Constant Expressions . . . 112
6.2. Generic Lambda Expressions . . . 113
6.3. Variable Templates . . . 113
6.4. Return Type Deduction . . . 114
6.5. Other Improvements and Additions . . . 115

7. C++17 Features **117**
7.1. Guaranteed Copy Elision . . . 117
7.2. Compile-Time Branches . . . 119
7.3. Structured Bindings . . . 120
7.4. Utility Classes . . . 121
7.5. Fold Expressions . . . 122
7.6. Improvements for Templates . . . 122
7.7. Mathematical Special Functions . . . 123
7.8. Filesystems Library . . . 124

8. C++20 Features **125**
8.1. Modules . . . 125
8.2. Concepts . . . 126
8.3. Ranges . . . 128
8.4. Concurrency with Coroutines . . . 128
8.5. Mathematical Constants . . . 130
8.6. Text Formatting . . . 130

A. Exercises **132**
A.1. Fundamentals and Standard Library . . . 132
A.2. Advanced Topics . . . 140
A.3. Modern C++ Features . . . 149

1. Introduction

1.1. Preface

The following pages present the content of a lecture called "Object-Oriented Programming for Scientific Computing" that I have given regularly at Heidelberg University for over half a decade. While earlier iterations of this lecture where indeed geared towards object-oriented programming, in the last few years the focus has more and more shifted towards a comprehensive introduction to scientific programming techniques in C++, which may be of interest to a larger audience. I have therefore decided to make these teaching materials available online in the form of an electronic document, hoping that they will prove useful.

In the last few years, the C++ language has undergone several shifts in how the language is typically written: while C++98/03 code can quite often look strikingly like C code, with C-style arrays, enums, and explicit allocations and deallocations, code based on the upcoming C++20 standard can be quite similar to some other popular modern languages like, say, Python. Most importantly, the old way of writing C++ remains perfectly valid, meaning that C++ is a rather flexible language that can be adapted to a variety of coding styles and layout preferences.

The strong focus of C++ on runtime performance, portability and flexibility makes it quite suitable for scientific computing and high-performance computing. Unfortunately, the language has a reputation for excessive verbosity and complexity, mainly because template-based code is not only quite complex and longwinded itself, but additionally tends to produce compilation error messages which are exceedingly long and difficult to understand. Modern C++ techniques, however, can often alleviate this problem, and lead to much clearer and more concise code. It therefore certainly makes sense to discuss these new additions in detail. Nevertheless, we first discuss older techniques, mainly for two reasons: first, to provide adequate context for the more modern constructs, and second, because these superseded approaches remain ubiquituous in legacy code bases, and knowing about them is therefore important for anyone working on such projects.

The following sections are based on my LaTeX `beamer` lecture slides, and are set using the `beamerarticle` companion document class. This has the huge advantage that the content will, hopefully, stay in sync with updates to the lecture slides, but has the drawback that the layout may sometimes be less than optimal. I have strived for exactness in all the presented topics, but have sometimes chosen to slightly simplify matters to avoid overly verbose and pedantic formulations. Corrections, additions, and suggestions are more than welcome — please do not hesitate to let me know about potential ways to improve these materials.

— *Ole Klein, Heidelberg, Winter Semester 2020/21*

1.2. Acknowledgments

This document doesn't exist in a vacuum, instead it references and makes use of several online resources, among them:

- The C++ Super-FAQ at isocpp.org/faq
- The C++ Reference at en.cppreference.com
- Herb Sutter's websites: gotw.ca / herbsutter.com
- Matt Godbolt's Compiler Explorer: godbolt.org
- Jean Guegant's blog at jguegant.github.io/blogs/tech/
- Anders Schau Knatten's C++ Quiz at cppquiz.org

I'd like to use this opportunity to thank the authors and contributors of the websites referenced within this document for their efforts in creating and maintaining these online resources.

Several real-world code examples from the *Dune project*, dune-project.org, and *PDELab sub-project*, dune-project.org/modules/dune-pdelab, are discussed within this document. Dune, the *Distributed and Unified Numerics Environment*, is a set of open-source C++ libraries for Scientific Computing, licensed under the GPL. I'd like to acknowledge the efforts of the contributors of the Dune project, and thank them for the opportunity to use their code base to demonstrate real-world applications of some of the concepts that are discussed in these notes.

1.3. Why C++?

A (non-exhaustive) list of programming languages for scientific computing:

Fortran (1957) old (think punchcards) but still very relevant today (e.g., numerical linear algebra, legacy physics/astronomy codes)

C (1972) widespread language for high-performance low-level programming (operating systems, compilers, ...)

C++ (1985) started as "C with classes", newer iterations favor a style that is closer to, e.g., Python

Python (1990) popular language with large ecosystem of numerical software libraries

Julia (2012) relatively new language specifically designed for high-performance scientific computing

There are also several domain specific languages (DSL) that are strong in their respective domains, e.g.:

MATLAB (1984), *R (1993)*

Each language has its advantages and disadvantages, so which should be used for a course like this one?

Ideally, such a programming language for scientific computing ...

- is general-purpose, so no DSL
- produces highly efficient and portable programs
- provides a large ecosystem of numerical / scientific libraries
- has proven its suitability over the years
- has a large following, which provides support and makes it unlikely that the language will vanish
- can serve as a starting point to learn other, related languages

Fortran has a small community nowadays, is used for very specific applications, and its writing style has little overlap with the other aforementioned languages.

C is basically a subset of C++, apart from some technicalities.

Python would be a really good choice, but it is easier to move from C++ to Python than vice versa, and Python tends to hide some aspects of scientific programming that should be taught.

Julia is a relatively new language that is not yet in wide-spread use.

... which leaves us with *C++* as a solid choice.

Versions of C++

The C++ language has evolved significantly over the last few years:

- "Classic" C++ code is officially known as *C++98/03*.
- Current versions of C++, i.e., *C++11* with its relatively minor updates C++14 and C++17, have quite a different feel.
- In general, modern C++ constructs should be preferred wherever possible.
- But: these are often not covered in introductory courses.

We therefore start the lecture with a quick review of C++98/03, which is then used as starting point to discuss modern C++, how it evolved, and how it can be used to produce more readable and more maintainable code.

Evolution of C++

Classic C Style

```
// C-style fixed-length array
int fix[10] = {0,1,2,3,4,5,6,7,8,9};

// "fix" doesn't know its own size
for (int i = 0; i < 10; i++)
  if (fix[i] % 2 == 0)
    std::cout << fix[i] << " ";
std::cout << std::endl;

// C-style "variable-length" array
int* var = new int[n];
for (int i = 0; i < n; i++)
  var[i] = i;

// "var" isn't a real variable-length array:
// adding elems requires copying (or tricks)

// "var" doesn't know its own size
for (int i = 0; i < n; i++)
  if (var[i] % 2 == 0)
    std::cout << var[i] << " ";
std::cout << std::endl;

// oops, forgot to delete array: memory leak!
```

- C-style arrays are just references to *contiguous blocks of memory* (basically pointers to first entry)
- They *don't follow value semantics*: copies refer to same memory blocks
- Their *length is not stored* and has to be specified explicitly, inviting subtle errors

- Runtime fixed-length arrays aren't true variable-length arrays
- May lead to *nasty memory leaks* if they aren't explicitly deallocated

Evolution of C++

C++98/03

```
// C++ variable-length array
// from header <vector>
std::vector<int> var(n);
for (int i = 0; i < n; i++)
  var[i] = i;

// std::vector is a real variable-length array
var.push_back(n+1);

// no need to remember size of "var"
for (int i = 0; i < var.size(); i++)
  if (var[i] % 2 == 0)
    std::cout << var[i] << " ";
std::cout << std::endl;

// very general (also works for maps, sets,
// lists, ...), but reeeally ugly
for (std::vector<int>::const_iterator it
    = var.begin(); it != var.end(); ++it)
  if (*it % 2 == 0)
    std::cout << *it << " ";
std::cout << std::endl;
```

- C++ introduced `std::vector`, a *true variable-length array*, i.e., elements can be added and removed
- Vectors *have value semantics*: copies are deep copies
- A vector always *knows its current size*, no need to keep track
- Same performance as C-style arrays (drop-in replacement)
- Can be used in generic code via iterators (but leads to very verbose code)

Evolution of C++

C++11 / C++20

```
// C++ variable-length array
std::vector<int> var(n);
// C++11: fill using algo from <numeric> header
std::iota(var.begin(),var.end(),0);

// C++11: range-based for loop
// hides ugly iterators
for (const auto& e : var)
  if (e % 2 == 0)
    std::cout << e << " ";
```

```
std::cout << std::endl;

// C++11: lambda expression (ad-hoc function)
auto even = [](int i){return i % 2 == 0;};

// C++20: filters and transforms
for (const auto& e : var
    | std::views::filter(even))
    std::cout << e << " ";
std::cout << std::endl;
```

- C++11 introduced *range-based* `for` *loops*, making iterator-based code much more readable
- C++20 will introduce *filters and transforms* that can operate on such loops, here in the example based on a C++11 lambda expression (ad-hoc function definition)

Evolution of C++

C++11

```
// C-style fixed-length array
int fix[10] = {0,1,2,3,4,5,6,7,8,9};

// C++11: range-based for works with
// C-style arrays, but only for those
// with compile-time fixed length!
for (const auto& e : fix)
  if (e % 2 == 0)
    std::cout << e << " ";
std::cout << std::endl;

// C++11: modern array type from header <array>
std::array<int,10> fix2 = {0,1,2,3,4,5,6,7,8,9};

// no need to remember size of "fix2"
for (int i = 0; i < fix2.size(); i++)
  if (fix2[i] % 2 == 0)
    std::cout << fix2[i] << " ";
std::cout << std::endl;
```

- C++11 range-based `for` loops can be used with legacy arrays, since the compiler knows their size implicitly
- However, this doesn't work when the length is runtime-dependent!
- C++11 also introduced `std::array` as a drop-in replacement for fixed-length C-style arrays, with *known size and value semantics* like `std::vector`

Versions of C++

Which version of C++ should I learn / use?

C++98/03 remains relevant due to (a) vast collections of *legacy codebases* resp. (b) programmers that still use old constructs and are set in their ways.

C++11 is the *current baseline* and should be supported on virtually all platforms and compute clusters by now.

C++14 is a minor update of C++11 and is also a safe choice. Most software projects should accept C++14 code by now.

C++17 is a second minor update of C++11. This is perfectly fine for your own code, but keep in mind that large projects may be restricted to older standards to support some architectures.

C++20 is the *upcoming new version* and will bring major changes. This will become relevant in the near future.

2. Fundamental Concepts of C++

The modern components of C++ are often built upon older constructs of the language, may serve as superior replacements for some of them, or both.

These *fundamental concepts* are:

- variables and types
- pointers and references
- control structures
- functions and templates
- classes and inheritance
- namespaces and structure

They are taught in practically any introductory course that is based on C++. We will quickly review them to make sure that everyone has the prerequisites for the following lectures.

2.1. Variables and Data Types

Variables, Temporaries, Literals

C++, like any other programming language, concerns itself with the *computation and manipulation of data*.

This data represents many different things, from simple numbers and strings, to images and multimedia files, to abstract numerical simulations and their solutions.

Put simply, C++ knows three different categories of data:

Variables are *names for locations* where data is stored, e.g., `int i = 5;` referring to an integer value in memory.

Temporaries represent values that aren't necessarily stored in memory, e.g., *intermediate values* in compound expressions and function return values.

Literals are values that are explicitly mentioned in the source code, e.g., the number `5` above, or the string `"foo"`.

Data Types

C++ is a *strongly-typed language*, which means that each such representation of data, i.e., variable, temporary or literal, must have an associated *data type*.

This data type specifies how the underlying binary sequence encodes the data (*semantics*), and more importantly ensures that it isn't possible to accidentally misinterpret data or use it in the wrong context (*type safety*).

C++ has a number of *built-in data types* and allows the introduction of *user-defined types* based on certain rules. Each type is associated with a range of valid values of that type.

Fundamental Types

C++ provides a set of *fundamental, or built-in, types*, mostly inherited from C.

`void`**:**

A type that has no valid values. Represents "nothing" (e.g., as return type), sometimes "anything" (when using pointers).

`nullptr_t`**:**

A type with one value, `nullptr`, indicating an invalid pointer. Introduced to make pointer handling safer.

`bool`**:**

Two values, `true` and `false`, for standard Boolean algebra (truth values).

`char` **et al.:**

ASCII characters, also similar types for unicode support (`wchar_t`, `char16_t`, `char32_t`).

`int` **et al.:**

Integer numbers, with different ranges (`short`, `int`, `long`, `long long`) and signedness (`signed` and `unsigned`). `signed` is default and may be omitted. Also a standard type for container sizes (`std::size_t`), one for pointer differences (`std::ptrdiff_t`), and a very long list of fixed width types, like `std::int8_t`, etc.

`float` **et al.:**

Floating point numbers, with single precision (`float`: 32 bit), double precision (`double`: 64 bit), or extended precision (`long double`: usually 80 bit). Nowadays, `double` is used as default floating point type if there is not a good reason to use something else.

Each of the integer and floating point types gives certain guarantees about the representable range. These and other properties can be queried using `std::numeric_limits`.

Introducing New Types

These built-in types can be combined with four different mechanisms to produce new type definitions. We already saw the first one, C-style arrays, the others are:

`enum`:

A *user-defined set of constants*:

```
enum Color = {red, blue, green};
```

These are actually integers behind the scenes, and may accidentally be used as such: prefer → *scoped enums*.

`struct`:

The *cartesian product* of some types, i.e., the set of all possible tuples of values:

```
struct PairOfInts {int a; int b;};
```

`union`:

The *union set* of some types, i.e., a type that can hold values from all specified types:

```
union IntOrChar {int c; char d;};
```

Unions don't store the type they currently contain, which is dangerous: consider → *variants* instead.

The resulting data types may then be used in further type definitions, e.g., structs as members of other structs, or data types that are augmented with additional components using → *inheritance*.

Examples from DUNE

Whenever possible, we will have a quick look at real-world code from the *DUNE project*[1], preferably from the *PDELab subproject*[2].

```
// classic enum defining boundary types for a domain
enum Type { Dirichlet=1, Neumann=-1, Outflow=-2, None=-3 };

// modern C++11 enum for DG finite element basis definition
enum class QkDGBasisPolynomial
    {lagrange, legendre, lobatto, l2orthonormal};

// In C++, structs are actually just classes with
// public attributes, we therefore refer to class
// examples in that regard.

// DUNE currently contains no unions (which is a good thing!)
```

2.2. Pointers and References

Pointers

Each type `T`, whether built-in or user-defined, has an associated type `T*` (*pointer to* `T`) defined, with the meaning "address of a value of type `T` in memory".

```
int i = 5;
int* p = &i;

int* p2 = new int;
*p2 = 4;
delete p2;
```

- An ampersand `&` in front of a variable produces its address
- An asterisk `*` in front of a pointer *dereferences* the pointer, providing access to the variable itself
- The keyword `new` can be used to acquire a slot in memory that is unnamed, i.e., not associated with a variable

[1] https://www.dune-project.org
[2] https://www.dune-project.org/modules/dune-pdelab

It is imperative to release such memory with `delete` after it is no longer needed. Doing so too early leads to nasty bugs, forgetting to do so causes memory leaks. A possible solution are → *smart pointers*.

References

In constrast to C, C++ introduces a second indirection that serves a similar purpose: *references* `T&`. While pointers simply refer to some memory, and may be modified to point to other locations, references are always an *alias for an existing entity*.

```
int a = 5;
int& b = a; // b is an alias for a
b = 4;      // this changes a as well
```

Note that the symbol `&` is used in two different contexts, first to take addresses of variables and second to specify reference types. This is a bit unfortunate, but can no longer be changed (both constructs have seen widespread use over the last few decades).

Rvalue References

The third kind of indirection in C++ is the *rvalue reference* `T&&`, introduced in C++11. Ordinary references `T&` are now also known as *lvalue references* to distinguish the two concepts.

The (oversimplified) definitions of lvalue and rvalue are:

lvalue could be on the lefthand side of an assignment, is an actual entity, occupies memory and has an address

rvalue could be on the righthand side of an assignment, temporary and ephemereal, e.g., intermediate values or literals

Such rvalue references are mostly restricted to two use cases: as forwarding references in → *range-based* `for` *loops*, and as a vital component of the implementation of → *move semantics*.

Const-Correctness

The type of variables, pointers and references may be marked as `const`, i.e., *something that can't be modified*. This guards against accidental modification (bugs), and makes programmers' intent clearer. Therefore, `const` should be used wherever possible.

```
int i = 5;         // may change later on
const int j = 4;   // guaranteed to stay 4
const int& k = i; // k can't be modified
i += 2;            // ... but this still changes k indirectly!

const int* p1 = &i;       // pointer to const int
int const* p2 = &i;       // same thing
int* const p3 = &i;       // constant pointer to modifiable int
int const* const p4 = &i; // const pointer to const int
```

Read right-to-left: `const` modifies what's left of it (think `int const i`), but the leftmost const may be put on the lefthand side for readability (`const int i`).

2.3. Control Structures

Selection (Conditionals)

Branches are a fundamental form of program flow control. An `if` statement consists of a condition, some code that is executed if that condition is fulfilled, and optionally other code that is executed if it isn't fulfilled:

```
if (i % 2 == 0)
  std::cout << "i is even!" << std::endl;
else
  std::cout << "i is odd!" << std::endl;
```

There is also a `switch` statement, but `if` is used significantly more often.

```
switch(color) // the Color enum we introduced earlier
{
  case red:   std::cout << "red"   << std::endl; break;
  case blue:  std::cout << "blue"  << std::endl; break;
  case green: std::cout << "green" << std::endl;
}
```

Repetition (Loops)

C++ provides two different kinds of *loops*, `for` and `while` loops. The former is executed a fixed number of times[3], while the latter is repeated until some condition is met.

```
for (int i = 0; i < 10; i++)
  std::cout << i << std::endl;

int j = 9;
while (j > 1)
{
  std::cout << j << std::endl;
  j /= 2; // half, rounded down
}
```

If `j` were one, the loop would be skipped completely. There is also a variant `do{...} while(...)` that runs at least once.

Jump Statements

Sometimes code becomes more readable if certain parts of a loop can be skipped. The `continue` statement can be used to skip the rest of the current loop iteration, while `break` exits the loop prematurely:

```
for (int i = 0; i < 100; i++)
{
  if (i % 2 == 0)
    continue;
  if (i > 10)
    break;
  // prints 1, 3, 5, 7, 9
  std::cout << i << std::endl;
}
```

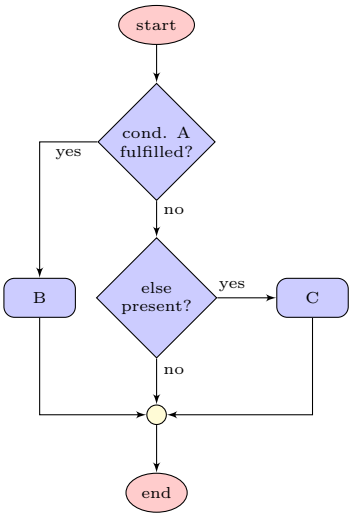
These two statements jump to the end of the current iteration resp. the loop, i.e., they are constrained. C++ also has an *unconstrained* `goto` *jump statement*, but its use is strongly discouraged. There is *one accepted use case*: exiting several nested loops simultaneously (`break` would leave the innermost loop only).

[3] in practice — on a technical level, `for` and `while` are perfectly equivalent

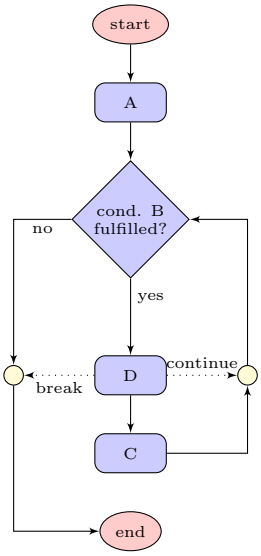
Diagram: Loops and Branches

The two fundamental control structures:

`if (A) B; [[else C;]]`



`for (A;B;C) D;`



The code `while (A) B;` is equivalent to `for (;A;) B;`.

The code `do A while (B);` is equivalent to `A; while (B) A;`.

2.4. Functions

Subprograms (Functions)

A *function* is a subprogram that may be reused in different parts of the program, which reduces verbosity and helps in structuring the code. Functions have a name, zero or more arguments with fixed type, and a return type:

```
// expects one double argument, returns double
double square(double x)
{
  return x * x;
}

// expects an int and a double as args, returns nothing
void printSquares(int a, double b)
{
```

```
  std::cout << square(a) << " and " << square(b) << std::endl;
}
```

The special type `void` indicates a function that doesn't return anything. Such functions typically have *side effects* (I/O, or modifications to their arguments).

Call-by-Reference

C++ is one of the languages that always create copies of arguments when a function is called (*call-by-value*). This means that local changes to these variables don't modify the originals. In some other languages, the local names refer to the actual memory locations that were passed to the function (*call-by-reference*).

To emulate this behavior in C++, one passes a pointer or reference instead[4]:

```
// modifies its argument
void square(int* i)
{
  *i = *i * *i;
}

// prefer references: code is more readable
void square(int& i)
{
  i = i * i;
}
```

Call-by-reference is also often used when a function should return more than one value: one emulates this by modifying one or more reference arguments. C++17 and later standards provide → *guaranteed copy elision* and → *structured bindings* as a better alternative.

For large entities (e.g., vectors, matrices) it often makes sense to pass them by reference even if they should not be modified, since this *avoids costly copy operations* (both in terms of runtime and memory use):

[4] There's still a copy (of the pointer/reference), but it refers to the original location.

```
// directly access original vector,
// but guarantee that it isn't modified
int sum(const std::vector<int>& vec)
{
  int out = 0;
  for (int i = 0; i < vec.size(); i++)
    out += vec[i];

  return out;
}
```

Default Arguments

C++ supports *default arguments for functions*. Arguments with defaults may then be omitted when calling the function[5]. This simplifies the function interface when these arguments have certain values most of the time:

```
void print(const std::vector<int>& vec, std::string sep = ", ",
           std::string pre = "(", std::string post = ")")
{
  std::cout << pre;
  for (int i = 0; i < vec.size() - 1; i++)
    std::cout << vec[i] << sep;
  if (!vec.empty())
    std::cout << vec.back();
  std::cout << post;
}
```

A call `print(v)` will then use a default layout, but other variants can be used if desired.

Function Overloading

C++ offers *function overloading*, i.e., using the same name for several different functions, as long as each function call is uniquely determined by the arguments (including handling of default arguments).

[5] That's why they always come last: to keep the call unambigious.

```
int maximum(int a, int b)
{
  if (a > b)
    return a;
  return b;
}

// a default argument for c would have to be
// smaller than any possible integer
int maximum(int a, int b, int c)
{
  return maximum(maximum(a,b),c);
}
```

Operator Overloading

C++ provides a large assortment of *operators*, i.e., tokens that are placed inline to specify some operation, like assignment (`a = b`), arithmetics (`a + b`), or comparison (`a < b`).

Most of these operators can also be defined for custom data types like `PairOfInts`. The definition works like that of an ordinary function overload:

```
PairOfInts operator+(const PairOfInts& p1, const PairOfInts& p2)
{
  return PairOfInts{p1.a + p2.a, p1.b + p2.b};
}
```

Given this definition, the following two expressions produce the same result:

```
// function call syntax
Pair p3 = operator+(p1,p2);
// operator syntax
Pair p4 = p1 + p2;
```

Function Pointers

In addition to the usual pointers, C++ also knows *pointers to functions*, e.g., `int (*f)(int)`, a pointer called `f` for functions expecting and returning `int`. To simplify notation, the asterisk `*` and ampersand `&` may be omitted when referring to function pointers and addresses of functions.

Using function pointers, *functions may be used as arguments of other functions*:

```
int square(int i) {return i * i;}

int applyTwice(int f(int), int i) // optional "*" omitted
{
  return f(f(i)); // no "*" allowed in call syntax
}

// computes pow(7,4), optional "&" omitted when taking address
std::cout << applyTwice(square,7) << std::endl;
```

Example from DUNE

I have found only one instance of function pointers in DUNE (*but it's hard to search for...*).

This code creates *custom MPI parallel operation handles* for given data types and binary functions (specified as `Type` and `BinaryFunction` template parameters). The address of `operation` is used to pass the function the handle should represent.

A *C-style cast* is used to remove argument data types.

```
static MPI_Op get ()
{
  if (!op)
  {
    op = std::shared_ptr<MPI_Op>(new MPI_Op);
    MPI_Op_create((void (*)(void*, void*, int*,
        MPI_Datatype*))&operation,true,op.get());
  }
  return *op;
}

static void operation (Type *in, Type *inout,
    int *len, MPI_Datatype*)
{
  BinaryFunction func;

  for (int i=0; i< *len; ++i, ++in, ++inout) {
```

```
    Type temp;
    temp = func(*in, *inout);
    *inout = temp;
  }
}
```

The Main Function

The execution of a C++ program starts in the first line of a special function called `main`, and ends when its last line is reached. Every C++ program has to define *exactly one such function*.

The signature of the main function has to be one of

- `int main()` (this means command line arguments are ignored)
- `int main(int argc, char** argv)`
- `int main(int argc, char* argv[])`

The second and third variant have the same meaning: `argc` is the number of arguments, and `argv` an array of C-style strings.

The return value of `main` is an error code — not returning anything is equivalent to writing `return 0;` (implying success).

2.5. Templates

Often, one has to define the *same functionality for several different data types*. This can become tedious, both during initial implementation and when fixing bugs.

C++ provides a language feature for this, where all the different versions are auto-generated from a special construct, called a *function template*:

```
template<typename T>
  T square(T x)
  {
    return x * x;
  }
```

A function `square<T>` is then available for any type `T` that has a multiplication operator `*`:

```
int i    = square<int>(5);      // int version
float f  = square<float>(27.f)  // float version
double d = square<double>(3.14) // double version
```

Function definitions aren't the only use case for templates, one can also automate the generation of data types. These are known as *class templates*, since structs are a special case of classes in C++.

```
template<typename T>
  struct Pair {T a; T b;};

Pair<int> ip;   // a pair of ints
Pair<float> fp; // a pair of floats

// Pair<int> is a data type, and can be used as such
Pair<Pair<int>> ipp; // pair of pair of ints
```

Function templates and class templates were the only types of templates until C++14, when → *variable templates* were introduced.

Non-Type Template Parameters

Typically, one uses template parameters to introduce abstract types `T`, but there are also *non-type template parameters*. These can be used as compile-time constants:

```
// simple fixed-length vector
template<typename T, int n>
  struct Vector
  {
    enum {dim = n}; // knows its size
    T vals[n];
  }
```

An advanced use case for such non-type template parameters is → *template meta programming*, where function values and number sequences can be precomputed at compile time.

Template Template Parameters

In addition to types and values, one may also use templates themselves as arguments for other templates, so-called *template template parameters*. These can, e.g., be used to allow choosing between different implementations:

```
// accepts any template that expects a type and an int
template<template<typename,int> class V>
  double norm(const V<double,3>& vec);
```

One such level of templates-in-templates can be a very powerful tool, but you shouldn't overdo it:

```
template<template<typename> class U, typename T>
  struct Wrap {U<T> u;};

// just don't...
template<template<template<typename> class,typename> class W>
  struct Meta {W<Pair,double> w;};
```

Default Template Parameters

All three types of template parameters, i.e., types, non-type template parameters, and template template parameters, can have defaults, just like function arguments:

```
// use our vector struct by default
template<template<typename,int> class V = Vector>
  double norm(const V<double,3>& vec);

// generalization of our Pair struct
template<typename U, typename V = U>
  struct Pair {U a; V b;};

// provides special case for square matrices
template<int n, int m = n>
  struct Matrix
  {
    // "misuse" enum for defs at compile time
    enum {dim1 = n};
    enum {dim2 = m};

    // ...
  };
```

Parameters with defaults may be omitted, i.e., one can write `Pair<int>` (just as before!), or `Matrix<5>` for square matrices.

Renaming Types

C and C++ provide the keyword `typedef`, which may be used to *introduce new names for types*. It is actually a slight misnomer, since there is no real new definition[6]: they are akin to C++ references, but on the conceptual level of data types.

```
typedef unsigned long long int ull;
ull a = 12345678901234567890u; // huge unsigned int
```

While such new names can be introduced for any type, it is especially helpful for the types from template instantiations:

```
typedef Pair<Vector<double,3>,Vector<double,2>> VecPair3D2D;
```

In C++11 and above, consider using → *type aliases* instead, as they are more general and more readable.

Example from DUNE

A template with template template parameter and default parameters introducing

- an inner template `MatrixHelper`
- two typedefs: `size_type` and `type`
- an → *alias template* named `Pattern`

```
template<template<typename> class Container = Simple::default_vector,
         typename IndexType = std::size_t>
  struct SparseMatrixBackend
  {
    typedef IndexType size_type;

    //! The type of the pattern object passed to the GridOperator for pattern construction.
    template<typename Matrix, typename GFSV, typename GFSU>
    using Pattern = Simple::SparseMatrixPattern;

    template<typename VV, typename VU, typename E>
    struct MatrixHelper
    {
      typedef Simple::SparseMatrixContainer<typename VV::GridFunctionSpace,
              typename VU::GridFunctionSpace,Container, E, size_type> type;
    };
  };
```

[6] Note that `typedef struct{int a; int b;} PairOfInts;` *is* a valid definition, albeit a rather convoluted one.

Function Template Parameter Deduction

The specification of template parameters is often redundant when using function templates, because *type template parameters are readily deduced from the function arguments*. If this is the case, they can be omitted, and the call looks like a normal (non-template) function call:

```
template<typename T>
  T square(T x) {return x * x;}

int i    = square(5);   // short for square<int>
double d = square(27.); // short for square<double>
```

Note that sometimes this isn't possible:

```
// can't deduce return type from call
template<typename T>
  T generate();
```

2.6. Classes

Classes / Methods

The original name of C++ was "C with classes", so *classes, objects and object-oriented programming* are clearly an important part of C++.

While a classic C struct is simply an aggregation of data, C++ structs and classes typically contain *methods*, functions that are closely linked to the data and part of the type definition:

```
struct Vector2D
{
  std::array<double,2> comp;

  // const: method may be called for const vectors
  double norm() const
  {
    return std::sqrt(comp[0]*comp[0] + comp[1]*comp[1]);
  }
}

Vector2D v{{3,4}};
std::cout << v.norm() << std::endl; // prints 5
```

Access Specifiers

C++ provides three *access specifiers*:

`private`: accessible by *the object itself* and other objects of the same class

`protected`: like `private`, but additionally accessible in → *derived classes*.

`public`: always accessible

Sometimes it is helpful to exempt certain classes or functions from these restrictions using a `friend` *declaration*. This should be used sparingly, since it breaks encapsulation and exposes implementation details.

```
struct BankAccount
{
  // full access -- maybe not the best of ideas?
  friend class Customer;

  private:
    double balance; // hide this!
};
```

Encapsulation

The only difference between C++ structs and classes is default visibility: in a struct, everything is `public` by default (but may be declared `private`), and vice versa for classes.

The hiding of implementation details using `private` is known as *encapsulation* and is generally a good idea, mainly for the following reasons:

- *ensures consistent state:* in a plain struct, data is either `const` or open to arbitrary, potentially nonsensical, changes
- *facilitates modularity:* `private` components are unknown outside of the class itself, and may therefore be exchanged and modified at will
- *improves communication of intent:* anything marked `public` is part of the intended interface

Constructors

Data that was declared `private` is inaccessible outside of the class, so we need special `public` methods to initialize such data. These methods are called *constructors*. In C++, they have the same name as the class they belong to and are *the only functions without return type*.

```
class Vector2D
{
  std::array<double,2> comp;

  public:

  Vector2D() : comp{0,0} {};

  Vector2D(double a, double b)
    : comp{a,b}
  {};

  // ...
};
```

This (incomplete) class provides two constructors, one for arbitrary points, and one for the origin.

Why not use default arguments instead? Because then one could omit `b` while supplying `a` (design decision).

There are three types of constructors with special names:

- The *default constructor* `T()` without arguments: called, e.g., when mass producing `vector` entries during initialization
- The *copy constructor* `T(const T&)`: has the task of creating a copy of the object specified in the function call
- The *move constructor* `T(T&&)`: like the copy constructor, but may cannibalize the original object (which should be left in some valid default state)

Converting Constructors

Any constructor that is not marked as `explicit` is a so-called *converting constructor*, and is called for *implicit type conversions*[7]. Assume we have defined

```
Matrix(double); // diagonal matrix, or constant matrix?
operator*(double, const Matrix&);         // scaling
operator*(const Matrix&, const Matrix&); // matrix multiplication
```

[7] type promotions, like from `int` to `double` when needed

but not `operator*(const Matrix&, double)`. Then a call `a * 2.`, with `a` a `Matrix`, will call `Matrix(2.)` followed by matrix multiplication. This may lead to unexpected results.

If such conversions aren't intended, the constructor has to be marked:

```
explicit Matrix(double); // no implicit type conversions
```

Conversion Operators

Closely linked to constructors are *conversion operators*. While converting constructors are used to convert to the class type, these operators are used to convert *from* the class type to some other specified type, mainly when

- the target type is a fundamental type
- the target type is some class provided by an external library

In both cases it is impossible to simply provide the right constructor for the target type.

Here, conversion operators can be employed instead:

```
struct Vector
{
  operator Matrix(); // implicit promotion to column matrix
  explicit operator double(); // only explicit conversion
  ...
}
```

Delegating Constructors

Function overloading can be used to forward function calls to other versions of the same function, e.g., to swap function arguments, or as in the `maximum` function we introduced, or as a form of mutual recursion.

Similarly, one may define *delegating constructors* which call other constructors:

```
// constructor for square matrices uses general constructor
Matrix(int n)
  : Matrix(n,n) // no further entries allowed
{}
```

Rules:

- The delegating constructor may not initialize anything itself (only call this second constructor)
- The calls cannot be recursive (at some point, initialization has to take place)
- The function body is executed after the other constructor has finished (thereby allowing local modifications)

Destructors

Destructors are the counterpart to constructors: they clean up data when an object goes out of scope and its lifetime ends. Most of the time explicitly declaring a destructor isn't necessary, but it is vitally important if, e.g., memory was allocated by the object.

The name of the constructor is the class name with a tilde `~` as prefix.

The correct use of destructors leads directly to the technique of → *Resource Acquisition is Initialization (RAII)*.

```
class IntStorage
{
  int n;
  int* data;

  public:

  IntStorage(int n_)
    : n(n_), data(new int[n])
  {};

  // copies are neither forbidden
  // nor handled correctly
  // --> segmentation fault waiting to happen

  ~IntStorage()
  {
    delete data;
  }
};
```

Default Methods

For any type `T`, the compiler automatically generates several methods for us if applicable:

- *default constructor* `T()`
- *default destructor* `~T()`
- *copy constructor* `T(const T&)`

- *copy assignment* `T& operator=(const T&)`
- *move constructor* `T(T&&)`
- *move assignment* `T& operator=(T&&)`

In each case, the method is not created automatically if that is impossible, e.g., if the class is storing some reference, or if there are user-defined versions.

In the case of the default constructor `T()`, the presence of *any user-defined constructors* prevents its creation.

The move constructor and move assignment operator aren't created if *any of the other mentioned methods except the default constructor* has been user-defined.

The assignment `= default` as in

```
T() = default
```

can be used to explicitly state that not providing some method is actually intended and not a mistake, or force the generation of the default constructor in the presence of other constructors.

Rules of Zero and Five

The *rule of zero* states that it is often a good idea to implement *at most* some custom constructors and none of the aforementioned methods. This is concise and perfectly appropriate when the class is just a collection of data with some methods.

However, sometimes it is necessary to provide replacements for these methods, e.g., because the default copy methods perform *flat copies of pointer structures*.

The *rule of five*[8] states that if a user-defined copy constructor, copy assignment operator, move constructor, move assignment operator, or destructor is present, then it is very likely that all five should be explicitly defined: if one of these methods has to be specialized, then the underlying reason is typically relevant for all of them.

[8] formerly known as "rule of three", before move semantics were introduced

Deleted Methods

Returning to the `IntStorage` object, we have two different options:

- provide user-defined copy/move constructors and assignment operators to *enable deep copy semantics*
- simply *prohibit the creation of copies* of such objects

Let's go with the second option for the sake of argument. Before C++11, one would have implemented the methods to prevent their automatic generation, and then declared them `private`. Any attempt to copy such an object would then trigger a compilation error, but the construct is a rather poor choice in terms of communicating intent.

Since C++11, one can *declare methods as deleted*, preventing their automatic creation:

```
T(const T&) = delete;
T& operator=(const T&) = delete;
```

Mutable Members

In C++, declaring an object `const` doesn't mean that its representation (byte sequence) has to be immutable, just that any such changes are invisible from the outside. The keyword `mutable` marks members as modifiable in `const` methods:

```
class Matrix
{
    mutable bool detValid = false;
    mutable double det;

  public:

    double determinant() const
    {
      if (!detValid)
      {
        det = calcDet(); // expensive (private) helper function
        detValid = true;
      }
      return det;
    }
}
```

Note that any method that modifies the matrix has to set `detValid = false`!

Static Members

The keyword `static` indicates that something *belongs to the abstract definition, not the concrete instance*:

- In functions definitions, `static` variables belong to the function itself, not the current function call (and therefore persist across function calls)
- In class definitions, `static` variables and methods belong to the class itself, not the created objects (i.e., they are shared between all objects of this class)

In functions, `static` variables can serve as function "memory", e.g., in *function generators*. In classes, `static` members can be used to, e.g., count the number of instantiated objects, manage a common pool of resources (memory, threads, ...), or provide small private helper functions.

The Singleton Pattern

A *design pattern* using `static` is the *singleton pattern*, which creates a class that provides exactly one instance of itself and prevents the creation of further such objects.

This pattern tends to be overused. It should be restricted to situations where

- the notion of two such objects doesn't make sense under any circumstance
- the single instance needs to be accessible from everywhere in the program

Standard applications concern the centralized management of system resources:

- a centralized logging facility, printing queues, or network stacks
- thread and memory pools, or the graphical user interface (GUI)

Realization of the singleton pattern in C++:

```
class Singleton
{
  public:
    static Singleton& getInstance()
    {
      // kept alive across function calls
      static Singleton instance;
      return instance;
    }

    // prevent creation of copies
    Singleton(const Singleton&) = delete;
    void operator=(const Singleton&) = delete;

  private:
    // only callable within getInstance()
```

```
    Singleton(){...}; // hide constructor
};

// in user code:
Singleton& singleton = Singleton::getInstance();
```

Example from DUNE

An application of the singleton pattern in DUNE:

```
class MPIHelper
{
  public:
  ...

    DUNE_EXPORT static MPIHelper&
        instance(int& argc, char**& argv)
    {
      // create singleton instance
      static MPIHelper singleton (argc, argv);
      return singleton;
    }

  ...
  private:
  ...

    MPIHelper(int& argc, char**& argv)
    : initializedHere_(false)
    {...}
    MPIHelper(const MPIHelper&);
    MPIHelper& operator=(const MPIHelper);
};
```

- Conceptually, there can be only one instance of the *Message Passing Interface (MPI)*.
- Its initialization and finalization functions must be called *exactly once*, the singleton pattern guarantees this.
- Copy constructor and assignment operator have been hidden instead of deleted (pre-C++11 style).

2.7. Inheritance

Quite often, there is a natural relationship between several classes based on their purpose:

- Several matrix classes (dense vs. sparse, small vs. millions of entries, local vs. parallely distributed, hierarchically blocked or not,...)
- Different algorithms for matrix inversion / decomposition

- A range of solvers for nonlinear problems, spatial discretizations, time stepping schemes,...

This relationship can be expressed using *inheritance*:

- Extend and augment existing classes
- Collect and maintain common code in *base classes*
- Express and enforce interfaces through → *abstract base classes (ABCs)*

Inheritance establishes a relationship between classes, one *derived class* and an arbitrary number of *base classes* (typically just one).

```
struct A1 {int a};
struct A2 {int a};

struct B : A1, A2 // B inherits from A1 and A2
{
  int b;

  B(int c1, int c2, int c3) : b(c1), A1::a(c2), A2::a(c3)
};
```

The class `B` is an extension of both `A1` and `A2`, and contains three ints (two `a`s, and one `b`), since it inherits all data members and methods from `A` and `B`.

Inside of `B` the two different `a`s have to be accessed via `A1::a` and `A2::a`, because the simple name `a` would be ambiguous.

Class Hierarchies

Derived classes may themselves have derived classes, leading to a *hierarchy of data types*:

```
struct A {int i};
struct B : A {int j}; // B IS-A A
struct C : B {int k}; // C IS-A B (and therefore an A)
```

Again, `i` can be accessed in `C`, possibly under the name `B::A::i` if `i` alone would be ambiguous.

Conceptually, this hierarchy always forms a tree:

```
struct D : B, C {int l}; // contains two independent A and B each
```

`B::j` and `C::B::j` are two independent variables! We will see more about this when discussing the → *Deadly Diamond of Death* and → *virtual inheritance*.

Access Specifiers in Inheritance

Access specifiers can also be used when specifying inheritance relationships, as in

```
class A : public B {...};
```

If this access specifier is omitted, it defaults to `public` for structs and `private` for classes[9].

Access rights combinations for inherited methods and data members:

...is inherited...	`public`ly	`protected`ly	`private`ly
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	—	—	—

Public Inheritance

Public inheritance models the *subtype relationship* from entity-relationship models: a derived class object IS-A base class object, in the sense that it fulfills the same interface.

Possible examples of this are:

- A Circle IS-A Shape
- A DiagonalMatrix IS-A Matrix
- A NewtonSolver IS-A NonlinearSolver

An object of the derived class is expected to be a *perfect replacement* for objects of the base class. This puts additional responsabilities on the person implementing the derived class, e.g., code operating on pointers and references of the base class should continue to work for those pointing to objects of the derived class.

[9] This is the reason why "`: public`" is typically used instead.

The Liskov Substitution Principle

In principle, the subtype (derived class) should exhibit the same behavior as the supertype (base class). However, this is hard to verify in general. The *Liskov Substitution Principle (LSP)* defines some constraints that are meant to aid in this task.

Methods that share a name with one of the methods of the base class (and thereby *override* them) should not lead to surprising behavior:

Contravariance of arguments The method may also accept *supertypes of the original arguments*[10].

Covariance of return types The method may return a *subtype of the original return type*[11].

Exception safety The method may only throw the original → *exceptions* or subtypes thereof.

Additionally, the following things should hold:

Preconditions The subtype may not strengthen preconditions (put additional constraints on the environment where its methods are called).

Postconditions The subtype may not weaken postconditions (leave conditions unfulfilled that are always fulfilled by the supertype).

Invariants The subtype must honor invariants of the supertype (things that are generally true for its objects)

History constraint The subtype may not allow state changes that are impossible from the viewpoint of the supertype.

In short, the base class part of the derived class should perform according to expectations. `private` members becoming inaccessible in the derived class helps in this regard.

Is a Square a Rectangle?

According to the LSP, it depends on the concrete implementation whether a Square is indeed a Rectangle:

- If squares and rectangles are immutable (unchangable after their creation), or only provide a methods for scale adjustments and rotation/translation, then a Square can be a Rectangle.
- If changing the length of a Square would also change its width, then assumptions about the supertype Rectangle would be violated.
- Keeping the width constant instead means it fails at being a Square.

[10] not available in C++
[11] in C++ only in the context of → *dynamic polymorphism*

- Therefore, Squares cannot be Rectangles if the length and width of rectangles can be controlled independently.

Again: Is a DiagonalMatrix a Matrix?

Protected/Private Inheritance

Private inheritance implements one class in terms of another:

- the members of the base class become private members of the derived class
- this is invisible from outside of the class
- the derived class may access the public interface of the base class

Mostly equivalent to just storing the base class as a private member instead, except for so-called *empty base class optimization*.

Protected inheritance works the same, but the inheritance is visible to children of the derived class as well. Seldom used and few applications.

Examples from DUNE

DUNE currently contains

- of course a large number of instances of public inheritance,
- exactly one instance of protected inheritance,
- and a small handful of cases with private inheritance.

```
class MacroGrid
  : protected DuneGridFormatParser
{...}

template <int block_size, class Allocator=std::allocator<bool> >
class BitSetVector : private std::vector<bool, Allocator>
{...}

template<int k>
struct numeric_limits<Dune::bigunsignedint<k> >
  : private Dune::Impl::numeric_limits_helper
      <Dune::bigunsignedint<k> > // for access to internal state of bigunsignedint
{...}
```

- `MacroGrid` and descendants may access internal `DuneGridFormatParser`
- `BitsetVector` is implemented using `std::vector`, but one may not be used as replacement for the other

2.8. Code Structure

We have now reviewed the fundamental building blocks of C++, and finish this section with a short look at *code structure*. C++ supports a range of tools that can be used to structure code bases and make large collections of code more maintainable:

- *header files* and the accompanying *header guards* to structure the concrete file layout and maintain code dependencies
- *source files* to provide implementations separate from declarations, thereby guaranteeing stable interfaces while allowing modification of implementation details
- *namespaces* to structure code on the conceptual level and prevent name clashes between different libraries

C++20 will introduce → *modules* as known from other programming languages, which will significantly improve importing other C++ libraries and exporting one's own constructs as a library.

Header Files

By convention, C++ code is separated into *two different types of files*:

- header files (`.hh`), containing declarations and interfaces
- source files (`.cc`), containing definitions and implementations

Both types of files are needed to build a given code project, but only the header files are necessary when writting code that should link against some external library.

Templates typically go against this convention, with their complete definition put into header files: the definition is needed during template instantiation, and without it dependent code could only use a given, fixed set of predefined and preinstantiated variants.

Header Guards

The `#include` preprocessor statement simply includes raw text from header files, recursively if necessary:

- Typically, header files are included several times within a program (e.g., `<iostream>`, `<vector>`, etc.).
- This would lead to redefinitions, and therefore *compilation errors*.
- Even without such errors, reparsing of these files would lead to *long parse times*, especially when considering header files including other header files.

Therefore, use header guards:

```
#ifndef HEADER_HH // only read file if not yet defined...
#define HEADER_HH // ...and define to prevent second read

... // actual header content (only parsed once)

#endif // HEADER_HH // reminder what is closed here
```

Namespaces

The *one definition rule (ODR)* of C++ demands that names are unambiguous:

- local definitions take precedence over those from enclosing scopes
- providing two differing definitions for the same name with the same visibility is forbidden

This leads to problems when

- a certain name should be reused in different parts of a very large program
- by coincidence, two (or more) external libraries define the same name

Solution: encapsulate libraries, sublibraries, and independent project parts using *namespaces*.

A simple namespace example:

```
namespace Outer
{
  namespace Inner
  {
    struct Data{};
  }
}
```

In this example, the `struct Data` is known as

- `Data`, `Inner::Data`, or `Outer::Inner::Data` in the innermost scope
- `Inner::Data` or `Outer::Inner::Data` in the outer scope
- only `Outer::Inner::Data` in the rest of the program

Example from DUNE

In DUNE, a namespace called (fittingly) `Dune` encapsulates the whole project. This namespace is used for the core modules.

Downstream modules like PDELab typically introduce subnamespaces, e.g., `Dune::PDELab`, for their own classes and functions. This way, these modules may use names that would otherwise collide with each other or parts of the core modules.

```
#ifndef DUNE_PDELAB_NEWTON_NEWTON_HH
#define DUNE_PDELAB_NEWTON_NEWTON_HH
...
namespace Dune
{
  namespace PDELab
  {
    // Status information of Newton's method
    template<class RFType>
    struct NewtonResult : LinearSolverResult<RFType>
    {
      ...
    }
    ...
  } // end namespace PDELab
} // end namespace Dune
#endif // DUNE_PDELAB_NEWTON_NEWTON_HH
```

3. The Standard Library

The *Standard Library* is a set of classes and functions that is part of the C++ language standard. It provides most of the common "tools of the trade": data structures and associated algorithms, I/O and file access, exception handling, etc. The components are easily recognized because they are in the `std` namespace.

Large parts of the Standard Library were already available in C++98/03 and are based on the *Standard Template Library (STL)*, which is the library where common container classes like `std::vector` originate from. For this reason, it is still sometimes called "the STL".

Other parts were introduced in C++11 and later standards, often originating in the *Boost C++ Libraries*, a well-known set of open-source libraries that provide advanced utility classes and templates.

We are going to discuss the following older parts of the Standard Library in detail:

- input and output streams
- containers (a.k.a. data structures) and iterators
- algorithms and functors
- companion classes (pair and tuple)
- exceptions

C++11 additions like → *smart pointers*, → *random number generation*, → *threads*, and → *regular expressions* will be discussed at a later point, and the same holds for the → *filesystems* library from C++17.

Note that one usually has to include a certain header for a library feature to be available, e.g., `#include <iostream>` for I/O, or `#include <vector>` for `std::vector`.

3.1. Input / Output

C++ provides *stream-based I/O*: each stream is an abstraction for some source and/or destination of data, and each such stream is used in the same way, whether it represents standard C I/O, the content of a file, or simply the content of some string.

The relevant types are:

`[i,o,io]stream` generic input, output, or bidirectional I/O stream

`[i,o,io]fstream` specialization for reading / writing files

`[i,o,io]sstream` sprecialization for strings as data sources/sinks

There are four predefined global variables for standard C I/O:

`cin` standard C input stream

`cout` standard C output stream

`cerr` standard C error stream, unbuffered

`clog` standard C error stream, buffered

For larger programs it is good practice not to write to these streams directly, since extraneous output can make it difficult to use the software in new contexts and other projects. Instead, one writes into intermediate stream objects, which may then redirect the data at their discretion.

The standard way of using streams are the well-known `<<` *and* `>>` *operators*, maybe with some *I/O modifiers* like floating-point formatting. Additionally, there are also special types of → *iterators* available that make a more or less seamless interaction with container classes possible.

If it should be possible to read and/or write an object from/to a stream (*serialization*[12]), one has to specialize the stream operators.

- Use free functions, because the stream is the left-hand side operand
- `friend` declaration, because we need access to the object internals

```
struct Streamable
{
  // ...

  // return streams to facilitate operator chaining
  friend std::istream& operator>>(std::istream& is, const Streamable& s);
  friend std::ostream& operator<<(std::ostream& os, const Streamable& s);
};
```

Example from DUNE

```
//! Print a std::array
template<typename Stream, typename T,
    std::size_t N>
inline Stream& operator<<(Stream& stream,
    const std::array<T,N>& a)
{
  stream<<"[";
```

[12] https://isocpp.org/wiki/faq/serialization

```
  if(N>0)
  {
    for(std::size_t i=0; i<N-1; ++i)
      stream<<a[i]<<",";
    stream<<a[N-1];
  }
  stream<<"]";
  return stream;
}
```

This code defines an *output format for* `std::array` as long as the stored type `T` itself can be printed.

The stream is returned to the caller, which enables *operator chaining* as in

```
std::cout << a1
          << " "
          << a2
          << std::endl;
```

3.2. Container Classes

Sequential Containers

C++ provides three different types of *containers (data structures)*: sequences, container adaptors, and associative containers. They are the C++ versions of common and well-known data structures (array, linked list, tree, hash map, etc.).

The elements of container objects all have the same type, specified as a template parameter. This restriction can be somewhat lifted through → *dynamic polymorphism*. Since C++17 one may also use a → *variant*, or the → `std::any` *class* as element type.

The simplest type of container is a *sequence*, where the elements are associated with an integer index (either explicitly or implicitly).

C++ provides the following sequences:

array array with fixed size and random access

vector array with variable size and random access

deque double-ended queue with random access

list doubly linked list

forward_list singly linked list

```
std::vector<int> a(3,5); // array [5,5,5]
std::vector<int> b{3,5}; // array [3,5]
std::list<double> c;     // empty doubly-linked list
```

Container Adaptors

In C++, a *container adaptor* implements some data structure in terms of another. Three such adaptors are provided:

stack LIFO (last in, first out) structure

queue FIFO (first in, first out) structure

priority_queue a priority queue (surprise!)

```
std::stack<double> a;          // stack based on deque
std::stack<int,std::vector> b; // stack based on vector
std::queue<int,std::list> c;   // queue based on list
```

Associative Containers

In contrast to sequential containers, which are indexed by integers, *associative containers* may have an arbitrary index type. This index is associated with a value, and the container can be searched for this key, producing the value if the key is found.

C++ knows four *sorted associative containers*:

set key is unique and identical to its associated value

multiset like set, but key may appear multiple times

map key is unique, value is pair of index and some mapped type

multimap like map, but key may appear multiple times

The keys of these sorted containers need to be equipped with a *strict weak ordering relation*. The containers are typically implemented using *red-black trees*.

Additionally, C++ also provides *unsorted associative containers*. While the sorted ones typically use trees internally, the unsorted ones are usually based on *hash tables*.

Each sorted container has an unsorted counterpart:

unordered_set, *unordered_multiset*, *unordered_map*, and *unordered_multimap*.

These unsorted containers are more universally applicable, since they don't need the underlying ordering relation. However, the elements appear in random order (based on the hash function in practice).

Unsorted containers tend to be faster than sorted containers but may be worse if there are many collisions in the hash function. For certain applications the guaranteed worst-case performance of sorted containers may be an important feature.

Complexity Guarantees

An important part of the container library are the accompanying *complexity guarantees*. Any implementation of C++ has to provide methods with certain upper bounds on performance. For the most part, these follow naturally from the default underlying data structures.

- Accessing an element of a `vector` is in $\mathcal{O}(1)$, and adding an element at the end is amortized[13] $\mathcal{O}(1)$, worst-case $\mathcal{O}(N)$, where N is the number of elements.
- Accessing an element of a `list` is in $\mathcal{O}(N)$, and adding or deleting elements is $\mathcal{O}(1)$ anywhere in the list.
- Operations on sorted associative containers are typically in $\mathcal{O}(\log N)$.
- For unsorted associative containers this is replaced with amortized $\mathcal{O}(1)$, worst-case $\mathcal{O}(N)$.

A table with all the container methods is available online[14], and complexity guarantees are linked from there.

Sequence Test: Vector vs. List

Task by J. Bentley and B. Stroustrup (*Element Shuffle*):

- For a fixed number N, generate N random integers and insert them into a sorted sequence.

  Example:

  - *5*
  - *1*, 5
  - 1, *4*, 5
  - 1, *2*, 4, 5

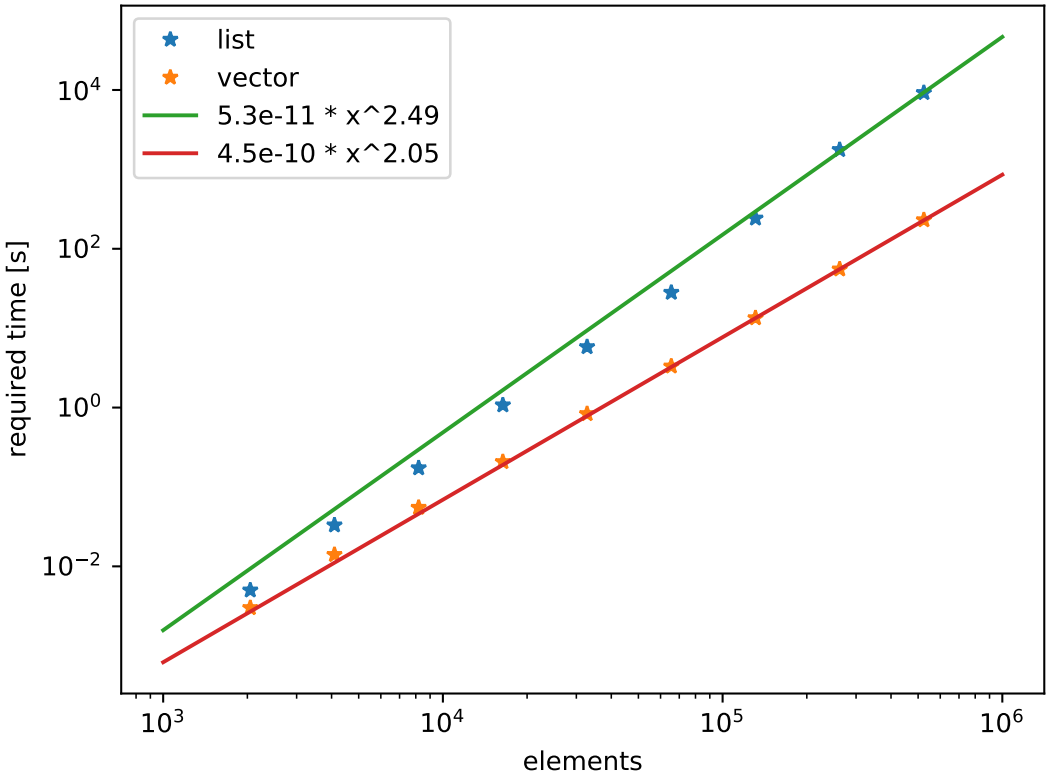
[13] averaged over many method calls
[14] https://en.cppreference.com/w/cpp/container

- Remove elements at random while keeping the sequence sorted.

  Example:

  - 1, *2*, 4, 5
  - 1, 4, *5*
  - *1*, 4
  - *4*

- For which N should a `list` be used, and in which cases a `vector`?



Potentially suprising results:

- Despite random insertion / deletion, `vector` is faster by an order of magnitude
- Linear search for both containers, despite bisection being available for `vector` (!)
- *Search completely dominates move required by* `vector`
- Non-optimized `list` performs one allocation / deallocation per element (!)

Use vector as default — and if not, back up assumptions with measurements

3.3. Iterators

The main way to interact with containers is via *iterators*, which are generalizations of the pointer concept, i.e., objects that can be dereferenced (`*`) and advanced (`++`). For each container type `T`, there is:

- an associated type `T::iterator`, for read / write access to the container
- an associated type `T::const_iterator`, providing read-only access
- a method `begin()`, which returns an iterator pointing to the first element
- a method `end()`, returning an iterator one position past (!) the last element
- equivalent methods `cbegin()` and `cend()` for read-only access

The element order is defined by the index for sequences, the keys for sorted containers, and the hash function for unsorted containers. There are also additional types / methods that reverse this order (`rbegin()`, `rend()`, etc.).

Using iterators, one can write functions that work for different containers:

```
template<typename T>
  void print(const T& cont)
  {
    for (typename T::const_iterator it
        = var.begin(); it != var.end(); ++it)
      std::cout << *it << " ";
    std::cout << std::endl;
  }
```

- The keyword `typename` is used inside templates to specify that a *dependent name* (identifier that depends on at least one template parameter) refers to a type, since the compiler could mistake it for a variable name, etc.
- The *prefix increment operator* is usually more efficient than the postfix one, since there is no need for temporary iterator copies.

The properties of iterators depend on the underlying container:

array, vector, deque Bidirectional (`++` / `--`), random access (i.e., instant jumps of arbitrary stride possible)

list Bidirectional, no random access (must follow pointer chain)

forward_list Forward direction only, neither backward direction nor random access

sorted assoc. cont. See `list`

unsorted assoc. cont. See `forward_list`[15]

The iterators provide their properties in the form of *iterator tags* (public members), which may be used to, e.g., write more efficient → *template specializations* of algorithms for iterators that provide random access.

3.4. Algorithms

The Standard Library provides a large number of *algorithms*[16] that are tailored to these different iterator categories, and automatically make full use of the capabilities of the container they are operating on.

Example algorithms:

for_each apply some function to each element (*lifting*)

count_if count elements with certain properties

find_if find first element with such property

copy_if insert applicable elements into other container

shuffle randomly re-order container contents

sort sort container according to some criterion

Try to use predefined algorithms instead of writing your own function templates

Many of these algorithms expect some criterion, transform, or operation, which has to be supplied as a *functor* (function object): an object that has an `operator()` with the required number of arguments.

The Standard Library provides some default functors, e.g., `std::less`:

```
// possible implementation of std::less
template<typename T>
  struct less
  {
    // first set of () is operator name
    bool operator()(const T& lhs, const T& rhs) const
    {
      return lhs < rhs;
    }
  }
```

[15] mainly because their order is arbitrary anyways
[16] full list: https://en.cppreference.com/w/cpp/algorithm

User-implemented functors can be used to customize the provided algorithms, or one can use → *lambda expressions* or function pointers instead.

3.5. Companion Classes

The Standard Library also provides a number of class templates that are not containers, but serve similar purposes as containers and are often used in conjunction with them.

`std::pair<T,U>`:

The official version of the `Pair` struct we defined ourselves. Contains a `T first` and a `U second`.

`std::tuple<T...>`:

Generalization of `std::pair` to general tuples using → *variadic templates*. Member access through a `std::set` function, which expects either the type of the component as template parameter, or its index if the former is ambiguous.

C++17 adds → *optional*, → *variant*, and → *any* to the list.

3.6. Exceptions

C++ knows several different mechanisms for error handling:

`assert`:

Runtime check of some condition that should always be fulfilled (*sanity check*). Aborts the program if condition evaluates to `false`.

`static_assert`:

Compile-time check with similar purpose, see → *template meta programming*. Produces compilation error if condition is not met.

Exceptions:

Error handling mechanism for situations that should not be the norm, but *may sporadically occur during normal operation*: memory exhausted, file not found, matrix is singular, solver failed to converge...

An *exception* is an arbitrary object that can be interpreted as an *alternative return value*, delivered using a mechanism that differs from the standard `return`. The Standard Library provides some predefined exceptions for convenience, like `std::domain_error` and `std::range_error`, and new exceptions may be defined by inheriting from `std::exception`.

```
double nthroot(double x, int n)
{
  if (n % 2 == 0 && x < 0)
    // throw statement: execution of function is stopped here...
    throw std::domain_error("even powers require non-negative argument");
  ...
}

try // try block: register for treatment of potential exceptions
{
  double y = nthroot(-5.,2);
}
catch(std::domain_error e) // catch block: ...program control jumps here
{
  // try to do something better than just printing a message in practice
  std::cout << "nthroot failed, message: " << e.what() << std::endl;
}
```

Points to consider:

- Exceptions are for error conditions that *can't be handled locally*
- A `return` always returns to the immediate caller, but a `throw` *unwinds the call stack* until a matching `catch` block is found
- If none is found at all, the program is aborted (should be avoided if possible)
- All function calls between the `throw` statement and the `catch` block are stopped prematurely

This means that *local resources have to be handled* in those intermediate functions (allocated memory, open file handles, ongoing communication) during stack unwinding. An elegant mechanism to do this automatically is → *RAII*.

4. Advanced Topics

After having reviewed the basic building blocks of C++, i.e., the fundamental concepts and at least parts of the Standard Library, we discuss more advanced topics:

- template specializations
- interactions between templates and inheritance
- Resource Acquisition is Initialization (RAII)
- template metaprogramming
- dynamic polymorphism (virtual functions)
- static polymorphism (CRTP)
- Substitution Failure is not an Error (SFINAE)

4.1. Template Specialization

The main idea of templates is the reduction of code duplication through generalization, but sometimes there are special cases that should / have to be treated differently. This can be done by providing an *explicit template specialization*:

```
// make sure that int pointers are safe
template<> // empty set of remaining parameters
  struct Pair<int*> {int* a = nullptr; int* b = nullptr;};
```

For class templates[17], C++ additionally allows partial *template specialization*, where the template parameters are constrained but not fully specified:

```
// make sure pointers are safe
// note local change in meaning for U and V!
template<typename U, typename V>
  struct Pair<U*,V*> {U* a = nullptr; V* b = nullptr;};
```

Which version is chosen for a given instantiation, e.g., `Pair<int*,int*>` *?*

[17] and variable templates since C++14

```
// 1) base template: general version
template<typename U, typename V = U>
  struct Pair;

// 2) partial specialization: U = V
template<typename U>
  struct Pair<U,U>;
// or shorter: Pair<U>

// 3) partial specialization: pointers
template<typename U, typename V>
  struct Pair<U*,V*>;

// 4) full specialization: int pointers
template<>
  struct Pair<int*,int*>;
// again, alternatively Pair<int*>
```

C++ always chooses the *most specialized* version:

- `Pair<int*,int*>` and `Pair<int*>` are (4), the latter via default argument in (1)
- `Pair<int,int>` and `Pair<int>` are both (2)
- `Pair<int*,double*>` is (3)

But `Pair<double*,double*>` and `Pair<double*>` are ambiguous, both (2) and (3) would fit!

Avoid overlapping specializations — they cause compiler errors when triggered

Things are slightly more complicated for function templates. Assume for a moment that we have a function template for matrix-vector products:

```
template<typename Mat, typename Vec>
  Vec multiply(const Mat& mat, const Vec& vec);
```

If we had a class called `SparseMatrix` for sparse matrices, i.e., matrices where almost all entries are zero, this generic function would likely be very inefficient for such a matrix. It makes sense to provide a partial template specialization:

```
template<typename Vec>
  Vec multiply<SparseMatrix,Vec>
      (const SparseMatrix& mat, const Vec& vec);
```

Unfortunately, this isn't possible in C++.

A full specialization would be allowed, and we could even omit the parameters:

```
template<>
  VectorClass multiply<SparseMatrix,VectorClass>
      (const SparseMatrix& mat, const VectorClass& vec);

// short version
template<>
  VectorClass multiply
  (const SparseMatrix& mat, const VectorClass& vec);
```

Alas, we would have to specialize for any of possibly many vector classes we would like to use together with `SparseMatrix`.

But why is that the case? Why can't we simply provide a partial specialization?

Dimov/Abrahams Example

Consider the following two code snippets[18]:

```
// (1) first base template
template<typename T> void f(T);

// (2) second base template (overloads)
template<typename T> void f(T*);

// (3) full specialization of (2)
template<> void f(int*);
```

```
// (1') first base template
template<typename T> void f(T);

// (3') full specialization of (1')!
template<> void f(int*);

// (2') second base template (overloads)
template<typename T> void f(T*);
```

(2) and (2') could be both a simple overload for, or a specialization of, (1) resp. (1'). In C++, the former is the case.

Now, consider the call `f(p)` for an `int* p`. This calls (3) as expected, but (2') for the second snippet[19]! *Why?*

[18] see http://www.gotw.ca/publications/mill17.htm
[19] interactive version: link to godbolt.org

Because in C++, *overloads are resolved using base templates* and normal functions, and *then* specializations are taken into account!

As we have seen, even *full* function template specializations can lead to counterintuitive results, which may explain why partial ones are currently not part of the language.

This is a common issue with C++, where a newer feature (here: templates) has to take older ones (here: overloading) into account. The growth of C++ can be likened to *onion layers*, or *strata of subsequent civilizations*, and newer additions have to interact with well-established features.

This is also the reason why objects have an implicit self-reference pointer called `this`, and not a reference with that name: references were introduced after classes.

A particularly involved example we will consider throughout the lecture is compile-time template selection, which started with → *SFINAE*, is supported via → `enable_if` in C++11, → `enable_if_t` in C++14, and → `if constexpr` in C++17, and will be a standard application of → *concepts* in C++20.

Template Specialization (cont.)

Conclusions drawn from the aforementioned observations:

Class template specializations

These are perfectly fine, whether explicit (full) or partial specializations.

Function template specializations

Partial specializations are forbidden, use a helper class with partial specialization or incorporate the function as a method instead. Explicit specializations may interact in counterintuitive ways and are completely unnecessary: all types are fully specified, so simply provide a normal function overload instead.

"Use specializations for class templates, and overloads for function templates"

What about our motivating example, the sparse matrices?

We have the following options:

- Use overloading for one of the arguments, and templates for the other:

```
template<typename Vec>
  Vec multiply(const SparseMatrix& mat, const Vec& vec);
// + versions for each alternate matrix class
```

- Variant: make the `multiply` function a method for each of the matrix classes, maybe with some default that can be inherited from some base class.
- Hand computation over to some *helper class template, which may freely use partial specialization*:

```
template<typename Mat, typename Vec>
  Vec multiply(const Mat& mat, const Vec& vec)
  {
    return multHelper<Mat,Vec>::multiply(mat,vec);
  }
```

4.2. Templates and Inheritance

Classes and Templates: Interactions

The combination of object-oriented programming (inheritance) and generic programming (templates) leads to complications during name lookup that have to be studied in detail. C++ treats *dependent and independent base classes* in different ways.

Independent base classes are those base classes that are *completely determined without considering template parameters*, and independent names are unqualified names that don't depend on a template parameter.

- Independent base classes behave essentially like base classes in normal (non-template) classes.
- If a name appears in a class but no namespace precedes it (an unqualified name), then the compiler will look in the following order for a definition:
  1. Definitions in the class
  2. Definitions in independent base classes
  3. Template arguments
- This order of name lookup can lead to surprising results.

```cpp
#include <iostream>

struct Base {typedef int T;};

template<typename T>
  struct Derived : Base
  {
    T val;
  };

int main()
{
  Derived<double> d;
  d.val = 2.5;
  std::cout << d.val << std::endl;
}
```

- This program prints `2`, not `2.5` as it may seem: link to godbolt.org
- `T` is not defined in the class, but in the independent base class
- Therefore, the template argument is ignored, and `T` is `int`! Main issue: this can't be seen when looking at `Derived`!
- *Use different naming schemes for types and type placeholders (template parameters)*

Dependent base classes are those that are not independent, i.e., they *require the specification of at least one template argument to be fully defined.*

- The C++ standard dictates that independent names appearing in a template are resolved at their first occurrence.
- The strange behavior from the last example relied on the fact that independent base classes have higher priority during name lookup than template parameters.
- However, for names defined in a *dependent* base class, the resolution would depend on one or more template parameters, unknown at that point.
- This would have consequences when an *independent* name would have its definition in a *dependent* base class: it would have to be looked up before that is actually possible.

```cpp
template<typename T> struct Base {int n;};

template<typename T>
  struct Derived : public Base<T>
  {
    void f()
    {
      // (1) Would lead to type resolution
```

```
      //     and binding to int.
      n = 0;
    }
  };

template<>
  struct Base<bool>
  {
    // (2) Template specialization wants to
    //     define variable differently.
    enum {n = 42};
  };

void g(Derived<bool>& d)
{
  d.f();  // (3) Conflict
}
```

1. In the definition of class `Derived<T>`, the first access to `n` in `f()` would lead to binding `n` to `int` (because of the definition in the base class template).

2. Subsequently, however, the type of `n` would be modified into something unchangeable for the type `bool` as template parameter.

3. In the instantiation (3) a conflict would then occur.

In order to prevent this situation, C++ defines that *independent* names won't be searched in *dependent* base classes. Base class attribute and method names must therefore be made dependent, so that they will only be resolved during instantiation (*delayed type resolution*).

Instead of simply writing `n = 0;`, use:

- `this->n = 0;` (implicitly dependent through `this` pointer)
- `Base<T>::n = 0;` (explicitly mentions template parameter)
- or import `n` into the current namespace:

```
template<typename T>
  struct Derived : Base<T>
  {
    // name is now dependent for whole class
    using Base<T>::n;
    ...
  };
```

4.3. RAII: Resource Acquisition is Initialization

Multiple Resource Allocation

Often, especially in constructors, resources must be allocated several times in succession (opening files, allocating memory, entering a lock in multithreading):

```
void useResources()
{
  // acquire resource r1
  ...
  // acquire resource r2
  ...
  // acquire resource rn
  ...

  // use r1...rn here

  // release in reverse
  // release resource rn
  ...
  // release resource r1
  ...
}
```

- If acquiring r_k fails, $r_1, \ldots, r_{k-1}$ have to be released before cancellation is possible, otherwise a resource leak is created.
- What should be done if allocating the resource throws an exception that is caught outside? What happens to $r_1, \ldots, r_{k-1}$?

Common variant of this problem:

```
class X
{
  public:
    X();
    ~X();

  private:
    A* pointerA;
    B* pointerB;
    C* pointerC;
};
```

```
X::X()
{
  pointerA = new A;
  pointerB = new B;
```

```
  pointerC = new C;
}

// How can we guarantee that
// pointerA is freed if
// allocating pointerB or
// pointerC fails?
```

RAII

In C++, the correct solution for this problem is called *"Resource Acquisition is Initialization" (RAII)*, which:

- is based on the properties of constructors and destructors and their interaction with exception handling.
- is actually a misnomer: *"Destruction is Resource Release" (DIRR)* would be more appropriate, but the acronym RAII is now too well-known to change it.

RAII is a specific way of thinking about resources that originated in C++ and provides an elegant alternative to strategies used in Java or Python, etc. (and has a really unfortunate name for something so central...).

RAII: Rules for Constructors and Destructors

Central rules that enable RAII:

1. An object is only fully constructed when its constructor is finished.
2. A compliant constructor tries to leave the system in a state with as few changes as possible if it can't be completed successfully.
3. If an object consists of sub-objects, then it is constructed as far as its parts are constructed.
4. If a scope (block, function...) is left, then the destructors of all successfully constructed objects are called.
5. An exception causes the program flow to exit all blocks between the `throw` and the corresponding `catch`.

The interplay of these rules, especially the last two, automatically frees resources before leaks can happen, even when unexpected errors occur.

```
template<typename T>
  class Ptr
  {
    public:
      Ptr()
      {
        pointerT = new T;
      }

      ~Ptr()
      {
        delete pointerT;
      }

      T* operator->()
      {
        return pointerT;
      }

    private:
      T* pointerT;
  };
```

```
class X
{
  // no constructor and destructor
  // needed, the default variant
  // is sufficient
  private:
    Ptr<A> pointerA;
    Ptr<B> pointerB;
    Ptr<C> pointerC;
};

int main()
{
  try
  {
    X x;
  }
  catch (std::bad_alloc)
  {
    ...
  }
}
```

(This is actually a simple mock-up of → *smart pointers*)

Basic principle:

- The constructor `X()` calls the constructors of `pointer{A,B,C}`.

- When an exception is thrown by the constructor of `pointerC`, then the destructors of `pointerA` and `pointerB` are called and the code in the `catch` block will be executed.
- This can be implemented in a similar fashion for the allocation of other resources (e.g. open files).

Main idea of RAII:

- Tie resources (e.g., on the heap) to handles (on the stack)[20], and let the scoping rules handle safe acquisition and release
- Repeat this recursively for resources of resources
- Let the special rules for exceptions and destructors handle partially-constructed objects

4.4. Template Metaprogramming

Template metaprogramming refers to the use of templates to perform computations at compile-time. This comes in basically two flavors:

- Compute with numbers as usual, but during the compilation process
- "Compute" with types, i.e., automatically map some types to other types

The former precomputes results to speed up the execution of the finished program, while the latter is something that is impossible to achieve during runtime.

Template metaprogramming can't make use of loops and is therefore *inherently recursive when performing nontrivial computations*, and may become arbitrarily complex (it is Turing complete!). We will only look at some small introductory examples.

→ *SFINAE* can be seen as a special case of template metaprogramming. → *Constant expressions* can often serve as a modern replacement for template metaprogramming, especially since loops in `constexpr` functions have been added in C++14.

Compile-Time Computations

An example of compile time single recursion to compute the factorial:

[20] heap: anonymous memory, freely allocatable; stack: automatic memory for local variables

```
template<int N>
  struct Factorial
  {
    enum {value = Factorial<N-1>::value * N};
  };

// base case to break infinite recursion
template<>
  struct Factorial<0>
  {
    enum {value = 1};
  };
```

In modern C++, this can be simplified significantly using → *variable templates*, because one doesn't need enums or static attributes for the values anymore.

Recursive Type Construction

Automated type generation using template metaprogramming:

```
template<typename T, int N>
  struct Array : public Array<T, N-1>
  {
    template<int M>
      T& entry()
      {
        return Array<T, M+1>::val;
      }

    // hide implementation detail
    protected:
      T val;
  };

// empty base case
template<typename T>
  struct Array<T,0> {};

// use like this:
Array<double,3> a;
a.entry<0>() = 0;
a.entry<1>() = 1;
a.entry<2>() = 2;
```

- *Recursive definition*: array of N elements is array of $N-1$ elements plus additional value
- Helper method for element access
- Nice feature: going beyond array bounds triggers compilation error

- Storing different types as in `std::tuple` would require → *variadic templates* and be significantly less straight-forward to implement
- Also see `dune-typetree`, a library for compile-time construction and traversal of tree structures, link: gitlab.dune-project.org/staging/dune-typetree

4.5. Dynamic Polymorphism

Inheritance is based on *new code utilizing old code*: we augment an existing class with new data/methods, and these can make use of the interface and/or implementation of the base class.

The main idea behind *dynamic polymorphism (subtyping)* is trying to make the inverse work: have *old code utilize new code*, i.e., we want to inject new behavior into classes and function without modifications to existing code.

Here, the concept polymorphism (greek: "many forms") refers to several functions sharing a common interface, with the concrete variant that is chosen depending on the provided arguments. The desired injection of new behavior is achieved by *making this selection independent of the function call site*.

Types of Polymorphism

There are different types of polymorphism:

- *Static polymorphism*, with the function being chosen at compile-time:
  - *Ad-hoc polymorphism*, in the form of function and operator overloading
  - *Parametric polymorphism*, in the form of templates and specializations
  - "True" → *static polymorphism*, trying to emulate dynamic polymorphism using template metaprogramming

  Also known as *early binding*, i.e., during program creation.
- *Dynamic polymorphism*, with the function being chosen at runtime (also known as *late binding*).

In C++, static polymorphism is *multiple dispatch* (the combination of types determines the chosen variant), while dynamic polymorphism is always *single dispatch*[21] (only depends on the object itself, not the method arguments).

[21] Dynamic multiple dispatch exists, e.g., in the Julia language.

Slicing

Using the copy constructor or assignment operator of a base class on some object results in *slicing*: anything belonging to the derived class is cut away, and only the base class part is used in the assignment.

Something similar happens when a *base class pointer or base class reference referring to an object of the derived class* is created: only the base class methods and attributes are accessible through such pointers and references. If the derived class redefines certain methods, then the base class version is used anyways, i.e., *the pointer/reference type dictates behavior, not the type of the object itself.*

Polymorphic Types

Polymorphic types are classes that have at least one method defined as `virtual`. For such methods, the type of the object itself determines which version is actually called, not the type of references or pointers:

```
struct Base
{
  virtual void foo() {...};
};

struct Derived : public Base
{
  void foo() override {...};
}

Derived d;
Base& b = d;
b.foo(); // calls foo() of Derived
```

- In C++, methods have to be explicitly declared as `virtual` for this to work. In some other languages this behavior is actually the default, e.g., in Java.
- There is no reason to redeclare the same method as `virtual` in derived classes, this happens automatically. But one may use `override` to state that this method should override a base class method, and will get a compilation error if this doesn't actually happen.
- Without the `override` qualifier the compiler would silently introduce an overload if the signature doesn't match, and consequently the base class method might be called when that isn't expected to happen.

Implementation Detail: vtables

Using polymorphic types, i.e., virtual functions, incurs a certain runtime cost: the concrete version to use for a certain function call has to be decided at runtime, because it depends on the actual type of the object pointed / referred to (that's the whole point).

A standard way of implementing virtual functions is via *vtables (dispatch tables)*:

- Each class with at least one virtual method stores *hidden static tables of virtual functions* ("vtables"), one for each base class with virtual methods, and potentially one for the class itself.
- These tables contain *function pointers to the right method versions.*
- Each object of such a class contains *hidden pointers to the relevant vtables.*
- These are inherited from the base class and therefore remain after slicing, etc., but are *redefined to point to the local version of the tables.*

```
struct Base1
{
  // creates entry in vtable
  virtual void foo1();
}

struct Base2
{
  // creates entry in vtable
  virtual void foo2();
}

struct Derived : Base1, Base2
{
  // changes entry in vtable
  void foo2() override;
  // creates entry in vtable
  virtual void bar;
}

Derived d;
Base2& b2 = d;
// follow two pointers for call here
// (p. to table and p. to function)
b2.foo2();
```

- The class `Derived` contains three implicitly defined static tables, one for itself, and one for the two base classes each.
- The table from `Base1` is copied, but that of `Base2` is changed locally, with its entry pointing to `Derived::foo2` instead.
- The call `b2.foo2()` accesses the vtable through the hidden pointer, and then uses the function pointer to `Derived::foo2` it finds there.

- Cost for lookup: follow two pointers (relevant when the method is very simple and called very often)

Virtual Destructors

Polymorphic types can be stored in containers and similar classes via pointers to base classes, and retain their specialized behavior. This makes it possible to *use containers for heterogeneous collections of objects*, as long as they all have a common base class.

However, the container would trigger the destructor of the base class when it goes out of scope, not the destructors of the derived classes. For this reason it is common practice to *declare a public virtual destructor when at least one other virtual method is present*, to ensure that the destructors of the derived classes are called.

Note that this suppresses the automatic generation of copy/move constructors and operators, but normally directly copying polymorphic types isn't a good idea anyways.

Copying Polymorphic Types

If a polymorphic object is copied when accessed through a base class pointer, the base class constructor is used. This means that unintended slicing occurs: only the base class part is copied, and virtual method calls revert back to the version of the base class.

The desired behavior would usually be a full copy of the object, i.e., based on its true type and consistent with dynamic polymorphism. This would require *something like a "virtual constructor" that constructs the correct type*. But constructors can't be virtual, because they are not tied to objects — they are part of the class itself, like static methods.

The standard solution to this problem is:

- explicitly forbid copying (and moving) polymorphic objects
- provide a special *clone method*, that serves the purpose of such virtual constructors, but operates on the level of pointers

```
class Base
{
  // define copy/move constructors
  // and operators here

  public:
    virtual Base* clone() const
    {
      return new Base(*this);
    }
```

```
    // virtual destructor
    virtual ~Base() {}
};

class Derived : public Base
{
  // as above

  public:
    Derived* clone() const override
    {
      return new Derived(*this);
    }
};
```

- Calling the `clone` method on a `Base` pointer will create a copy of the correct type and return a pointer to it.
- Using *covariant return types (see LSP)* we may return a pointer to the actual type.
- This pattern doesn't follow RAII at all. This can be changed using → *smart pointers*, but then a pointer to the base class has to be used throughout, since smart pointers of covariant types are not themselves covariant.

Abstract Base Classes

Abstract base classes (ABCs) are base classes that have at least one method declared as *purely virtual*, i.e., declared as a virtual function, but without actually providing a default implementation:

```
struct ABC
{
  virtual void foo() = 0; // pure virtual: no definition provided
  virtual ~ABC() {} // virtual destructor
};
```

Abstract base classes are used to define interfaces, because of the following two properties:

- It is impossible to instantiate objects of such a class, because at least one method is missing a definition.
- Every derived class has to provide such a definition to become instantiable.

Example from DUNE

```
template<class X, class Y>
class Preconditioner {
  public:
    //! \brief The domain type of the preconditioner.
    typedef X domain_type;
    //! \brief The range type of the preconditioner.
    typedef Y range_type;
    //! \brief The field type of the preconditioner.
    typedef typename X::field_type field_type;

    //! \brief Prepare the preconditioner. (...)
    virtual void pre (X& x, Y& b) = 0;

    //! \brief Apply one step of the preconditioner (...)
    virtual void apply (X& v, const Y& d) = 0;

    //! \brief Clean up.
    virtual void post (X& x) = 0;

    ...

    //! every abstract base class has a virtual destructor
    virtual ~Preconditioner () {}
};
```

Abstract base class for preconditioners:

- defines some types
- declares some methods, but doesn't provide implementations
- includes virtual destructor

Multiple Inheritance

Multiple interfaces can be combined by inheriting from several ABCs:

```
// VectorInterface: interface for vector types
// FunctionInterface: interface for functors
class Polynomial : public VectorInterface, FunctionInterface
{
  // define any methods required by the ABCs
}
```

`VectorInterface` might define (but not implement) all the usual methods for vector arithmetics, while `FunctionInterface` would require an appropriate `operator()`.

In the above code example, `FunctionInterface` is not a template, and therefore would describe the usual functions of a single real variable, but it wouldn't be difficult to provide a

class template as ABC instead. This would also cover more general functions (and technically define a parameterized family of ABCs).

Multiple inheritance is simple for ABCs, because they typically don't contain data. Therefore, the interface conditions are simply restated, i.e., this form of multiple inheritance is perfectly fine.

Multiple inheritance of base classes containing data, however, may lead to duplication of data members. To avoid this, *virtual inheritance* can be used: the derived class contains one copy of the base class per non-virtual derivation, and a single one for all virtual derivations combined.

This diamond pattern, sometimes called *Deadly Diamond of Death*, typically leads to code that is hard to maintain and may contain subtle bugs:

- Forgetting one of the `virtual` specifiers silently creates a second copy of the base class data.
- Accessing the data in this second unmaintained version by accident will make the state of the derived object inconsistent.

Example from DUNE

In practice, the diamond pattern is discouraged because of the resulting high maintenance cost. However, earlier versions of PDELab contained a Newton method based on this pattern that may serve as demonstration.

A Newton method consists of:

- a basic algorithm
- steps that must be performed at the start of each Newton iteration (e.g. reassembly of the Jacobi matrix)
- a test whether the process has converged
- optionally a linesearch to enlarge the convergence area

Each of these intermediate steps is outsourced to a separate class, so you can replace all the components independently.

The common data and virtual methods are placed in a base class.

```
class NewtonBase // stores data to operate on, iteration count, etc.
{...};

// perform linear solve, compute step direction
class NewtonSolver : public virtual NewtonBase
{...};

// check termination criterion
class NewtonTerminate : public virtual NewtonBase
{...};

// perform line search strategy
class NewtonLineSearch : public virtual NewtonBase
{...};

// local linearization (jacobian), thresholds, etc.
class NewtonPrepareStep : public virtual NewtonBase
{...};

// combine above classes into one complete class
class Newton : public NewtonSolver, public NewtonTerminate,
               public NewtonLineSearch, public NewtonPrepareStep
{...};
```

The actual implementation combined this with templatization on all levels.

4.6. Static Polymorphism

Just as dynamic polymorphism refers to the ability of code to adapt to its context *at runtime*, with dynamic dispatch on the type of objects, *static polymorphism* refers to polymorphism at compile-time with similar goals.

We have already discussed two versions of such static polymorphism:

- function and operator overloading
- templates and their specializations

Older code may then be adapted to new types by adhering to the relevant interfaces. But there is also a *specific pattern for static polymorphism that mimics virtual function calls*, but resolved at compile-time: base class templates using the curiously recurring template pattern.

In the *Curiously Recurring Template Pattern (CRTP)*, some class is used as a template parameter of its own base class. This is actually valid C++, because the full definition of the derived class isn't required at that point, only during instantiation.

```
template<typename T>
  class Base
  {
    // access to members of T through template parameter
    ...
```

```
  };

class Derived : public Base<Derived>
{
  ...
};
```

Also sometimes called *Upside-Down Inheritance*, because class hierarchies can be extended through different base classes using specialization.

```
// base class: provide interface, call impl.
template<typename T>
  struct Base
  {
    static void static_interface()
      {T::static_implementation();}

    void interface()
      {static_cast<T*>(this)->implementation();}
  };

// derived class: provide implementation
struct Derived : public Base<Derived>
{
  static void static_implementation();
  void implementation();
}

// call this with Derived object as argument
template<typename T>
  void foo(Base<T>& base)
  {
    Base<T>::static_interface();
    base.interface();
  }
```

- `static_cast` converts pointer types at compile-time, is type-safe (i.e., only works if `Base` object is actually part of a `T` object)
- *Base class provides interface definition* like in ABCs
- *Avoids cost of virtual functions*
- Not a full replacement for dynamic polymorphism, e.g., no common base class as needed for STL containers

Example from DUNE

Mixin to define finite element Jacobian in terms of residual evaluations, with `Imp` being both template parameter and derived class:

```
template<typename Imp>
class NumericalJacobianVolume
{
public:
  ...

   //! compute local jacobian of the volume term
   template<typename EG, typename LFSU, typename X, typename LFSV, typename Jacobian>
   void jacobian_volume (const EG& eg, const LFSU& lfsu, const X& x,
                         const LFSV& lfsv, Jacobian& mat) const
   {
     ...
     asImp().alpha_volume(eg,lfsu,u,lfsv,downview);
     ...
   }

private:
  const double epsilon; // problem: this depends on data type R!
  Imp& asImp () { return static_cast<Imp &> (*this); }
  const Imp& asImp () const { return static_cast<const Imp &>(*this); }
};
```

4.7. SFINAE: Substitution Failure is not an Error

SFINAE

We have already discussed how the compiler chooses between several available template specializations: it picks the "most specialized" version for which the template parameters lead to successful instantiation.

During this selection process, other (more specialized) versions may have to be tried out, but ultimately rejected when substituting the parameters with the given types fails. Therefore:

"Substitution Failure is not an Error" (SFINAE)

Failing to instantiate one of the specializations doesn't terminate the compilation process, that happens only when the pool of possible choices has been exhausted and no viable specialization was found, or several that are equally suitable.

SFINAE, the programming technique with the same name, provides a mechanism to *select between different template specializations at will*, and achieves this by *triggering substitution failures on purpose*.

The main tool for this is a small template metaprogram named `enable_if`, which provides a type definition or doesn't, depending on some external condition:

```
// possible implementation of enable_if
// ``false case'' --> no dependent type defined
template<bool B, class T = void>
  struct enable_if {};

// ``true case''  --> dependent type is defined
template<class T>
  struct enable_if<true, T> {typedef T type;};
```

Picking Numerical Solvers

Assume we have a set of numerical problems, say, `ProblemA`, `ProblemB`, and maybe others, and also a set of solvers for such problems, `Solver1`, `Solver2`, and potentially some others. We decide to manage the different combinations using a common interface:

```
template<typename Problem,
         typename Solver = Problem::DefaultSolver>
  void solve(...)
  {
    // instantiate Problem and configure it
    // instantiate appropriate Solver
    // apply Solver to Problem
    // postprocessing, etc.
  }
```

This works fine as long as every problem class defines an appropriate solver default.

There are two points to consider:

- The default solver is only a suggestion and another one may be chosen by the user. This includes *solvers that compile fine but are maybe not numerically stable for the given problem*!
- Maybe we want to eliminate the `Solver` template parameter altogether, instead *automatically choosing a suitable solver from our collection* for any problem that is passed to the `solve` function.

For this to work, we have let our different `solve` variants know about certain "properties" of the problem classes, and maybe also of the solver classes. These can then be used to mask those combinations we don't want to be used via SFINAE.

Solving Linear Systems

To discuss a smaller application in more detail and show how SFINAE is actually used, let us assume we have several matrix classes available, and some of them provide a method named `multiplyWithInverse` that solves

$$Ax = b$$

in a very specialized and efficient way[22], and others don't. We want to *make use of this functionality when it is available*, of course, i.e., *we have to check whether the method exists.*

We need the following ingredients:

- A traits class template that checks for the existance of the above method
- The aforementioned `enable_if` to potentially define a type, depending on the "return value" of the traits class
- Different `solve` functions, picking the right one with SFINAE

Traits class for existance of method, adapted from Jean Guegant's example[23]:

```
template <class T1, class T2>
  struct hasMultiplyWithInverse
  {
    // Truth values at compile time
    typedef struct{char a; char b;} yes; // size 2
    typedef struct{char a;}         no;  // size 1

    // Helper template declaration
    template <typename U, U u> struct isMethod;

    // Picked if helper template declares valid type, i.e. if signature matches:
    // - const C1 as implicit first argument to method [(C1::*) ... const]
    // - non-const reference to C2 as second argument, and void as return type
    template <typename C1, typename C2>
      static yes test(isMethod<void (C1::*)(C2&) const, &C1::multiplyWithInverse>*) {}

    // Catch-all default (templatized C-style variadic function)
    template <typename,typename> static no test(...) {}

    // Export truth value (as enum, alternatively as static const bool)
    // Trick: sizeof works without actually calling the function
    enum {value = (sizeof(test<T1,T2>(0)) == sizeof(yes))};
  };
```

Two versions of the `solve` function, one of them being selected by SFINAE:

[22] e.g., some precomputed matrix decomposition, a specialized spectral method, etc.
[23] https://jguegant.github.io/blogs/tech/sfinae-introduction.html

```
template<typename M, typename V>
  typename std::enable_if<hasMultiplyWithInverse<M,V>::value>::type
  solve(const M& m, V& v)
  {
    // implement specialized version here, can use multiplyWithInverse
  }

template<typename M, typename V>
  typename std::enable_if<!hasMultiplyWithInverse<M,V>::value>::type
  solve(const M& m, V& v)
  {
    // implement general version here, has to avoid multiplyWithInverse
  }
```

Standard placements of `enable_if`:

- Additional template parameter, hidden by assigning default value
- Additional function argument, hidden by assigning default value
- Return type of function (chosen above)

SFINAE (cont.)

Central steps to make this form of SFINAE work:

- Use template specialization to determine at compile time whether some type has a certain method or not
- Use this inside `enable_if` to trigger substitution failures on purpose
- Guide compiler to select correct specialization (i.e., remove ambiguity)

The SFINAE code presented above is valid C++98/03. C++11 and later standards introduced several features that make SFINAE significantly simpler, or help avoid it altogether:

- → *Constant expressions*, e.g., to avoid the `sizeof` trick/hack
- Predefined → *type traits*, to simplify writing conditionals
- Mapping between values and their types (→ *decltype/declval*)

A simplified version based on C++11, extracting type information using *decltype*:

```
template<typename T1, typename T2>
  struct hasMultiplyWithInverse
  {
    template<typename T, typename U = void> // (1)
      struct Helper
      {enum {value = false};};

    template<typename T>                    // (2)
```

```
      struct Helper<T,decltype(&T::multiplyWithInverse)>
      {enum {value = true};};

    // matches (2) if function exists, else matches (1)
    enum {value = Helper<T1,void (T1::*)(T2&) const>::value};
  };
```

By the way, `T1::*` is a so-called *pointer to member*, which are, e.g., the types of pointer stored in the vtables we discussed.

Example from DUNE

The product of two functions on a numerical grid can be defined if

- both have the same dimensionality (i.e., *scalar product of vector fields*)
- or one of them is a scalar function (i.e., *simple scaling*)

Base template, handling the first case (also an example of CRTP):

```
//! Product of two GridFunctions
template<typename GF1, typename GF2, class = void>
class ProductGridFunctionAdapter :
  public GridFunctionBase<
    GridFunctionTraits<
      typename GF1::Traits::GridViewType,
      typename GF1::Traits::RangeFieldType, 1,
      FieldVector<typename GF1::Traits::RangeFieldType, 1> >,
    ProductGridFunctionAdapter<GF1,GF2> >
{
  static_assert(unsigned(GF1::Traits::dimRange) ==
                unsigned(GF2::Traits::dimRange),
                "ProductGridFunctionAdapter: Operands must have "
                "matching range dimensions, or one operand must be "
                "scalar-valued.");
  ...
};
```

Specializations for the two different semi-scalar cases, using SFINAE:

```
//! Product of two GridFunctions
template<typename GF1, typename GF2>
class ProductGridFunctionAdapter<GF1, GF2,
  typename std::enable_if<
    GF1::Traits::dimRange == 1 && GF2::Traits::dimRange != 1
    >::type> :
  public GridFunctionBase<typename GF2::Traits, ProductGridFunctionAdapter<GF1,GF2> >
{
  ...
};

//! Product of two GridFunctions
```

```
template<typename GF1, typename GF2>
class ProductGridFunctionAdapter<GF1, GF2,
  typename std::enable_if<
    GF1::Traits::dimRange != 1 && GF2::Traits::dimRange == 1
    >::type> :
  public ProductGridFunctionAdapter<GF2, GF1>
{
public:
  ProductGridFunctionAdapter(GF1& gf1, GF2& gf2)
    : ProductGridFunctionAdapter<GF2, GF1>(gf2, gf1)
  { }
};
```

5. C++11 Features

C++11 was a major update that fundamentally *changed the "look and feel" of C++ programs*. Several minor improvements of C++11 have been directly incorporated into the previous sections, e.g., *rvalue references*, *defaulted/deleted methods*, *delegating constructors*, etc. We will now discuss further topics in greater detail:

- automatic type deduction
- compile-time evaluation
- move semantics
- smart pointers
- lambda expressions (closures)
- variadic templates
- threads and concurrency
- assorted new features and libraries

5.1. Automatic Type Deduction

Quite often, the type of a variable is can be inferred from the surrounding code, e.g., from:

- the type of literals
- the return type of functions

C++11 introduces the keyword `auto`, which instructs the compiler to *automatically deduce the type from context*:

```
auto i = 27;    // int literal -> i is int
auto j = 5u;    // unsigned literal -> j is unsigned
auto x = 5.4;   // x is double
auto y = 2.9f;  // y is float
auto z = foo(); // type of z is return type of foo()
```

Qualifiers may be added if needed, e.g., `const auto&` to refer to constant references instead.

Benefits of automatic type deduction:

- More concise code, very similar to code from weakly-typed languages
- No need for complicated lines to extract type information
- No need for tricks when the required types vary between template instantiations

```
// very verbose piece of code
// only works for types T with these exact dependent types
typename T::grid_type::element_type::dof_type dof = ...
// ==> auto is more concise and more general
```

Drawbacks of automatic type deduction:

- May have to look up function signatures, because resulting type may depend on some far away function definition (may make code harder to read)
- Doesn't work if another type is desired (type promotions and automatic conversions)

Trailing Return Type

C++11 also introduces trailing return types, using the `auto` keyword:

```
// deferred definition of return type
auto multiply(const Matrix& m, const Vector& v) -> Vector
{ ... }
```

This is particularly useful when the return type should be automatically deduced from the arguments:

```
// return type is whatever sum of types X and Y produces
auto foo(X x, Y y) -> decltype(x + y)
{ ... }
```

Here `decltype` produces the type that `auto` would deduce for its argument.

Trailing return types become mostly irrelevant in C++14 and above, because these standards can deduce the return type from return statements.

Example from DUNE

Trailing return types remain useful for *SFINAE on trailing return types*: the existance or nonexistance of a certain operation is tested using the function arguments. This is a direct precursor of concepts as they are introduced by C++20.

```
  template<class A, class B>
  auto dot(const A & a, const B & b)
      -> typename std::enable_if<!IsVector<A>::value
          && !std::is_same<typename FieldTraits<A>::field_type,
          typename FieldTraits<A>::real_type> ::value, decltype(conj(a)*b)>::type
  {return conj(a)*b;}

  // same for a*b instead of conj(a)*b

template<typename A, typename B>
  auto dot(const A & a, const B & b)
      -> typename std::enable_if<IsVector<A>::value, decltype(a.dot(b))>::type
  {return a.dot(b);}

  template<class A, class B>
  auto dotT(const A & a, const B & b) -> decltype(a*b)
  {return a*b;}
```

5.2. Compile-Time Evaluation

We have already discussed template metaprogramming as a way to perform computations at compile-time. C++11 introduced *constant expressions* using the keyword `constexpr`, making such computations significantly simpler.

In C++11 these computations are rather restricted, e.g., `constexpr` functions must

- not call non-`constexpr` functions
- consist of a single `return` line

Later standards lift some of the restrictions, e.g., the latter one above.

```
// template metaprogramming factorial
template<int N>
  struct Factorial
  {
    enum {value = Factorial<N-1>::value * N};
  };

// base case to break infinite recursion
template<>
  struct Factorial<0>
  {
    enum {value = 1};
  };

  // equivalent constexpr function
  constexpr int factorial(int n)
  {
    return (n > 0) ? (factorial(n-1) * n) : 1;
  }
```

5.3. Move Semantics

Before C++11, C++ knew essentially two types of semantics for entities:

value semantics: assignment creates a new, independent entity (behavior for normal variables)

reference semantics: assignment creates an alias (behavior for references, and pointers through indirection)

Programmers can choose between these two types of semantics for their own classes: the behavior is defined by the copy constructor and assignment operator producing

- a deep copy (value semantics)
- or a shallow copy (reference semantics)

C++11 introduced a third type of semantics, *move semantics*. Here, assignment creates neither a copy nor an alias, but moves the resources from one object to the other.

Move semantics are based on rvalue references: while a normal, classical reference `T&` is an alias for a separate, persistent entity, *an rvalue reference* `T&&` *refers to, e.g., temporary values that don't have to be kept around for a later point in time.* More precise, but maybe also more confusing, categorizations (lvalue, prvalue, xvalue, glvalue, and rvalue) are available online[24].

The contents of such rvalue references may therefore be used to initialize other variables by *simply reusing the memory, eliminating the need for explicit copies* and associated memory allocations.

The function `std::move()` can be used to create rvalue references to variables, thereby explicitly marking them as movable. After such an explicit move, the original variable is typically in a valid but unspecified state.

Move Constructors and Assignment

User-defined data types need *move constructors and assignment operators* to make use of this efficient form of transfer. They are auto-generated as long as all attributes are movable and the user provided neither copy nor move constructors / assignment operators.

```
struct Foo
{
  int i;

  Foo(Foo&& other)
  : i(std::move(other.i))
```

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

```
{}

Foo& operator=(Foo&& other)
{
  i = std::move(other.i);
  return *this;
}
```

- Move constructors / assignment operators are quite similar to copy constructors / assignment operators.
- The members of rvalue references aren't rvalues themselves, therefore they have to be moved explicitly.
- The methods should leave the donor in an arbitrary valid state (e.g., assign `nullptr` to any pointer members).

Forwarding References

There are two locations where the two ampersands `&&` don't actually denote an rvalue reference, but rather a *forwarding reference, or universal reference*:

- the construct `auto&&` in → *range-based for loops*
- the argument `T&&` in a function template parameterized by `T`

In these special situations, what looks like an rvalue reference is actually something that *can bind to both lvalues* and *rvalues*. These special references serve the following purposes:

- Accept both lvalue references to container contents and rvalue references to values that are generated on-the-fly in loop iterations
- *Perfect forwarding*: forward both lvalue and rvalue arguments to other functions without having to implement a long list of template specializations (which would cover cases / combinations)

5.4. Smart Pointers

We have seen that manual memory management can lead to bugs that are quite severe, and that in C++ the standard solution to this problem is RAII. C++11 introduced *smart pointers*, replacements for raw pointers that offer automatic memory management based on RAII.

This is an application of the *one responsibility principle*: introducing wrapper classes means that the user-defined class itself doesn't have to concern itself with memory management.

There are three different types of smart pointer:

- `unique_ptr` (exclusive ownership)
- `shared_ptr` (shared ownership, reference counting)
- `weak_ptr` (non-owning handle)

Unique Pointers

The class `unique_ptr` (for *unique pointer*) provides memory management with *exclusive ownership*: it manages an internal raw pointer and prevents any copies of the stored address, which means that the allocated memory will be freed exactly once.

```
// create a unique pointer
std::unique_ptr<int> p(new int);

// * and -> work as expected
*p = 8;

// unique pointers can't be copied, only moved
auto p2 = std::move(p);
// p is now empty (i.e., invalid)
```

Unique pointers have the following functionality:

- Conversion to `bool` to check for validity (replaces `nullptr` comparison)
- A method `get()` for access to the raw pointer (e.g., for passing the stored object to legacy code)
- Methods for releasing the stored raw pointer, or replacing it with another one

There is also a special `unique_ptr` version for arrays that doesn't have dereference operators and instead provides an `operator[]` that forwards to the array. This can be used to manage the lifetime of C-style arrays with a drop-in replacement, while still being able to hand the raw array to legacy code.

Shared Pointers

While unique pointers have exclusive ownership, the class `shared_ptr` (for *shared pointer*) provide memory management with *shared ownership*, as the name implies. Shared pointers can be used to, e.g., provide pointers to several different objects to share some common central resource or facility. *Reference counting* guarantees that the managed object stays alive while there is at least one shared pointer storing its address.

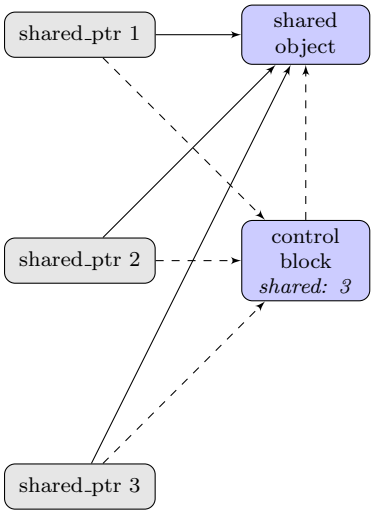
```
std::shared_ptr<int> p(new int);
*p = 5;
auto p2 = p;
*p2 = 7; // also changes contents of p

// allocated int stays around as long as either
// p or p2 point to it (or any other copy in
// some function or object)
```

Shared pointer provide the same methods as unique pointers, and they have an additional `use_count()` method that reports the total number of pointers sharing the resource. There is also again an `operator[]` for the array case, but only from C++17 onwards.

Internally, shared pointers store two raw pointers:

- a raw pointer to the stored object, so that dereferencing has basically the same cost as for raw pointers
- a raw pointer to a *manager object (control block)*, which is responsible for bookkeeping tasks (reference counting)

The manager object in turn also stores a pointer to the managed object, because it has to manage its lifetime (and grants access to weak pointers, as we will see).

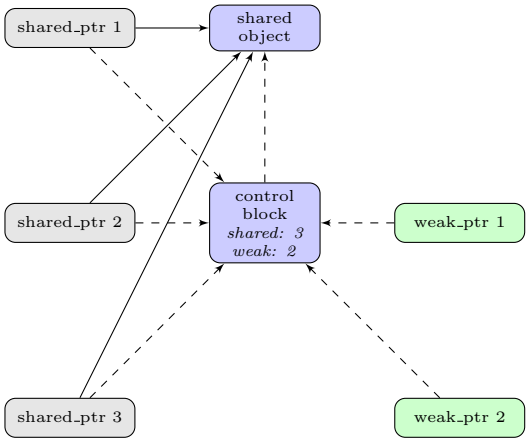


- Solid arrows represent pointers for direct access to the shared object
- Dashed arrows represent pointers used in reference counting and management
- The shared count is incremented when a shared pointer constructor is called
- It is decremented when a shared pointer destructor is called
- The shared object can be destroyed once the shared count reaches zero

Weak Pointers

The third kind of smart pointer is called `weak_ptr`, and strictly speaking these *weak pointers* aren't pointers at all: they are non-owning observers of shared resources, and can't be dereferenced. Instead, if the shared object is still valid, i.e., there is at least one `shared_ptr` to it, a `lock()` can be called to gain temporary access through a shared pointer.

```
std::shared_ptr<int> p(new int);
std::weak_ptr<int> w = p; // link w to p

// w is not expired, since p exists
if (! w.expired())
{
  // create shared pointer
  auto p2 = w.lock();
  // resource is kept alive through p2
  ...
}
```



- The control block maintains a second counter, the weak count, which keeps track of the number of weak pointers
- Locking a weak pointer increases the shared count, because the shared pointer constructor is called
- The method `expired()` returns `true` when the shared count has reached zero, and `false` otherwise
- The shared object is destroyed once the share count reaches zero, while the control block is only destroyed once *both* counts reach zero
- This allows weak pointers to still query the status after all shared pointers are gone

Smart Pointer Creation Functions

There is also a creation function `make_shared`, and from C++14 onward a companion function `make_unique`. These functions forward their arguments to the constructor of the stored object, thereby eliminating the last bit of raw pointer handling by hiding the `new` call.

```
auto p = std::make_shared<double>(7.);
```

In contrast to the `shared_ptr` constructor call, *this function allocates space for the stored object and the manager object in one go.* This means the two ways of constructing shared pointers are not fully equivalent:

- The function `make_shared` uses less allocations, and may therefore be faster than the constructor call.
- For `make_shared`, the space occupied by the stored object is linked to that of the manager object due to the joint memory allocation.

In other words, the destructor of the managed object is called when the last shared pointer vanishes, but the memory becomes available again when all shared *and* weak pointers are gone.

Smart Pointer Use Cases

Each type of pointer is for a specific set of use cases:

`unique_ptr` Pass around object that doesn't have to be available at different locations simultaneously, capture raw pointers from legacy code that have to be managed.

`shared_ptr` Pass around object that has to be available at different locations, share object ownership and manage lifetime collaboratively.

`weak_ptr` Non-owning, observe shared object and its lifetime, may lock if temporary access is required.

raw pointers For interfaces with legacy code, implementation details of classes that are safely encapsulated, and sometimes used in modern code to mark pointers as explicitly non-owned.

If explicitly managed raw pointers are completely avoided, then each type of pointer has a clear purpose, and for each managed object there are two sets of pointers:

Owning Pointers Either a single `unique_ptr` or one or more `shared_ptr`, which own the object and manage its lifetime.

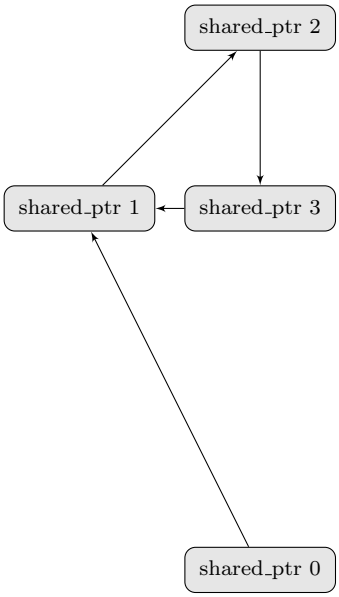
Non-owning Pointers An arbitrary number of `weak_ptr` and raw pointers, which don't own the object. The former can check object validity, while it has to be guaranteed by the calling code for the latter. This also means that raw pointers should never be returned or stored, only used and passed along to other functions.

Pitfalls When Using Shared Pointers

With modern compilers, using shared pointers has surprisingly little overhead: dereferencing them has basically the same cost as directly using pointers. The creation, destruction, and storage of shared pointers, however, creates a certain overhead:

- Each set of such pointers requires storage for two raw pointers per shared pointer, and additionally for a central control block
- Creating and destroying shared / weak pointers is more expensive than using raw pointers, due to reference counting

However, these overheads are usually not significant, and a small price for the guaranteed proper memory management.



While shared pointers are usually quite safe, subtle bugs may be introduced when loops are created:

- Such loops occur in ring buffers, doubly linked lists, and similar constructs
- If all external shared pointers go out of scope, the loop becomes unreachable
- But the control blocks keep the loop alive, because there is always at least one remaining shared pointer per manager object
- One can replace one or all of the internal shared pointers by weak pointers to prevent this silent memory leak

Another pitfall occurs when an object tries to return a shared pointer to itself, e.g., during the construction of the loops we just discussed. One might be tempted to simply return a shared pointer wrapping the `this` pointer:

```
return std::shared_ptr<Foo>(this);
```

However, this would create a separate set of shared pointers, and the two distinct control blocks would result in calling `delete` twice. There is actually an official workaround for this: derive from a special class, and then call a special method.

```
struct Foo : public std::enable_shared_from_this<Foo> // CRTP
{
  auto get_ptr() {return shared_from_this();}
}
```

The class `enable_shared_from_this` works like this:

- The shared pointer constructor is specialized for classes that inherit from this class, and places a `weak_ptr` in the base class object.
- The method `shared_from_this()` accesses this weak pointer, locks it, and returns the resulting shared pointer.
- The original shared pointer and the new one use the same manager object, therefore the problem is avoided.

Before C++17, calling `shared_from_this()` on an object that isn't actually stored in a shared pointer lead to silent undefined behavior, but since C++17 this case thankfully throws an appropriate exception.

5.5. Lambda Expressions (Closures)

We have already discussed functors: objects that provide an `operator()` method, which can therefore be called like functions. The main benefit of functors over simple function pointers is their flexibility, based on their constructors and internal attributes one may:

- provide some form of parameterization (constructor arguments)
- have internal state / function memory (data members)
- grant access to local variables (by passing them to the constructor)

C++11 introduced a framework to generate such functors automatically from short descriptions called *lambda expressions*. These expressions are treated as if the equivalent functor class had already been provided, and are therefore a convenient way to provide *ad-hoc function definitions (closures)*.

A lambda expression consists of the following components:

- a *capture list* in brackets, specifying which local variables (if any) should be made available within the expression
- a list of function arguments in parentheses, as usual
- a trailing return type, but without `auto` in front
- a function body in braces, as usual

```
int a = 4;
int b = 5;

// store function object in local variable
auto f = [a,&b] (int x) -> int {b += a*x; return a-x;};
```

```
std::cout << f(3) << std::endl; // 4 - 3 = 1
std::cout << b << std::endl;    // 5 + 4*3 = 17
```

The capture list may:

- be empty, thereby defining an anonymous function that is not a closure (i.e., nothing is captured[25])
- mention the variables that should be captured, with a `&` in front of those that should be captured by reference instead of value
- start with a single `=`, followed by zero or more captures by reference
- start with a single `&`, followed by zero or more captures by value

In the last two cases, any variables that is used in the expression is automatically captured by-value (for `=`) resp. by-reference (for `&`).

```
int a = 1, b = 2, c = 3;
// captures b by-reference, a and c by-value (if actually used)
auto f = [=,&b] () -> void {...};
// captures a by-value, b and c by-reference (if actually used)
auto g = [&,a]  () -> void {...};
```

Behind the scenes, each lambda expression defines a *local functor class, with an automatically generated, unknown name* that is guaranteed to not clash with any names that are actually present in the code.

- Such *local classes*, i.e., classes defined within functions and other classes, are actually a general concept in C++. Typically, they are only used to provide private helper classes as implementation detail of some other class.
- Local classes can access the static local variables of the surrounding scope, therefore variables marked `constexpr` and `static` are always available.
- Anything else that is needed inside the expression has to be captured explicitly or with a capture default (`=` or `&`).
- The constructor is auto-generated from the intersection of the capture list and the set of variables that are needed for the lambda body, while the body itself and the arguments are used for the definition of `operator()`.

[25] Technically, compile-time constant expressions are available, anyway.

Example from DUNE

Two vector norms, implemented as STL algorithms over some internal container, using lambda expressions as operations / predicates:

```
typename Dune::template FieldTraits<E>::real_type one_norm() const
  {
    typedef typename Dune::template FieldTraits<E>::real_type Real;
    return std::accumulate(_container->begin(),_container->end(),Real(0),
                           [](const auto& n, const auto& e) {
                             using std::abs;
                             return n + abs(e);
                           });
  }

  typename Dune::template FieldTraits<E>::real_type infinity_norm() const
  {
    if (_container->size() == 0)
      return 0;
    using std::abs;
    typedef typename Dune::template FieldTraits<E>::real_type Real;
    return abs(*std::max_element(_container->begin(),_container->end(),
                                 [](const auto& a, const auto& b) {
                                   using std::abs;
                                   return abs(a) < abs(b);
                                 }));
  }
```

5.6. Variadic Templates

C++11 introduced *template parameter packs*, which are placeholders for an arbitrary number of template parameters (type parameters, non-type parameters, or template template parameters, but not mixed). In the case of type template parameters, these may be accompanied by a *function parameter pack* containing the corresponding values.

Parameter packs may be used to create templates that accept an arbitrary number of template parameters, so-called *variadic templates*:

```
// arbitrary number of int parameters
template<int... args>
  class Foo { ... };

// arbitrary number of type parameters, plus arguments
template<typename... Ts>
  void bar (Ts... args)
  { ... };
```

We have already discussed such a variadic template, namely `std::tuple`.

The code of a variadic function template has to be written in a way that it treats a variable number of parameters, and the standard way of achieving this is via specifying the first parameter (the "head") separately, and then using recursion.

The remaining list of parameters / arguments (the "tail") can be used within the function body: one replaces

- the parameter list `typename... Ts` with `Ts...` and
- the argument list `Ts... args` with `args...`.

The ellipsis `...` may also be placed after any expression containing these packs, instead of directly behind them.

Every expression containing at least one such pack is subject to *pack expansion*: at instantiation, the expression is replaced by a comma-separated list of copies of itself, with the pack replaced by each one of its constituents in turn.

An implementation of Horner's method, efficiently evaluating a given polynomial:

$$a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot \cdots))$$

```
// this is an overload, not a specialization!
template<typename T>
  double eval_poly(double x, T head)
  {
    return head;
  }

// general case: used when the pack is non-empty
template<typename T, typename... Ts>
  double eval_poly(double x, T head, Ts... tail)
  {
    // pack expansion: insert remaining coefficients
    return head + x * eval_poly(x,tail...);
  }

  // evaluates f(x) = 3 + 5x + 2x^2
  eval_poly(x,3,5,2);
```

For the sake of demonstrating a bit of the power of pack expansion, we may choose to multiply each of the remaining coefficients with x instead. This makes the algorithm significantly less efficient, of course: it implements the naive way of evaluating polynomials, after all.

```
template<typename T>
  double eval_poly(double x, T head)
  {
    return head;
  }

template<typename T, typename... Ts>
  double eval_poly(double x, T head, Ts... tail)
  {
    // pack expansion: remaining coefficients times x
    return head + eval_poly(x, x * tail...);
  }
```

One can even expand several packs in the same expression, leading to quite powerful constructs:

```
template<typename... Ts1>
  struct Left
  {
    template<typename... Ts2>
      struct Right
      {
        using Zip = std::tuple<std::pair<Ts1,Ts2>...>;
      };
  };

  // defines the pairing [(A,X),(B,Y),(C,Z)]
  using Zip = Left<A,B,C>::Right<X,Y,Z>::Zip;
```

5.7. Concurrency with Threads

Concurrency and Parallel Computing

Moore's Law states that the number of transistors in integrated circuits doubles roughly every two years (i.e., *exponential growth*). This typically translates to a similar increase in computational power and available memory. However, there are clear physical limits to this growth (atom size, speed of light).

The computational power of single CPU cores has been stagnant for some years, and instead *multi-core systems* have become dominant: most modern computers have CPUs with multiple

cores, not just compute clusters, but also personal laptops and desktops, and even smart phones.

As a consequence, *concurrency and parallel computing* are no longer just means to accelerate computations: nowadays, a solid understanding of parallel programming techniques is important to make full use of the resources a computer provides.

There is a wide variety of compute architectures available today. A simplified categorization might be:

- Multi-core CPUs with associated main memory on a single machine, every core has full access to the system memory (*uniform memory architecture, UMA*), concurrent programming using *threads* (separate execution flows within one program)
- Clusters, i.e., tightly interconnected computers as a unit, each with its own CPU and memory (*non-uniform memory architecture, NUMA*), parallel programming based on *message passing* to exchange memory contents and computation results

While message passing requires external liberaries, concurrency using threads is provided natively by C++11 and above.

Concurrency with Threads

C++ had thread suppport before C++11, of course, but the availability and characteristics of these threads where dependent on the architecture and operating system, e.g., *POSIX threads (pthreads)* under UNIX and Linux. C++11 provides a unified interface to thread creation and management, thereby increasing interoperability and platform independence of the resulting code.

Note that there is typically still the need to choose an appropriate thread backend, e.g., using the `-pthread` compiler flag to use POSIX threads.

There are two different ways of writing such concurrent programs:

- *low-level code based on threads, using mutexes and locks* to manually manage thread interactions and data exchange
- *higher-level code based on tasks, using futures and promises* to abstract away most of the interdependencies

We have discussed smart pointers as handles for memory. A `std::thread` object is also a handle, but one that represents an executing subprogram.

- The first argument of its constructor is a function pointer, functor, lambda expression or similar. This is the function the thread will execute, and may be interpreted as the "main function" of the subprogram.

- The remaining arguments are passed to this function using perfect forwarding. There may be an arbitrary number of them, because the constructor is a variadic template.
- Threads can't be copied, only moved, because there may be only one representation of the running subprogram (compare `unique_ptr`).
- Calling the `detach()` method releases the thread, letting it run indendently from the main program, while calling `join()` halts the main program until the thread is done and ready to be assimilated.
- The destructor aborts the program via `terminate()` if neither `join()` nor `detach()` have been called beforehand.

Thread-based programs have a *special thread-local namespace* called `std::this_thread`, which provides a number of functions for thread control:

- The function `get_id()` returns a number that identifies the thread itself. The thread handle `std::thread` has a method `get_id` with the same information, but this is inaccessible from within the running thread, of course.
- The function `yield()` signals the operating system and suggests that other threads should be allowed to run, e.g., because the current thread would be mostly idle anyway.
- The functions `sleep_for` and `sleep_until` can be used to let the thread wait for some time without resorting to resource-intensive busy-waiting.

The two sleep functions expect a duration resp. time point from the C++11 → *`chrono` date and time library*.

```
int val = 0;

// lambda as first argument, no explicit function arguments
// val captured by-reference through the lambda
std::thread t([&](){
    for (int i = 0; i < 10; i++)
    {
      std::cout << ++val << std::endl;
      // blocks thread execution for given duration
      std::this_thread::sleep_for(std::chrono::seconds(1));
    }
  });

// could do something else here in the meantime

// wait for thread to finish execution
t.join();
```

Mutexes and Locks

If two or more threads share some resources, *race conditions* may occur: the result of read and write operations on shared memory may depend on their exact order of execution. A well-known example is a *lost update*:

- Two threads read a value from memory, increment it, and store the result.
- If these two sets of operations don't overlap, then everything is fine: the value is incremented twice.
- If there is an overlap, then only one of the increments is stored, and the program doesn't produce the expected results.

This form of race condition can be prevented through *mutual exclusion (mutexes)*: only one thread at a time may enter the critical section where the shared resource is accessed, thereby eliminating any potential for such overlaps.

Calling the `lock()` method on a mutex stops execution if another thread has currently locked the same mutex, while `unlock()` releases the lock, allowing the next thread to enter the critical section:

```
void count(int id, int steps, int& val, std::mutex& mut)
{
  for (int i = 0; i < steps; i++)
  {
    mut.lock();   // ensure exclusive access to val
    std::cout << id << ": " << ++val << std::endl;
    mut.unlock(); // allow other threads to access

    std::this_thread::sleep_for(std::chrono::seconds(1));
  }
}

int val = 0; std::mutex mut;
// pass val and mut as references using perfect forwarding
std::thread t1(count, 0, 10, std::ref(val), std::ref(mut));
std::thread t2(count, 1, 10, std::ref(val), std::ref(mut));
t1.join(); t2.join();
```

Mutexes aren't exception-safe, and shouldn't be used directly. Instead, they are usually wrapped in a `lock_guard` object, which locks the mutex in its constructor and unlocks it in its destructor, even if an exception is triggered:

```
void count(int id, int steps, int& val, std::mutex& mut)
{
  for (int i = 0; i < steps; i++)
  {
    // additional scope to keep lock_guard lifetime short
    // (should not stay around for the sleep period)
    {
      // constructor locks the mutex
      std::lock_guard<std::mutex> lock(mut);
```

```
      std::cout << id << ": " << ++val << std::endl;
    } // destructor unlocks the mutex

    std::this_thread::sleep_for(std::chrono::seconds(1));
  }
}
```

In short, lock guards use RAII to ensure exception-safe mutual exclusion.

There is a number of related methods and classes:

- Mutexes have a non-blocking method `trylock()`: the thread doesn't wait, instead it can do other work and maybe retry later.
- The class `recursive_mutex` can lock multiple times, e.g., when the same resource is needed simultaneously in a number of related functions / methods.
- There are timed variants `timed_mutex` and `recursive_timed_mutex`, which are able to *wait for a certain time instead of indefinitely.*
- The class `unique_lock` is a more flexible replacement for `lock_guard`.
- *Condition variables* (`condition_variable`) can be used to wait on a lock, until another thread wakes one or all waiting threads using `notify_one()` or `notify_all()`. This can be used to *create synchronization barriers and organize a structured transfer of data.*

Atomics

An alternative to mutexes and locks is the use of *atomics*. Data types can be made atomic by wrapping them in a `std::atomic<T>`, which provides special uninterruptible methods for the usual operations. This solves the lost update problem, because read—modify—write cycles are carried out as single operations.

Atomics can utilize *highly efficient special CPU instructions* on modern architectures, or, if these aren't available, fall back to using internal locks. In the latter case they are basically wrappers that hide the explicit use of mutual exclusion. There are also special *memory order models* that can guarantee a certain order of memory operations across different threads. These are quite powerful, but using them is not as straight-forward as the explicit use of flags and condition variables.

Examples of atomic operations for `std::atomic<T>`:

- Methods `load` and `store` for reading and writing. These have the conversion operator `operator T` and assignment `operator=` as aliases.
- Methods `compare_exchange_weak` and `compare_exchange_strong`, which replace an atomic value if it is safe to do so.

- If `T` is an arithmetic type, special atomic updates like `fetch_add`, `fetch_sub`, etc., which have the aliases `operator+=`, `operator-=`, `operator++`, and so on.

The methods are named after the special CPU instructions they represent, lending themselves to a more explicit coding style, while the operator aliases make it possible to write code that looks more or less like normal C++ code operating on thread-local variables.

The method `compare_exchange_weak` has the following signature:

```
bool replaced = atom_v.compare_exchange_weak(expected, desired);
```

If `atom_v` contains `expected`, it is replaced by `desired`, else `expected` is replaced by the value stored in `atom_v`. This can be used in a loop, checking the value until the update succeeds:

```
while (!atom_v.compare_exchange_weak(e,d))
{
  // react to fact that atom_v actually contains value in e
  ...
}
```

The method `compare_exchange_strong` works the same, and additionally guarantees that updates are only performed if actually needed, while the weak version may cause spurious updates[26].

Concurrency with Tasks

Many programming tasks allow a certain amount of *concurrency on two different levels*: fine-grained interaction, and the separate handling of subtasks.

Explicit use of threads is often a good choice when:

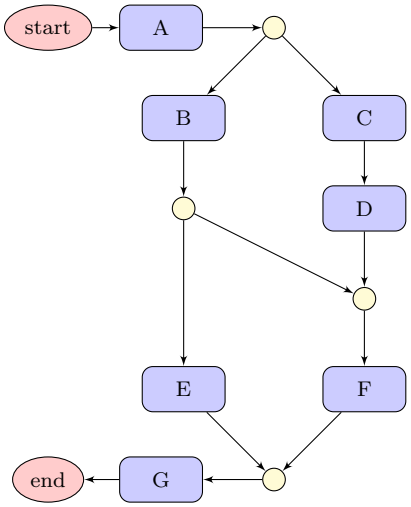
- there is extensive communication and exchange of data between the threads
- the threads operate collaboratively, maybe even executing the same code on different sets of data, exchanging intermediate results

Task-based concurrency is an additional layer of abstraction for cases where:

- the program can be decomposed into more-or-less independent sub-tasks
- each such task has well-defined sets of prerequisites it needs and results it will produce

Of course, threads and tasks may also be combined, resulting in two simultaneous levels of concurrency.

[26] See https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange



- Most scientific programs have setup (A) and output/cleanup (G) phases which have to be executed in sequence.
- Between these two phases, a certain number of tasks, here (B) to (F), have to be performed. Some of these have to be performed in order, but others may allow a certain degree of flexibility and parallel computation.
- Tasks may be algorithms or sub-algorithms, like some matrix factorization, or the solution of some nonlinear system, but also, e.g., background I/O operations.

Promises and Futures

At the heart of task-based programming are *promise—future pairs*: a promise represents the result some task will ultimately produce, while a future represents, well, the future inputs another task is waiting for. As such, promises and futures serve a similar purpose to pipes and data streams: *unidirectional transport of data*.

A `std::promise<T>` object has the following functionality:

- Create an associated future with `get_future`
- Store a result through method `set_value`, signaling task completion
- Store an exception instead using `set_exception`, signaling task failure
- Two methods that set a result resp. an exception at thread exit instead, so that local cleanup can be taken care of

A `std::future<T>` object provides the following:

- Methods `wait`, `wait_for`, and `wait_until`, which block execution and wait for the result, either indefinitely or for a certain time

- A method `valid`, as a non-blocking way of checking for task completion
- A method `get`, which returns the stored value if the future is ready, or throws the stored exception if one was set
- A method `share`, moving the internal shared state to a `shared_future`

If `T` is not a reference type, then `get` returns the stored value as an rvalue reference. Consequently, *the method `get` may be called exactly once* in this case[27].

The class `std::shared_future` can be used to provide the stored value to multiple threads. In constrast to a normal future, its `get` method returns `const T&` and keeps the internal state valid.

A matrix-vector product, handed over to a new thread:

```
void matvec (const Matrix& mat, const Vector& vec,
             std::promise<Vector> promise)
{
  Vector res = mat * vec;
  promise.set_value(std::move(res));
}

std::promise<Vector> promise;
std::future<Vector> future = promise.get_future();
// hand over matrix-vector pair, promise must be moved
std::thread t(matvec,mat,vec,std::move(promise));

// could do something else here

future.wait(); t.join(); // wait, then clean up
Vector res = future.get();
```

Packaged Tasks

A *packaged task* (`std::packaged_task`) is a convenient wrapper around functions and function-like objects, like functors and lambda expressions. Calling the packaged task as if it were a function executes its content concurrently, including low-level tasks like thread creation and destruction. The function return value is provided using a promise—future pair.

[27] This makes it possible to return large objects from tasks using move semantics.

```
// packaged task with trivial job, defined through a lambda
std::packaged_task<int(int,int)>
        task([](int a, int b){return a*b;});
// future object that will contain return value
std::future<int> future = task.get_future();
// execute task, i.e., run function
task(3,4);

// could do something else here

future.wait();          // wait for task to finish
int res = future.get(); // extract result
```

Asynchronous Execution

Packaged tasks may manage their own threads, or have a thread explicitly assigned to them, but they still have to be started manually. There is an additional construct that is even more abstract and high-level: `std::async`, a function providing *asynchronous execution*. The first argument of this function is an *execution policy*:

`std::launch::async`**:** launch a new thread that executes the task, similar to `packaged_task`

`std::launch::deferred`**:** execute the task on the current thread, but defer it until the result is actually needed for the first time (lazy evaluation)

default (if omitted): bitwise OR of `async` and `deferred`, which enables the `async` call to choose between the two options, i.e., start a new thread if that is convenient at some point in time, or wait until the result is needed

The `std::async` function returns a future that will contain the result:

```
std::future<Foo> future = std::async(expensiveFunction,args);

// do something else here

// wait for thread if one was spawned (asynchronous)
// or compute result here if none was created (deferred)
Foo result = future.get();
```

Futures from `std::async` are special: their destructor blocks and forces the contained function to run even if `get` was never called. This means that functions for side effects (e.g.,

background I/O) are guaranteed to actually run, and to be finished when the future is destroyed, with the `async` trying to run the function at a suitable moment[28].

5.8. Assorted New Features and Libraries

Types of Initialization

Before C++11, C++ provided several different *initialization mechanisms* with somewhat inconsistent notation:

Value: written `T();`, default constructor for nameless objects, componentwise value init. for array types, zero-initialization otherwise

Direct: written `T t(args...);`, constructor call for classes, conversion operator or narrowing conversion otherwise

Copy: written `T t = other;` or `T t(other);`, copy constructor, etc.

Aggregate: written `T t = {args...};`, direct assignment of array entries, or of struct components if struct is simple enough[29]

Reference: written `T& ref = t;`, creation of reference

Importantly, the default constructor call is `T t;`, not `T t();`, and using this expression for built-in types leaves them uninitialized!

These forms of initialization remain valid in C++11 and above, but there is also a new mechanism that has a more consistent notation and is more flexible: *initialization using braced lists* `T t{args...};`. The braces can contain zero, one or more than one argument.

Zero arguments:

Value initialization, using the default constructor for class types. Consistent notation for all types, and *fundamental types are zero-initialized.*

One Argument, same or derived type:

Direct initialization, possibly with slicing. *No narrowing conversions allowed for fundamental types* except for literals that are representable, i.e., no silent conversion from `double` to `int` or similar.

A different argument, or more than one:

Aggregate initialization for arrays and simple structs, else forwarded to a constructor taking a `std::initializer_list<T2>` if available, e.g., to mimic aggregate initialization, else forwarded to a matching constructor. *No narrowing conversions allowed for arguments.*

[28] A void future `std::future<void>` can be used in this case.
[29] See https://en.cppreference.com/w/cpp/language/aggregate_initialization for exact specification.

```
int a; // usual notation leaves a uninitialized
int b(); // zero-init., doesn't work with classes
int c{}; // zero-init., notation consistent with classes

std::vector<int> v(5,2); // vector [2,2,2,2,2]
std::vector<int> w{5,2}; // vector [5,2] (initializer_list)
```

Range-Based Loops

C++11 introduced *range-based* `for` *loops* as they exist in other high-level languages:

```
std::array<int,5> arr = {1,2,4,8,16};
for (const int& m : arr)
  std::cout << m << std::endl;
```

This is a more concise replacement for, and in fact built upon, the `for` loop using iterators:

```
for (typename std::array<int,5>::const_iterator it = arr.begin();
        it != arr.end(); ++it)
  std::cout << *it << std::endl;
```

Range-based `for` loops can also be used with user-defined classes. One just has to define three things:

- An iterator type, the one that would be used in the old-fashioned loop
- A `begin` method / free function for the start of the range
- An `end` method / free function denoting the end of the range

This means that anything that supports loops based on iterators also works with range-based loops.

One can implement the usual `begin` / `end` container methods, or use the free functions instead, e.g., if the class can't be modified for some reason. The iterator type has to support the expected operations, of course: pre-increment, dereference operator `*`, not-equal-to operator `!=`, and a public destructor, just like for the explicitly iterator-based loop.

The type definition in range-based `for` loops is an ideal candidate for the `auto` keyword, since the type of the elements is usually clear from context. This leads to two standard versions of such loops, one for const and one for non-const access:

```
// const version
for (const auto& element : range)
{ ... }

// non-const version
for (auto&& element : range)
{ ... }
```

As discussed, this `auto&&` is a forwarding reference: it decays to a normal lvalue reference when accessing the contents of some container, but also accepts values that are generated on-the-fly, e.g., through some generator function.

A braced initialization list can be assigned to `std::initializer_list<T>` if and only if all of its entries have the same type (which is then used as `T`), and such an initializer list can serve as a range. In fact, iterating over such a list is how the corresponding constructors are implemented.

This means we can use braced lists not just for initialization, but also for range-based loops:

```
// iterate over some powers of two
for (const auto& i : {1,2,4,8,16})
{ ... }

// iterate over the vowels
for (const auto& c : {'a','e','i','o','u'})
{ ... }
```

Type Aliases and Alias Templates

In C++11 and above, the keyword `using` provides an *alternative to typedefs*, which is more general and can also be used to give names to partial template specializations / instantiations.

```
// type alias instead of classical typedef
using Number = int; // replaces ``typedef int Number''

// partial specialization or instantiation
template<typename T>
  using Vector3D = Vector<T,3>; // defines ``Vector3D<T>''

// alias template
template<typename U, typename V>
  using Matrix = Long::Namespace::SpecialMatrix<U,V>;
```

```
// variadic alias template
// (can also be used to just omit the arguments!)
template<typename... T>
  using Foo = Bar<T...>; // Foo is alias of Bar
```

Of course, simply importing names like in `using std::cout;` or similar is still possible as well.

Initializers and Inheriting Constructors

C++11 simplifies writing classes:

- Initializers may now also be used for normal (i.e., non-`static`) attributes. The assigned value is used if the constructor doesn't explicitly define the member.
- The constructors of base classes may now be imported with a `using` directive, just like one could import normal methods before. This automatically produces forwarding constructors for the derived class.

```
struct Base {Base(int a, double b, bool c);};

struct Derived : public Base
{
  int i = 0; // used if constructor doesn't assign anything

  // defines Derived(int,double,bool) and potentially others
  using Base::Base;
};
```

Scoped Enums

Classic C enums are simply names for integer constants. This means they can be used where

- some simple integer expression or
- another completely unrelated enum

is expected, potentially leading to subtle bugs.

C++11 introduced *scoped enums*, which are type-safe and can only interact with enums of the same type:

```
enum OldColor {RED, BLUE, GREEN}; // old C-style enum
int i = RED; // perfectly fine assignment

enum class Color {red, blue, green}; // scoped enum
Color col = Color::red; // no implicit conversion to int
using Color::green; // local shorthands may be defined
```

Exception-Free Functions

Previous versions of C++ included a facility for specifying what kind of exceptions a function might throw. This was deemed too complicated and error-prone, and therefore C++11 switched to *specifying that a function will never throw exceptions*, used for optimization purposes. There are two versions:

- A simple `noexcept` qualifier, stating that exceptions are never thrown
- A `noexcept` qualifier with Boolean argument, typically using an operator with the same name

```
// never throws an exception
int five() noexcept {return 5;}

// doesn't throw if both addition and scaling of type T don't
template<typename T>
T scale_and_add(const T& t1, double a, const T& t2)
        noexcept(noexcept(a * t1) && noexcept(t1 + t2))
{
  return a * t1 + t2;
}
```

Type Traits

As seen in our discussion of template metaprogramming and SFINAE, it can be quite handy to have certain properties of data types available at compile time.

C++11 and above provide a set of such *type traits* in header `<type_traits>`:

- Type properties (e.g., `is_integral`, `is_floating_point`, `is_class`, `is_const`,...)
- Type relationships (e.g., `is_same`, `is_convertible`)
- Type modifications (e.g., `remove_const`, `remove_pointer`)
- Transformations (e.g., `enable_if`, `conditional`, `common_type`)

- Constants (`integral_constant` , `true_type` , `false_type`)

These have been defined using the same template tricks we have discussed.

For most of these traits, C++17 introduces an equivalent → *variable template*, for example `is_integral_v` , and a mayor use case are → *concepts* as introduced by C++20.

Time Measurements

The ***`chrono`*** *date and time library* is a replacement for the old C-style date and time utilities, with proper type support and automatic unit conversions. The chrono library has its own namespace `std::chrono` , where it provides:

- *A* ***`system_clock`*** *class for wall time*: equivalent to C-style time, usually Unix Time (UTC), can be nonmonotonic due to, e.g., NTP synchronization.
- *A* ***`steady_clock`*** *class for time measurements*: monotonic (time always moves forward), e.g., time since last system reboot.
- A `time_point` class template, representing time points for each of the clocks (template parameter).
- A `duration` class template, representing durations (time point differences).

Time measurements should always use ***`steady_clock`*** *, because the system clock may perform jumps backward in time (e.g., during synchronization) or into the future (e.g., when waking from suspend state).*

Time measurements using the library tend to be a bit verbose, because durations have to be explicitly cast to time units:

```
auto start = std::chrono::steady_clock::now();

// perform some computations, measure elapsed time

auto end = std::chrono::steady_clock::now();
auto ms = std::chrono::duration_cast<std::chrono::milliseconds>
          (end - start).count();
std::cout << "duration in ms: " << ms << std::endl;
std::cout << "duration in s:  " << ms/1000. << std::endl;
```

The cast converts the internal unit of the clock (ticks) to one of the usual units, e.g., milliseconds or seconds. The units provided by C++11 range from nanoseconds to hours, while days, weeks, months and years are introduced by C++20.

Random Number Generation

The legacy C *pseudo-random number generator (PRNG)* `rand()` can be of rather poor quality. Before C++11, this meant that people had to either make do with subpar random numbers, or rely on some external library. C++11 and above provide a set of very well-known PRNGs in header `<random>`, e.g.:

- The Mersenne Twister (Matsumoto and Nishimura)
- "Minimal standard" linear congrual engine (Park et al.)
- RANLUX generator (Lüscher and James)

The generation of random numbers is divided into three distinct steps:

- *Seeding the aforementioned engines*, using true random numbers (entropy) or simply system time if the latter is sufficient
- *Generation of uniformly distributed pseudo-random integers* using the engines
- *Transformation of these values* into discrete or continuous target distributions

```
std::random_device rd;              // entropy source
std::mt19937 gen{rd()};             // seed Mersenne Twister
std::normal_distribution<> dist{}; // standard normal dist

std::vector<double> vals(1000);
for (auto& e : vals)
  e = dist(gen); // draw value using generator
```

Predefined distributions include:

- *uniform distributions* (integer and real)
- *Bernoulli distributions* (Bernoulli, binomial, etc.)
- *Poisson distributions* (Poisson, exponential, Gamma, etc.)
- *normal distributions* (normal, lognormal, chi-squared, etc.)

Custom Literals

C++ provides a large number of predefined literals, e.g.:

integers binary (prefix `0b`), octal (prefix `0`), decimal or hex (prefix `0x`), postfix `u` for unsigned, `l` for long, `ll` for long long

floating-point number followed by optional exponent separated with `e`, postfix `f` for float, `l` for long, double if omitted

```
auto a = 0b100101 // binary encoded integer
auto b = 123u     // unsigned int
auto c = 0xCAFEul // unsigned long, hex encoded

auto x = 3.141f   // single-precision float
auto y = 1e10     // double-precision with exponent
```

All letters in these literals are case insensitive. There are also literals for different types of characters and strings (UTF-8, UTF-16, ...).

Since C++11, *custom literals* can be defined, based on two ingredients:

- A *user-defined suffix*, which has to start with an underscore
- A *literal operator* for this suffix, defining how a given numerical value or character sequence should be interpreted

```
// compute in meters, allow specifications in foot and inch
constexpr long double operator"" _inch (long double x)
{return 0.0254 * x;}

constexpr long double operator"" _ft (long double x)
{return 0.3048 * x;}

// compute in radian, allow specifications in degrees
constexpr long double operator"" _degs (long double x)
{return M_PIl / 180 * x;} // long double pi constant

auto x = 47_inch; auto y = 3_ft; auto z = 90_degs;
```

Raw Literals and Regular Expressions

A bit less relevant for scientific computing, but still quite useful, are the new text manipulation capabilities introduced by C++11. *Raw string literals* can be used to specify arbitrary texts without the need to escape any characters, like quotation marks or newline characters:

```
// delimiter "foo" is arbitrary, needed to allow ")" in text
// can use any symbol instead, e.g., "|", "!", or "@"
auto raw_string = R"foo(This is a
multi-line string
containing "quotation marks"
and \literal\ backslashes)foo";
```

Such raw literals can be particularly useful when writing *regular expressions* in one of the pattern matching grammars provided by the *regex library*[30].

[30] See https://en.cppreference.com/w/cpp/regex.

6. C++14 Features

C++14 was a comparatively minor update, but it is nevertheless brought several improvements to the language, mainly regarding constructs introduced in C++11:

- improvements to constant expressions
- generic lambda expressions
- variable templates
- return type deduction
- assorted other improvements and additions

6.1. Improvements to Constant Expressions

Except for possibly some checks (`static_assert`) and local definitions (`using` / `typedef`), C++11 `constexpr` functions are basically a single return statement.

C++14 lifts many restrictions and consequently makes constant expressions much more powerful. The body of a `constexpr` function may now contain normal C++ code, as long as:

- Every variable has literal type, i.e., it is either built-in or has a matching `constexpr` constructor and destructor.
- Every such variable is immediately initialized.
- All functions that are called are themselves marked `constexpr`.
- The code doesn't contain `try` / `catch` blocks (exception handling).

The second and fourth restriction are lifted by C++20, where `constexpr` exceptions are introduced.

Most importantly, `constexpr` functions may now contain branches and loops:

```
constexpr int factorial(int n)
{
  int out = 1;
  for (int i = 2; i <= n; i++)
    out *= i;
  return out;
}
```

Just like liberal use of the `const` keyword prevents bugs and helps the compiler during optimization, large parts of programs may now be marked `constexpr`. These parts will then be evaluated at compile-time where appropriate.

This means we can write whole subprograms that are run at compile-time, instead of relying on complex template metaprograms for such tasks.

6.2. Generic Lambda Expressions

C++14 introduced *generic lambda expressions*: lambda function arguments may now be declared `auto`, as in:

```
auto add = [] (auto x, auto y) {return x+y;};
```

The anonymous functor introduced by this lambda expression doesn't provide a single function `operator()`, instead it defines an appropriate `operator()` function template:

- *Each occuring* `auto` *introduces an (anonymous) template parameter*, say, `T1`, `T2`, ...
- *These parameters are independent*: there is no mechanism to specify that two types are the same or have some other relationship
- Any time the lambda is called with a certain argument type combination for the first time, the `operator()` function template is instantiated with these types.

6.3. Variable Templates

Before C++14, templates came in two different flavors, function templates introducing a parameterized family of functions, and class templates introducing a parameterized family of classes or structs.

C++14 introduced *variable templates*, i.e., variables that may have several variants existing at the same time, parameterized by some types or values. Variable templates *can have the same template parameters as other templates*, including template parameter packs, but they themselves *can't be used as template template parameters.*

```
// no longer a struct with internal enum or similar
template<int N>
  constexpr int factorial = factorial<N-1> * N;

template<>
  constexpr int factorial<0> = 1;
```

Potential use cases of variable templates:

- Cleaner representation of template metaprogramming, as seen in the example.
- Provide different versions of some generator object, e.g., for pseudo-random numbers, one for each data type.
- Make *named constants* (e.g., from C enums and macros) type-aware and usable as templates.

The Message Passing Interface (MPI) specifies data types through certain integer constants, make those available for the C++ type system:

```
template<typename T> constexpr MPI_Datatype MPIType;
template<> constexpr MPI_Datatype MPIType<int>    = MPI_INT;
template<> constexpr MPI_Datatype MPIType<float>  = MPI_FLOAT;
template<> constexpr MPI_Datatype MPIType<double> = MPI_DOUBLE;
...
```

This way, generic code can pick the right MPI data type automatically.

6.4. Return Type Deduction

In many cases the trailing return type of lambda expressions is optional, and can be deduced from the return statement instead:

```
auto f = [] (int i) {return 2*i;}; // returns int
```

C++14 extends this mechanism from lambdas to normal functions:

- The return type of lambdas may be omitted, and the return type of normal functions can be "`auto`" without trailing return type.
- In both cases the return type is deduced using the return statement. Multiple return statements with conflicting types cause a compilation error.
- The deduced type is always a value type, like with `auto` in general. Reference types can be specified using `auto&`, `const auto&`, etc. (as a trailing return type in the case of lambdas).

This leads to *two convenient ways to write functions returning tuples or structs*. We don't have to repeat the tuple information several times, doing so once in the return type *or* return statement is enough:

```
// type deduction: don't specify return type,
// has to be fixed somewhere in the function body
auto createTuple()
{
  return std::tuple<int,double,bool>{7,3.2,true};
}

// new init version: specify return type,
// can return simple braced list
std::tuple<double,char,int> buildTuple()
{
  return {5.6,'c',27};
}
```

6.5. Other Improvements and Additions

Deprecated Attribute

Sometimes legacy code has to be kept around for backwards compatibility, but shouldn't be used anymore for various reasons. Such code can be marked with the `[[deprecated]]` attribute, optionally including some reason:

```
[[deprecated("Use more efficient newFoo() function instead.")]]
void foo() { ... }
```

The attribute is placed:

- before function declarations / definitions (see above)
- before the class or struct name: `struct [[deprecated]] Foo { ... };`
- before variable declarations: `[[deprecated]] int i;`

Templates, individual template specializations, and whole namespaces can be marked as deprecated in similar fashion.

Type Traits: Helper Types

Template metaprograms based on type traits can become quite verbose: each such traits class introduces a dependent type named `type` or a Boolean value named `value`, and these have to be written out to specify the resulting type or value.

C++14 introduced *helper types like* `enable_if_t`:

```
template< bool B, class T = void>
  using enable_if_t = typename enable_if<B,T>::type;
```

This can help make SFINAE easier to read, for example. Additionally, there is now a method `constexpr operator()` for the Boolean values:

```
std::enable_if<typename std::is_same<T1,T2>::value>::type>
std::enable_if_t<std::is_same<T1,T2>()>> // C++14 version
```

They could also have introduced helper variables for the Boolean values using variable templates, but that didn't happen until C++17.

7. C++17 Features

C++17 is again a rather small update. The following points are noteworthy additions to the language:

- Guaranteed copy elision
- Compile-time branches
- Structured bindings
- Additional utility classes
- Fold expressions
- Class template argument deduction
- Mathematical special functions
- The filesystems library

7.1. Guaranteed Copy Elision

In C++, exchanging data with a function through arguments and return values typically requires copies:

- *copies of the arguments* are created in the scope of the function that is called
- a *copy of the return value* is created in the scope of the calling function (a nameless temporary), and then a *second copy is necessary during assignment*

Such copies can be memory-intensive and time-consuming, and therefore we often avoid them using references:

- Arguments are passed as `const` references to avoid copies.
- Return values are replaced by non-`const` references as arguments: instead of returning something, the function simply modifies its argument.

```
// computes out = mat * vec
void foo(const Mat& mat, const Vec& vec, Vec& out)
{ ... }
```

The modification of reference-type arguments is efficient, but it makes it harder to see intent and parse the signature: the return values are effectively hidden among the arguments.

Before C++17, the compiler was allowed to eliminate copies that happened when function values were returned, but there was no guarantee for this to happen. This was one of the main

reasons for using reference-type arguments to communicate results: there is no need to trust the optimization phase of the compiler. There were two different kinds of possible optimization, called NRVO and RVO.

Named Return Value Optimization (NRVO):

This is the elimination of copies when a named, local object is returned by the function. In cases where this is possible, the compiler removes any copy and move operations and instead constructs the object directly into the storage where it would end up after copying or moving anyway.

Temporaries and Return Value Optimization (RVO):

This is the elimination of copies when a nameless temporary is used in an assignment. The creation of the temporary is essentially skipped, and the storage of the assignee is used instead. The compiler may perform this optimization in several different contexts, and its application to unnamed return values is called RVO.

To summarize, returning a local variable or temporary and assigning its value to another variable caused up to two copies before C++17, which could be optimized away under certain circumstances:

named return value: NRVO for the return value, then elimination of the temporary that is used in the assignment

nameless return value: elimination of the two temporaries inside and outside of the function (with the former being RVO)

The C++17 standard contains a fundamental change with significant consequences, that is at the same time almost invisible:

- *C++17 introduced the notion of temporaries as values that only exist within the technical specification, and are not actually represented in memory.*
- As a consequence, the temporaries we discussed are never created, and there is no need to optimize them away. This is called *guaranteed copy elision.*
- This eliminates both copies for unnamed return values, and at least one of those for named return values.

In C++17 and above, using return values should almost always be preferred over modifying reference arguments. This holds even for multiple return values, thanks to → *structured bindings*.

7.2. Compile-Time Branches

C++17 introduced *compile-time branches using* `if constexpr`:

- The branch condition is evaluated at compile-time when the `constexpr` keyword is used. Therefore it must also be `constexpr`, of course.
- The non-matching branch is discarded (if present), and will not be compiled. It will not cause any compilation errors, even if its content doesn't make sense for a given set of template parameters.
- Non-matching branches don't take part in return type deduction, i.e., different branches may return different types.

```
// C++17 Boolean helper variable template
if constexpr (std::is_same_v<T1,T2>)
{ ... } // code for case where types coincide
else
{ ... } // code for when T1 and T2 differ
```

Compile-time branches can replace template specializations and function template overloads, and in particular:

- create recursive template metaprograms without base case specializations
- eliminate a subset of SFINAE applications, providing a much more readable replacement

```
// replacement for SFINAE:
// * select function body based on type T
// * deduce return type from chosen branch

template<typename T>
auto foo(const T& t)
{
  // integer case
  if constexpr (std::is_integral_v<T>)
  { ... }
  // floating point case
  else if constexpr (std::is_integral_v<T>)
  { ... }
  // default case
  else
    // have to make the assert depend on T
    // here, else it would fire in any case
    static_assert(not std::is_same_v<T,T>,
      "only integers and floating point!");
}
```

An implementation of nested loops of depth N, with N being a compile-time constant, based on `if constexpr`:

```
template<std::size_t N, typename Callable, std::size_t K = N-1>
  void metaFor(std::array<size_t,N>& indices,
               const std::array<size_t,N>& bounds, Callable&& callable)
  {
    static_assert(K < N, "K must be smaller than N");
    if constexpr (K == 0)
      for (std::size_t i = 0; i < bounds[0]; i++)
      {
        indices[0] = i;
        callable(indices);
      }
    else
      for (std::size_t i = 0; i < bounds[K]; i++)
      {
        indices[K] = i;
        metaFor<N,Callable,K-1>(indices,bounds,
                                std::forward<Callable&&>(callable));
      }
  }
```

7.3. Structured Bindings

C++17 added two new constructs, structured bindings and a version of `if` with initializer, similar to that of `for` loops.

Structured bindings make multiple return values possible by assigning names to components of an object:

```
auto [x,y,z] = f(); // f returns object with three components
```

This works with C arrays, C-style structs, `std::array`, `std::pair` and `std::tuple`. `x`, `y` and `z` are then references of the entries / data members.

Such bindings can be convenient in range-based `for` loops over maps:

```
for (auto&& [first,second] : map)
{...}
```

The `if` with initializer works similar to a `for` loop:

```
if (auto [x,y,z] = f(); x.isValid())
{...}
else
{...}
```

This avoids polluting the surrounding scope, just like the first entry of the `for` loop declaration.

C++20 introduces a similar initializer for range-based `for` loops. It may be used to, e.g., initialize the range of the loop, or provide a local index counter variable.

7.4. Utility Classes

The two Standard Library class templates `std::pair` and `std::tuple` are now accompanied by `std::optional`, `std::variant`, and `std::any`.

`std::optional<T>`:

A type-safe wrapper that may or may not contain a value of type `T`. Follows value semantics, but access uses `*` and `->` operators anyways.

`std::variant<T...>`:

A type-safe replacement for `union`. Provides several safe ways to check / access the currently held value, e.g., `std::get` like `std::tuple`.

`std::any`:

An implementation of *type erasure*, `std::any` can store objects of arbitrary types. Basically a type-safe, resource owning replacement for `void*`.

Comparison of Standard Library utility classes:

Class	Semantics	Copies	Validity	Types
`std::tuple`	value	copyable	always valid	fixed set
`std::shared_ptr`	reference	ref.-counted	nullable	fixed type
`std::unique_ptr`	reference	move only	nullable	fixed type
`std::optional`	value	copyable	nullable	fixed type
`std::variant`	value	copyable	always valid	fixed set
`std::any`	value	copyable	always valid	arbitrary

- All the classes own and manage their data
- Two vector replacements (shared vs. non-shared)
- Two aggregate types (Cartesian product vs. union set)
- Two relaxations on the requirements on C++ variables (optional value vs. arbitrary type)

7.5. Fold Expressions

C++17 introduces *fold expressions*, which can automatically expand a parameter pack (the argument with ellipsis in variadic templates) and apply operators inbetween.

```
template<typename... Args>
int sum(Args&&... args)
{
  return (args + ... + 0);
}

template<typename... Args>
void print(Args&&... args)
{
  (std::cout << ... << args) << std::endl;
}
```

This can be used with 32 predefined binary operators, including all arithmetic and logic operators.

Call a function for each argument, despite not knowing the number of arguments beforehand (folds using the comma operator):

```
template<typename Func, typename... Args>
void callForAll(const Func& f, Args&&... args)
{
  ( f(args), ...);
}
```

Folds can be expanded to the left (right fold, ellipsis on the right) or to the right (left fold, ellipsis on the left), and each version can have an initial value (binary fold) or not (unary fold).

This means: The one operation that is written explicitly is actually the *last* to be executed, not the first, since the fold is realized using recursion.

7.6. Improvements for Templates

C++17 brings some improvements for templates:

- *class template argument deduction*
- *type deduction for non-type template parameters*

Before C++17, only the parameters of function templates could be deduced, those of class templates had to be written out. From C++17 onwards, class template arguments can be omitted as long as they can be deduced from the constructor function call:

```
std::pair   p{1,7.3};      // deduces std::pair<int,double>
std::tuple  t{true,'c',2}; // deduces std::tuple<bool,char,int>
std::vector v{1,2,3,4};    // deduces std::vector<int>
```

Quite often, templates require both a type parameter and one or more non-type parameters of that type. Examples are:

- Constants for template metaprogramming across different types
- A type as parameter, with associated default values, initial values, etc.

In C++17 and above, type deduction can be used in this context:

```
template<typename T, T t> // redundancy
  struct OldConstant {enum {value = t};};

template<auto t>          // type deduction
  struct NewConstant {enum {value = t};};
```

This makes it possible to omit the redundant type information in instantiations, for example `NewConstant<5>`. The type of the parameter is available using `decltype`, should it be needed.

7.7. Mathematical Special Functions

C++ provides a number of mathematical functions, which are of course quite relevent in Scientific Computing:

exponential: exponential function and logarithms, e.g., ln

power: square and cubic root, arbitrary powers

trigonometric: sine, cosine, tangent, etc.

hyperbolic: hyperbolic sine (sinh) and cosine (cosh), etc.

others: e.g., gamma function (generalized factorial)

C++17 added special functions for, e.g., *cylindrical and spherical harmonics*:

- Orthogonal polynomial bases (Legendre, Hermite and Laguerre)

- Cylindrical and spherical Bessel functions
- Cylindrical and spherical Neumann functions
- Beta function and Riemann zeta function

7.8. Filesystems Library

The *filesystems library*[31] introduced capabilities that C++ was sorely missing, because they were either nonexistent or non-portable:

- Representation of directory paths, files, and file types
- Queries (file existence, file size, permissions, etc.)
- Handling of directories and files
- Creation of hard links and symbolic links
- Access to current directory
- Special path for temporary files

The library provides a unified framework that is portable across operating systems. Some filesystems don't provide certain features, e.g., the old FAT system doesn't know symbolic links. In these cases appropriate exceptions are thrown.

[31] See https://en.cppreference.com/w/cpp/filesystem.

8. C++20 Features

C++20, the upcoming new standard, is a major update similar to C++11, and *will drastically change the way modern C++ is written.* Additions to the language include:

- Module support
- Concepts
- Ranges as entities
- Coroutines
- Mathematical constants
- A modern text formatting framework

8.1. Modules

The traditional C++ compilation process is based on that from C, and inherits the following drawbacks:

- Header files are recursively combined into one large text document.
- Header guards are necessary to prevent redefinitions and conflicts.
- Preprocessor macros remain active across header boundaries.
- Consequently, the order of header includes may be important.

C++20 introduces *modules*, which are separate units that can be used to divide code bases into logical parts. The result of importing such a module is well-defined, and macro definitions are kept within their module and are unable to influence the content of other modules.

A module is a source file, just like those including the main function of programs, which contains a number of `export` and `import` statements:

```
export module MyModule; // module declaration
import <iostream>;      // import statements:
import <vector>;        //        replacement for includes

// not visible outside the module
void myInternalFunction() { ... };

// can be made available by importing the module
export void myFunction() { ... };
```

Modules are a replacement for the practice of using header files to represent libraries, which then include all the headers the library requires or provides. Namespaces are orthogonal to modules, and can be used in conjunction with modules or across them.

8.2. Concepts

C++20 introduces *concepts*, which are *named requirements for template parameters*. Their definition is very similar to that of `constexpr` Boolean variable templates:

```
template<typename T>
  concept Number = std::is_integral_v<T>
                   || std::is_floating_point_v<T>;
```

But in contrast to simple variable templates, one can also require that certain methods and operators exist and produce a certain type:

```
template<typename T>
  concept EqualityComparable = requires(T a, T b)
  {
    { a == b } -> std::same_as<bool>;
    { a != b } -> std::same_as<bool>;
  };
```

The main benefit of concepts is the fact that *requirements for template parameters can be checked early*, instead of triggering convoluted error messages somewhere within the instantiation process.

Ideally, *a C++ concept should express the identity and essence of a certain category of types*, e.g., the mathematical concepts of vectors and matrices, etc. Unfortunately, most such properties are hard to write down and can't be checked by the automatic compilation process:

- existence of neutral elements
- commutative, associative, and distributive properties
- ...

This means that one can often only check the interface and existence of operations in practice.

Templates can specify restrictions in several ways:

```
// verbose form, can be used for ad-hoc (i.e., unnamed) concepts
template<typename T>
  auto twice(T i) requires std::integral<T>
  {return 2*i;}

// shorter version, concept has to be defined beforehand
template<std::integral T>
  auto square(T i)
  {return i*i;}
```

The type deduction for function arguments of generic lambda expressions is now also available for normal functions:

```
// usual template declaration
template<typename T1, typename T2>
  auto foo(const T1& a, const T2& b);

// equivalent declaration using auto keyword
auto bar(const auto& a, const auto& b);
```

As with generic lambdas, each use of the `auto` keyword introduces an independent, new template parameter. But type relationships between the arguments can also be modelled in this form: simply specify an appropriate concept based on `decltype`.

This new way of writing templates can be combined with concepts, and the result is an *elegant and easy to read way of function template selection*:

```
// used for types that represent integers
void foo(const std::integral auto& t)
{ ... }

// used for types that represent real numbers
void foo(const std::floating_point auto& t)
{ ... }

// default: used for any other type
void foo(const auto& t)
{ ... }
```

This is a good replacement for many SFINAE constructs.

8.3. Ranges

A direct application of C++20 concepts is the *range concept*, which represents sequences. This includes the usual container classes, but also, e.g., infinite sequences based on generator functions (using lazy evaluation).

Ranges can be filtered and transformed, producing new ranges in the process:

```
for (const auto& e : v | std::views::filter(even))
{ ... } // do something for each element that is even
```

The STL algorithms can now operate on ranges instead of iterator pairs:

```
// old version, kept for backward compatibility
int n = std::count(v.begin(), v.end(), 7);

// new version based on the range concept
int m = std::ranges::count(v, 5);
```

8.4. Concurrency with Coroutines

C++ adds *coroutines* to the language, which are *special functions used for cooperative (non-preemptive) multitasking*. Coroutines are subprograms, like normal functions and threads, but with significant differences:

- Functions are subprograms that are always executed in one go from start to finish, and that don't store any information between invocations. Exactly one function is executed at any given time, and functions waiting for other functions to finish form a call stack, with the main function at the bottom.
- Threads are subprograms that are run independently. They are also executed as a unit, but may be interrupted at any time (preemptive multitasking). Execution continues where it was interrupted, and synchronization using, e.g, mutexes and locks is necessary because there is no well-defined execution order.

Coroutines differ from functions and threads in the following ways:

- A coroutine may *voluntarily yield* at any time, handing the execution flow over to another coroutine.
- Coroutine instances can have *internal state*, similar to functors, and this state is preserved when they yield.

- They can have *multiple reentry points*, i.e., the execution of a coroutine doesn't have to continue at the point where it was stopped.
- There is *no need for synchronization*, because coroutines are collaborative and only one of them runs at any given time.

The support for coroutines in C++20 isn't as fleshed out as that for threads: there are no predefined coroutines yet, only the facilities that are needed for their implementation.

Implementing a coroutine is somewhat similar to writing a functor class or using tasks with their promise—future pairs, yet significantly more verbose, at least for now. Let's consider a small generator semicoroutine[32].

First, we need a promise object, similar to those we know from task-based programming, then a handle, comparable to a future object, and finally a function definition:

```
template<typename T>
  struct GeneratorPromise
  {
    using Handle = std::coroutine_handle<GeneratorPromise<T>>;

    auto initial_suspend()         {return std::suspend_always{};}
    auto final_suspend()           {return std::suspend_always{};}
    auto get_return_object()       {return Handle::from_promise(*this);}
    auto yield_value(const T value){current_value = value; return std::suspend_always{};}
    void unhandled_exception()     {std::abort();}

    T current_value;
  };
```

```
template<typename T>
  struct Generator
  {
    using Handle = std::coroutine_handle<GeneratorPromise<T>>;
    using promise_type = GeneratorPromise<T>;

    // appropriate constructors and destructor
    // to forbid copies but handle moves cleanly
    ...

    T operator()() {return handle.promise().current_value;}
    bool next()    {handle.resume(); return !handle.done();}

    Handle handle;
  };

Generator<int> sequence(int start = 0, int step = 1) noexcept
{
  auto value = start;
  for (int i = 0;; ++i)
  {
    co_yield value;
```

[32] A semicoroutine is a simple coroutine that can only yield to its caller.

```
    value += step;
  }
}
```

- The generator is used by storing the return value of the function and calling its `next()` method to obtain numbers.
- The two structs can be reused to define other generators, simply by writing another function that also returns a generator object.

8.5. Mathematical Constants

While C++17 added special functions that are often needed in Scientific Computing, C++20 introduces variable templates for *mathematical constants*. Before C++20, well-known constants like, e.g., π or e are usually available as C-style macros like `M_PI`. The variable templates serve as a modern replacement that is part of the standard.

The predefined constants include:

`e_v<T>`: Euler's number $e \approx 2.7182918\ldots$

`pi_v<T>`: the circle constant $\pi \approx 3.1415926\ldots$

`egamma_v<T>`: the Euler-Mascheroni constant $\gamma \approx 0.57721566\ldots$

`phi_v<T>`: the golden ratio constant $\phi \approx 1.6180339\ldots$

Additionally, a small number of square roots and logarithms are available, like $\ln(2)$ and $\sqrt{2}$. Specializations for `float`, `double` and `long double` are provided for each variable template.

8.6. Text Formatting

Before C++20, there are two separate ways to format text output:

- The well-known I/O streams. These provide a safe and extensible interface, but format strings and arguments are intertwined, which...
  - makes automatic generation / modification difficult (e.g., localization)
  - makes code with many formatting commands hard to read
- The old `printf()` formated output function from C. While this clearly separates format and arguments, this function is neither safe nor extensible and should be avoided.

C++20 introduces modern text formatting capabilities to C++ in the from of its *format library*[33]:

- Based on templates and therefore extensible
- Performant implementation available
- Format specifiers consistent with those from other languages

[33] https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification

A. Exercises

This appendix contains the exercises accompanying the lecture. Some of them may reference "the lecture" or previous "sheets", but it should be clear from context which section or subsection of this document is meant by that. One of the exercises mentions "an implementation on the website". This code is not provided within this document, but is just a sample solution to one of the previous exercises anyway.

Several of the exercises (the quizzes) link to Anders Schau Knatten's website cppquiz.org. As with any other hyperlink, the referenced contents may vanish without notice, but hopefully the links will stay valid for quite some time.

A.1. Fundamentals and Standard Library

C++ Quiz

On https://cppquiz.org you can find a quiz with C++ specific questions. In this exercise, answer the following questions:

Question 1: https://cppquiz.org/quiz/question/197 (variable lifetime)

Question 2: https://cppquiz.org/quiz/question/161 (Duff's Device)

Question 3: https://cppquiz.org/quiz/question/9 (reference arguments)

Question 4: https://cppquiz.org/quiz/question/113 (overload resolution)

Question 5: https://cppquiz.org/quiz/question/5 (initialization order)

The questions are sorted (more or less) according to the structure of the lecture. For questions 1, 3, 4, and 5, write a short statement what information the question and its solution are trying to convey. Regarding question 2: inform yourself about the construct that is used. What is its purpose? Would you suggest using this in real-world code? Why, or why not?

Rational Numbers

Write a class for rational numbers. The number should always be represented as a *fully reduced fraction* of the form

$$\frac{\text{numerator}}{\text{denominator}}$$

with denominator > 0.

1. What is an appropriate data structure for rational numbers?
2. Start by writing a function `int gcd(int,int)` (greatest common divisor), you will need it to reduce fractions.
   - You can use the Euclidean algorithm to determine the greatest common divisor.

- For an algorithm see https://en.wikipedia.org/wiki/Greatest_common_divisor
- Implement this scheme as a recursive function.

3. Write a class `Rational`, which represents a rational number. The constructor should have the numerator and the denominator as arguments. Be sure to check for valid input. In addition, the class has two functions `numerator()` and `denominator()` that return the values of the numerator and denominator. The class should have three constructors:
   - a default constructor that initializes the fraction with 1,
   - a constructor that initializes the fraction with a given numerator and denominator, and
   - a constructor that initializes the fraction with a given whole number.
4. Supplement the class with operators for `*= += -= /=` and `==`.
5. Use the newly implemented methods to implement free operators `* + - /`.
6. Check your implementation using various test cases. Initialize three fractions

$$f_1 = -\frac{3}{12}, \quad f_2 = \frac{4}{3}, \quad f_3 = \frac{0}{1}.$$

   Test the operators with the following examples:

$$f_3 = f_1 + f_2, \quad f_3 = f_1 \cdot f_2, \quad f_3 = 4 + f_2, \quad f_3 = f_2 + 5,$$
$$f_3 = 12 \cdot f_1, \quad f_3 = f_1 \cdot 6, \quad f_3 = \frac{f_1}{f_2}.$$

   Print the result after each operation. The corresponding solutions are:

$$\frac{13}{12}, \quad -\frac{1}{3}, \quad \frac{16}{3}, \quad \frac{19}{3}, \quad -\frac{3}{1}, \quad -\frac{3}{2}, \quad -\frac{3}{16}.$$

Farey Sequences

A Farey sequence F_N of degree N (or: the Farey fractions of degree N) is an ordered set of reduced fractions

$$\frac{p_i}{q_i} \qquad \text{with} \qquad p_i \leq q_i \leq N \qquad \text{and} \qquad 0 \leq i < |F_N|$$

and

$$\frac{p_i}{q_i} < \frac{p_j}{q_j} \qquad \forall\, 0 \leq i < j < |F_N|.$$

Use the class `Rational` from the previous exercise to write a function

```
void Farey(int N)
```

which calculates the Farey fractions up to degree N and prints the resulting Farey sequences up to degree N on the screen.

Algorithm: The sequences can be computed recursively. The first sequence is given by

$$F_1 = \left(\tfrac{0}{1}, \tfrac{1}{1}\right)$$

For a known sequence F_N one can get F_{N+1} by inserting an additional fraction $\frac{p_i+p_{i+1}}{q_i+q_{i+1}}$ between two consecutive entries $\frac{p_i}{q_i}$ and $\frac{p_{i+1}}{q_{i+1}}$ if $q_i + q_{i+1} = N + 1$ holds for the sum of denominators.

Example: Determining F_7 from F_6 results in the following construction:

$$F_6 = \Big(\underbrace{\tfrac{0}{1}, \tfrac{1}{6}}_{\frac{1}{7}}, \tfrac{1}{5}, \underbrace{\tfrac{1}{4}, \tfrac{1}{3}}_{\frac{2}{7}}, \underbrace{\tfrac{2}{5}, \tfrac{1}{2}, \tfrac{3}{5}}_{\frac{3}{7} \text{ and } \frac{4}{7}}, \underbrace{\tfrac{2}{3}, \tfrac{3}{4}}_{\frac{5}{7}}, \tfrac{4}{5}, \underbrace{\tfrac{5}{6}, \tfrac{1}{1}}_{\frac{6}{7}} \Big)$$

The new elements are:

$$\tfrac{0+1}{1+6} = \tfrac{1}{7} \;\; ; \;\; \tfrac{1+1}{4+3} = \tfrac{2}{7} \;\; ; \;\; \tfrac{2+1}{5+2} = \tfrac{3}{7} \;\; ; \;\; \tfrac{1+3}{2+5} = \tfrac{4}{7} \;\; ; \;\; \tfrac{2+3}{3+4} = \tfrac{5}{7} \;\; ; \;\; \tfrac{5+1}{6+1} = \tfrac{6}{7}$$

The sorted sequence then is:

$$F_7 = \left(\tfrac{0}{1}, \mathbf{\tfrac{1}{7}}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \mathbf{\tfrac{2}{7}}, \tfrac{1}{3}, \tfrac{2}{5}, \mathbf{\tfrac{3}{7}}, \tfrac{1}{2}, \mathbf{\tfrac{4}{7}}, \tfrac{3}{5}, \tfrac{2}{3}, \mathbf{\tfrac{5}{7}}\ \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \mathbf{\tfrac{6}{7}}, \tfrac{1}{1}\right)$$

For checking:
The Farey sequences up to degree 6

$$\begin{aligned}
F_1 &= \left(\tfrac{0}{1}, \tfrac{1}{1}\right) \\
F_2 &= \left(\tfrac{0}{1}, \tfrac{1}{2}, \tfrac{1}{1}\right) \\
F_3 &= \left(\tfrac{0}{1}, \tfrac{1}{3}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{1}{1}\right) \\
F_4 &= \left(\tfrac{0}{1}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{1}{1}\right) \\
F_5 &= \left(\tfrac{0}{1}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{1}{2}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{1}{1}\right) \\
F_6 &= \left(\tfrac{0}{1}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{1}{2}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \tfrac{1}{1}\right).
\end{aligned}$$

There is a beautiful illustration of these fractions, the Ford circles[34].

Matrix Class Implementation

Note: this exercise will form the foundation for several of the upcoming exercises, so if you want to work on only a subset of the sheet, it might be a good idea to start on this one.

In Scientific Computing the concept and use of matrices is crucial. Regardless of the field of expertise — if it is in optimization, statistics, artificial intelligence, or the solution of partial differential equations — we need matrices and solutions of linear systems of equations in nearly all applications.

[34] see https://en.wikipedia.org/wiki/Ford_circle

In this exercise, we will implement a class `Matrix` in analogy to the class `Vector` from exercise sheet 1.

Take care of the following points:

1. Class `Matrix` should have all functionality that class `Vector` has (constructors, methods, etc.).
2. The entries should be stored in a container of type `std::vector<std::vector<double>>`.
3. Instead of the number of elements `int N` in class `Vector`, class `Matrix` should contain the two numbers `int numRows` and `int numCols` that represent the number of rows and the number of columns respectively. Use `numRows` and `numCols` in all places where the member functions of class `Vector` used `N`, and re-implement the functionality adapted to the use of a class that represents matrix objects.
4. Use the member function `double& operator()(int i, int j)` for accessing the (i, j)-th element of objects of class `Matrix`.
5. Use the member function `std::vector<double>& operator[](int i)` to return the i-th row of a matrix object.
6. Class `Matrix` should have an additional constructor that constructs square matrices.

   In addition to the member functions mentioned above, implement free functions that provide

7. the addition of two matrices,
8. the multiplication of a matrix with a scalar,
9. the multiplication of a scalar with a matrix,
10. a matrix-vector multiplication, where vectors are of type `std::vector<double>`,
11. a matrix-vector multiplication, where vectors are of type `Vector`.

Write a test program that tests all functionality of class `Matrix` (construction, the different kinds of multiplication, element access).

Linked List

Using the simple example of a chained list we will practice the interaction of constructors, destructors and pointers.

We want to program a linked list which can store an arbitrary number of values of type `int`. Such a list consists of an object of class `List`, which refers to a sequence of objects of class `Node`. The list elements are stored in a component `int value` within each node, and a pointer `Node* next` points to the next node. The end of the list is designated by the pointer `next` having the value `nullptr`.

1. What is special about a pointer having the value `nullptr`?

2. Implement the class `Node`. Make sure that all member variables are always initialized, especially the `next` pointer.

3. Implement the class `List` with the following methods:

```
class List
{
  public:

    List ();                            // create an empty list
    ~List ();                           // clean up list and all nodes
    Node* first () const;               // return pointer to first entry
    Node* next (const Node* n) const; // return pointer to node after n
    void append (int i);                // append a value to the end
    void insert (Node* n, int i);       // insert a value before n
    void erase  (Node* n);              // remove n from the list
};
```

   `List` must also store the beginning of the list, where would you place it in the class declaration? The `next` pointer of class `Node` should be `private` to ensure that the list structure isn't accidentally changed outside of class `List`. The member `value` is `public` to allow read and write access from outside the class. The line `friend class List;` has to be inserted into the declaration of the class `Node` to give the `List` class access to the `next` pointer. Additionally make sure that the destructor deletes all allocated `Node` objects.

4. Test your implementation with the following program:

```
int main ()
{
  List list;
  list.append(2);
  list.append(3);
  list.insert(list.first(), 1);

  for (Node* n = list.first(); n != 0; n = list.next(n))
    std::cout << n->value << std::endl;
}
```

5. What happens if one copies the list? And what happens if both lists are deleted?

```
int main ()
{
  List list;
  list.append(2);
  ...
  List list2 = list;
}
```

C++ Quiz 2

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/124 (template template parameters)

Question 2: https://cppquiz.org/quiz/question/32 (constructor calls)

Question 3: https://cppquiz.org/quiz/question/125 (static in templates)

Question 4: https://cppquiz.org/quiz/question/17 (con-/destructors and inheritance)

Question 5: https://cppquiz.org/quiz/question/208 (inserting elements into maps)

Questions 2 and 4 discuss important points about constructors and destructors, while questions 1 and 3 are about minor details that might be interesting. What would you say about question 5?

Linked List (Const Correctness)

On the previous sheet you programmed a linked list to practice the interaction of constructors, destructors and pointers. We now want to extend this implementation, practicing const correctness and clean encapsulation.

1. The `const` keyword has already been added to some of the methods and their arguments. Check if these qualifiers make sense and whether some are missing elsewhere, discuss:
   - Which methods should or should not be `const`, and why?
   - What is a good choice for the parameters and return values?
   - What about data members of `List`, and what about the `Node` class?
2. Add a `const` method `max()` that returns the maximum of the stored values, as long as the list is non-empty[35]. For efficiency, this value should be cached and only recomputed when it it requested and has become invalid in the meantime.

[35] Throwing an exception for empty lists would be good style, but here you might just return zero instead.

- What effect does the keyword `mutable` have and why is it needed here?
- What would be inefficient about updating the stored value every time the list is modified in some way?

3. A function for printing the list could look as follows:

```
void printList (const List& list)
{
  for (const Node* n = list.first(); n != 0; n = list.next(n))
    std::cout << n->value << std::endl;
  std::cout << "max: " << list.max() << std::endl;
}
```

- What exactly do the two instances of `const` refer to, especially the second one?
- Test your implementation with this function.

STL Container Types

In the lecture the different types of containers of the Standard Library have been discussed. Each type offers certain guarantees and is ideally suited for specific use cases. Here is a list of scenarios, what type of container would you use and why?

1. You are writing a finite difference discretization of the Laplace equation, and you are using a structured grid. For each grid node, the solution at this point is stored. All nodes of the grid are numbered consecutively and their number is known a priori.

   Which type of container is suitable for storing the node values?

2. You are writing a Quicksort sorting algorithm, and you want to store the pivot elements in a stack structure to avoid recursive function calls.

   Which type of container is well suited to save the pivot elements and which to store the data itself?

3. When implementing direct solvers for sparse matrices, it is common to first minimize the bandwidth of the matrix in order to reduce the storage requirements of the resulting LU decomposition. One method for this is the Cuthill-McKee algorithm. In the course of this algorithm elements have to be buffered in a FIFO (first-in, first-out) data structure. The order of the elements in this FIFO does not change.

   What type of container would you use as a FIFO?

4. In your finite difference program, the solution is defined a priori on a part of the boundary. Nodes in this part of the boundary (Dirichlet nodes) aren't real degrees of freedom and must be treated differently when assembling the finite difference matrix. The number of these nodes is small compared to the total number of nodes.

You want to dynamically read the list of Dirichlet nodes and their solution values from a configuration file and make them accessable node by node. When solving linear PDEs the limiting factor is typically the memory, as you want to solve very big problems.

In what container you would store the indices of Dirichlet nodes and their values?

Pointer Puzzle

Remark: you may actually try this with your compiler.

Look at the following program:

```
void foo ( const int** );

int main()
{
  int** v = new int* [10];
  foo(v);
}
```

The compiler will exit with an error message, because you make a `const int**` out of the `int**`, something like this:

```
g++ test.cc -o test
test.cc: In function 'int main()':
test.cc:6: error: invalid conversion from 'int**' to 'const int**'
test.cc:6: error:   initializing argument 1 of 'void foo(const int**)'
```

Actually, shouldn't it always be possible to convert from non-`const` to `const` ...? Why doesn't this apply here?

Tip: It's clear why the following program doesn't compile:

```
const int* bar ();

int main()
{
  int** v = new int* [10];
  v[0] = bar();
}
```

What is the relation between this program and the one above?

A.2. Advanced Topics

Matrices and Templates

In the lecture templates were presented as a technique of generic programming. They allow the development of algorithms independent of underlying data structures.

In this exercise we will extend an existing implementation of a matrix class to a template class. Such matrices are needed again and again in numerical software. On a previous sheet you had to implement matrices for `double` values. Depending on the application you want, however, it may be useful to have matrices based on other types of numbers, for example `complex`, `float` or `int`. Templates may reduce code redundancy, so that one has an implementation of `Matrix` that can be used for all the different types.

As a reminder, here is the functionality you had to implement:

- Matrix entries are stored as a vector of vectors, i.e., `std::vector<std::vector<double>>`.
- The matrix dimension is handled by two private `int` values, `numRows` and `numCols`.
- The parenthesis operator allows access to individual entries, while the bracket operator gives access to a whole row:

```
double& operator()(int i, int j);
std::vector<double>& operator[](int i);
```

- Free functions provide functionality for the multiplication of matrices with vectors and scalars, and for the addition of two matrices.

For your convenience, an implementation of such a matrix class can be found on the lecture website, based on the following files:

- `matrix.hh`
- `matrix.cc`
- `testmatrix.cc`

The first two contain the definition and implementation of `Matrix` for `double`, while the third file contains a test program. You may compile it like this:

```
g++ -std=c++11 -Og -g -o testmatrix matrix.cc testmatrix.cc
```

In addition to the methods specified in the previous exercise, this version also provides `const` versions of the parenthesis and bracket operators for read-only access, implements the free functions based on assignment operators `+=` and `*=`, provides `resize` methods, and in turn drops the second matrix-vector multiplication method.

You are free to use either your own version or the provided one as a basis for the following tasks. Do the following:

1. Change the implementation of `Matrix` to that of a template class, so that it can be used for different number types.
2. Turn `numRows` and `numCols` into template parameters. Remove any function that doesn't make sense under this change (e.g., `resize` if you use the provided version), adapt the constructors accordingly, and remove any runtime bounds checks that become unneccesary.
3. Change the free functions into template functions that match the aforementioned template parameters. You may assume that scalars, vectors and matrices all use the same element type. Replace `std::vector` by a template template parameter, so that the function would also work for, e.g., `std::array`.
4. Change the main function of the test program, so that your template variants are tested. The program should use all data types mentioned above, with `complex` referring to `std::complex<double>` and the required header being `<complex>`. Also test the matrix with your `Rational` class. For the `print` functionality you have to define a free function

   ```
   std::ostream& operator<< (std::ostream& str, const Rational& r)
   ```

   that prints the rational number by first printing its numerator, then a "`/`" and then its denominator.

Matrices and Templates (II)

This exercise is an extension of the templatization exercise above.

1. Split the templated class `Matrix` into two classes:
   - A class `SimpleMatrix`, without numerical operators. This represents a matrix containing an arbitrary data type and has the following methods, all taken from `Matrix`:
     - some constructors
     - `operator()(int i, int j)` for access to individual entries
     - `operator[](int i)` for access to whole rows
     - a `print()` method
   - A numerical matrix class `NumMatrix` derived from `SimpleMatrix`. The numerical matrix class additionally provides the arithmetic operations functionality mentioned above.

   Also modify the free functions accordingly.

2. Write a test program that covers the two classes `SimpleMatrix` and `NumMatrix`. Your test program does not need to test each number type again, an arithmetic test with `double` is enough. Instead, test your `SimpleMatrix` with `string`s as data, and check whether you can create and use objects of type `SimpleMatrix<NumMatrix<...>,...>` and `NumMatrix<NumMatrix<...>,...>`. Which methods are you able to use? What happens when you try to call the `print` method?

   The latter matrix type is known as a block matrix, a type of matrix that arises in numerical discretizations and can be used to design efficient algorithms if it has additional structure.

3. Write template specializations of `SimpleMatrix` so that print functionality is restored for block matrices. Block matrices should be printed as one big matrix, with "`|`" as horizontal separators between matrix blocks and rows of "`---`" to separate the blocks vertically (it is okay if it isn't the most beautiful output). *Note:* it might be helpful to assign the dimensional template parameters to public enums, so that you have the dimensions available during printing. What do you observe when implementing this?

   This would be a rather good opportunity to use SFINAE to provide two different `print` methods and choose the right one based on the template parameters, instead of having to provide several different base classes.

4. The provided matrix class (and most likely also your own) simply aborts the program when matrix-vector products with mismatching dimensions are attempted. Replace the relevant checks with adequate exception handling. Derive your own custom exception from `std::exception`, or pick a fitting one from the predefined ones if applicable. Add a try/catch block to your test that triggers an exception and prints an explanatory message.

C++ Quiz 3

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/162 (unqualified name lookup)

Question 2: https://cppquiz.org/quiz/question/44 (dynamic polymorphism)

Question 3: https://cppquiz.org/quiz/question/29 (virtual during con-/destruction)

Question 4: https://cppquiz.org/quiz/question/224 (abstract base classes)

Question 5: https://cppquiz.org/quiz/question/133 (diamond pattern)

Question 1 is about name lookup rules, and questions 2 to 5 have a look at dynamic polymorphism, virtual methods, and virtual inheritance. Do the results match with your expectations? Discuss.

Interfaces: Integration

In the lecture the concept of interfaces was introduced, using virtual methods and abstract base classes. This exercise considers numerical integration as an example application.

The goal is to write a code library that allows, via numerical integration, to determine the integral of an arbitrary function $f(t)$ on a given interval $t \in [A, B]$ ($A, B \in \mathbb{R}$). The interval $[A, B]$ is divided into N subintervals of uniform length Δt.

$$\Delta t = \frac{B - A}{N}, \quad t_i = A + \Delta t \cdot i, \quad i = 0, \ldots, N$$

On each of the subintervals the integral can be estimated using an appropriate quadrature rule. Summing up these estimates produces a *composite quadrature rule*:

$$\int_A^B f(t)dt = \sum_{i=0}^{N-1} \left(Q_{t_i}^{t_{i+1}}(f) + E_{t_i}^{t_{i+1}}(f) \right)$$

Here Q_a^b refers to the result of the quadrature rule on the interval $[a, b]$ and E_a^b to the error. Quadrature rules are normally based on polynomial interpolation. This means it is possible to specify the maximum degree of polynomials that are integrated exactly and the order of the error in the case of functions that can't be integrated exactly.

In this exercise, we want to limit ourselves to two quadrature rules:

- The trapezoidal rule:
$$Q_{\text{Trapez}}{}_a^b(f) = \frac{b - a}{2}(f(a) + f(b))$$
It is exact for polynomials up to first order, and the error of the resulting composite rule is $O(\Delta t^2)$.
- The Simpson rule:
$$Q_{\text{Simpson}}{}_a^b(f) = \frac{b - a}{6} \cdot \left(f(a) + 4f\left(\frac{a + b}{2}\right) + f(b) \right)$$
It is exact for polynomials up to degree two, and the error of the resulting composite rule is $O(\Delta t^4)$.

A good test for such a numerical integration is to study the order of convergence. For a problem with a known solution, calculate the error E_N for N subintervals and for $2N$ subintervals. Doubling the number of subintervals cuts the interval width Δt in half, reducing the error. The quotient

$$EOC = \frac{\log(E_N / E_{2N})}{\log(2)}$$

is an indication of the order of convergence of the error (e.g., for the trapezoidal rule you should observe order 2).

1. Design interfaces for the different concepts. To this end, proceed as follows:

- First, identify the abstract concepts in our problem.
  - We want to evaluate integrals of various functions; so we have the *function* as an abstract concept.
  - An integral is represented as a sum of integrals of subintervals.
  - On each subinterval a suitable *quadrature formula* is applied.
  - The quadrature formula evaluates the function to be integrated at certain points in order to determine an approximate solution.
  - A *composite quadrature rule* sums the results.
  - A suitable *test problem* would contain
    - a function to integrate
    - a choice of quadrature rule and composite rule
    - the interval bounds
    - the true resulting integral value
- How are these abstract concepts mapped to interfaces? Usually, one introduces a base class as an interface description for each concept.
  - What is a suitable interface for a function?
  - A quadrature rule evaluates a function on a given interval, how would a suitable virtual method look like?
  - Each quadrature rule can also report the degree of exactly integrable polynomials, and report the order of convergence of the integration error that results.
  - The actual integration (i.e., composite quadrature rule) is parameterized with a quadrature rule and a function. It evaluates the integral of the function on an interval that has to be specified, dividing the interval into N subintervals. In addition, the quadrature formula that is to be used must be specified.
  - Apart from the equidistant composite quadrature used here, there may be other approaches (see second exercise). In anticipation of extensions, also introduce the abstract concept of a composite quadrature rule, although right now there is only one actual implementation.
  - What would be a suitable interface for test problems?

2. Describe your interfaces and explain the design decisions. What are the reasons for your specific choice of interfaces?

3. Implement the interfaces using abstract base classes and dynamic polymorphism. Write implementations for all of the above variants of abstract concepts (trapezoidal rule, Simpson's rule, equidistant composite rule). Your functions and classes should be as general as possible, i.e., use interface classes for arguments and return values. Make use of the best practices introduced in the lecture.

4. Test your implementation with the integration of $2t^2 + 5$ on the interval $[-3, 13]$ and $\frac{t}{\pi}\sin(t)$ on the interval $[0, 2\pi]$. To do this, implement appropriate test problems for each combination of test function and quadrature rule. Write a free function that expects a test problem and a number of steps to perform, and then

   - reports the true integral value and expected convergence order
   - starts with one big interval without subdivisions
   - doubles the number of subintervals after each step
   - applies the composite rule in each step
   - reports in each step:
     - the current estimated value
     - the resulting error
     - the estimated order of convergence (EOC)
     - the deviation from the expected order

   Discuss the results.

Adaptive Integration

An improvement over the integration using equidistant intervals is adaptive integration. Here one tries to only use small intervals Δt where the error is actually large.

If you are in need of additional points or not challenged enough `;-)` , you may write a class for adaptive integration that complements the equidistant one from the previous exercise. This new version should use the same interface for functions and quadrature rules and fit into the test infrastructure you have written.

One can estimate the error in one of the subintervals by comparing the result of quadrature with the result of an interval that has been refined once, i.e., the local error is estimated through hierarchical refinement. The goal is to have the "error density", i.e., error per unit length of interval, to be about the same in all subintervals. Subintervals with large errors are divided again, iteratively or recursively. One starts with a small number of subintervals and repeats the error estimation and refinement until the desired number of subintervals is reached.

To avoid having to constantly reevaluate the integrals of the individual subintervals, it is helpful to store the subintervals, integrals and estimated errors in appropriate data structures: note that the refinement used to estimate the local error will immediately become part of the set of

subintervals if this error proves to be large. The containers of the C++ Standard Library, e.g., `std::map` and `std::deque`, might prove useful. The same holds for function evaluations: note that one half resp. one third of the function evaluations in the previous exercise were actually unnecessary and quite wasteful, because interval bounds were generally evaluated twice.

You are free to design more or less efficient algorithms for the local error estimation and for deciding which intervals should be bisected and in which order. Your primary concern is a program that produces good approximations w.r.t. the number of intervals used, thoughts about runtime efficiency are a good idea but not mandatory.

The points of this exercise don't count towards the total number of points that can be achieved during the semester. Consequently, you can do this exercise instead of another one later on or in addition to the other exercises to achieve a higher point percentage.

Template Metaprogramming: Number Sequences

Template metaprogramming might be an unusual programming technique, but it is Turing complete: this can be shown by simply implementing a Turing machine in C++ templates (you can find some online if you are interested). This shows that anything you can write as a normal program could, at least in principle, be achieved using template metaprogramming (see Church-Turing conjecture[36]). The main obstacles are the finite amount of recursive template levels the compiler allows, the fact that dynamic interaction with the user is impossible, the fact that floating-point numbers can't be template parameters, and of course the slightly awkward way things have to be written. The only way to provide some form of user input is through preprocessor macros, since the usual command line arguments and queries using `std::cin` are not known at compile time.

Solve the following two tasks with template metaprogramming:

1. **Fibonacci Numbers.** Write both a template metaprogram and a normal function that compute the k-th element of the Fibonacci sequence for given $k \in \mathbb{N}$:

$$a_k := \begin{cases} 1 & \text{for } k = 0 \\ 1 & \text{for } k = 1 \\ a_{k-2} + a_{k-1} & \text{else} \end{cases}$$

   Base your two implementations on the mathematical definition above, i.e., directly implement the given recursion formula. Then write a main function that calls both versions for the argument "`INDEX`", and prints the results. Here, `INDEX` is a preprocessor macro, which we use instead of command line arguments or user input to control which index k is used. You may define this index during the compilation process, e.g., for GCC:

```
g++ -D INDEX=<your value> -o main main.cc
```

[36] https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis

Remember that you have to recompile your program if you want to change this index.

Measure both the compilation time and runtime of your program from $k = 5$ to $k = 50$ in steps of 5 (under Linux you can use the "`time`" command for this task). What do you observe? Plot both times over the index k and discuss your observations. What is the explanation for the asymptotics? There are significantly faster algorithms for the computation of Fibonacci numbers[37]. Would you suggest using one of these other algorithms for the template metaprogram, the function, or both?

2. **Prime Numbers.** Write a second template metaprogram that prints the N-th prime number. Remember that a prime number is a natural number larger than one, that is only divisible by one and itself. Here are some steps that you may follow in your implementation:

   - Start with a struct `is_prime<int P>` that exports a `const static bool` that is `true` for prime numbers and false for all other numbers. A simple implementation uses the signature

     ```
     template<int P, int K = P - 1> struct is_prime;
     ```

     that checks whether P is not divisible by any number smaller or equal to K (except for one, of course). A recursive definition of this function is straight-forward, using the modulo operator and simple Boolean algebra. The default case then provides the desired struct.

   - Then write a second struct `next_prime<int N>` that exports the smallest prime P with $P \geq N$. If N is prime, then this is simply N itself, else we can use recursion again by exporting the smallest prime P with $P \geq (N + 1)$. Why is this recursion guaranteed to terminate? The recursion needs to branch on whether N itself is prime or not, but both conditionals (`if/else`) and the ternary operator (`X ? Y : Z`) would always instantiate the next recursion level, whether X is true or false, leading to infinite recursion. From C++17 onwards one could use the `constexpr if` construct to only instantiate one of the two branches, thereby solving the problem, but your code should be valid C++98/03. Once again, this can be solved with a slightly extended signature:

     ```
     template<int P, bool B = is_prime<P>::value> struct next_prime;
     ```

     How can you use this second (auto-evaluated) parameter to get the desired branching behavior?

   - Finally, write a struct `prime<int N>` that exports the N-th prime number, making use of the other structs you have defined. This definition is also recursive. What is the base case? How can you compute the correct prime number for all the other cases recursively?

[37] https://www.nayuki.io/page/fast-fibonacci-algorithms

Use the aforementioned preprocessor macro trick to evaluate your template metaprogram for different values of N, and check that the right sequence is created.

Matrices: An Application of SFINAE

One of the major drawbacks of class template specializations is the need to reimplement the whole class template, even if only a single method has to be modified. This leads to significant code duplication, especially for classes that provide a large number of methods, and inconsistencies between the different specializations may occur after some time if updates aren't copied to all the different specializations.

SFINAE can be used to solve this issue: there is no explicit specialization of the class template, instead there are simply two (or more) different versions of the one method that needs to be changed, with exactly one of them being instantiable for any parameterization of the class template. This technique can be used for both normal methods and methods that are function templates themselves: if the method is a template, then it simply gains an additional template parameter for the `std::enable_if`, and if it is an ordinary function it becomes a template with a single parameter instead.

1. Apply SFINAE to the `print()` method of your matrix implementation. As discussed in a previous exercise, the default implementation is only correct if the elements of a given matrix are actual numbers, and doesn't work anymore if the matrix is actually a block matrix, i.e., a matrix with entries that are themselves matrices. Instead of redefining the whole matrix class for such cases, introduce two versions of the `print()` method, one that is instantiated when the entries are scalars, and one that is instantiated when the entries are matrices.

   You can reuse the function bodies if you have completed the previous exercise, else you will have to implement one of them. To determine whether a given matrix is a block matrix or not, check the following properties of the entries:

   - Does the type of the entries export a number of rows? (You may have to introduce a public `enum` in your matrix class to make this available if you haven't already done so.)
   - Does the type of the entries export a number of columns? (With an `enum` as above.)

   You may assume that anything exporting these two constants is actually a matrix (and not, say, a higher-dimensional tensor or similar). Make sure that exactly one of the conditions in the `std::enable_if` clauses is true for any potential entry data type.

2. Assume for a moment that there are both matrix classes with public enums as above, and matrices that make the number of rows / columns available via methods (which is what a matrix implementation with dynamic size would do). How would your SFINAE have to change if it should work with

   - normal entries such as numbers (`double`, `int`, ...)

- matrices that export enums as treated by your implementation
- matrices that provide this information via methods instead

You don't have to actually implement the functionality needed for the checks: assume that all required structs are already provided, and write down what the signatures of the three implementations would look like.

3. SFINAE can also be used to provide two or more competing (essentially non-specialized) templates, which can eliminate the need for several different function overloads. Consider the multiplication operators, there are usually three different types:

   - with a scalar
   - with a vector
   - with another matrix

   The first two of these have already been implemented: for concrete types, or as templates parameterized by container types. However, the code for these three types of multiplication is usually quite generic, and would work the same way for several different types of scalars, vectors, and matrices. This makes the implementation via completely abstract function templates attractive, i.e., with the type of the second argument as template parameter. *The problem:* all three types of multiplication would have the same function signature.

   Use SFINAE to work around this problem:

   - The existence resp. nonexistence of which methods can be used to characterize scalars, vectors, and matrices?
   - Provide traits classes `is_vector<T>` and `is_matrix<T>` that can detect whether a given type matches the vector resp. matrix interface.
   - Implement all three types of multiplication as free function templates, parameterized by the type of the second argument, and use SFINAE to select the correct template version for a given type of operand.

   Modify one of your existing test cases to make sure that your SFINAE construct works and picks the right template in each case.

A.3. Modern C++ Features

C++ Quiz 4

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/163 (automatic type deduction)

Question 2: https://cppquiz.org/quiz/question/236 (operator auto)

Question 5: https://cppquiz.org/quiz/question/112 (move semantics)

Question 3: https://cppquiz.org/quiz/question/226 (move semantics II)

Question 4: https://cppquiz.org/quiz/question/116 (perfect forwarding)

Question 1 and 2 are about automatic type deduction, with the latter being more of a peculiarity. The other three questions are about rvalue references, forwarding references, and move semantics: what one might expect to happen, and what actually happens according to the standard.

Automatic Type Deduction

Take your most recent matrix class implementation (it doesn't actually matter which one as long as it is templatized). We would like to make use of automatic type deduction at several points within the matrix class, resp. classes.

1. Replace the return type of all the methods with `auto`. At least one method each should be of the following form:
   - trailing return type with explicit specification of the resulting type
   - trailing return type, declaring the return type with `decltype` and the function arguments
   - no trailing return type, simply deduct the resulting type from the return statements (requires compiling with C++14 or above)
2. Check the attributes of the class: is it possible to replace some of their types with `auto`? If yes, which ones? If not, why not?
3. Check the local variables of the methods, and replace the explicit type with `auto` wherever possible.
4. After having converted all possible locations to `auto`: which instances of using `auto` are actually a good idea, if any? Are there locations where this modification doesn't add any value, in your opinion? Discuss.

Constant Expressions

In a previous exercise you were tasked with writing template metaprograms to compute two number sequences, namely the Fibonacci sequence and the prime number sequence. In this exercise we will reimplement these two sequences, but with constant expressions instead of templates.

Solve the following two tasks using `constexpr` functions. Note that these functions should be valid in C++11, i.e., each function must consist of a single return statement. This means that `if` / `else` branches can't be used, you will have to rely on the ternary operator (`x ? y : z`) instead. The value of this expression is y if the condition x is true, else z.

1. **Fibonacci Numbers.** If you solved the previous exercise, you should already have a normal function that computes the Fibonacci sequence. Modify this function so that is becomes a valid C++11 `constexpr` function (the extent of modifications depends on your chosen implementation, of course). Simply implement the function directly if you didn't do the previous exercise. Here is a quick reminder of the mathematical definition of the k-th element, $k \in \mathbb{N}$:

$$a_k := \begin{cases} 1 & \text{for } k = 0 \\ 1 & \text{for } k = 1 \\ a_{k-2} + a_{k-1} & \text{else} \end{cases}$$

2. **Prime Numbers.** Reimplement the prime number sequence using `constexpr` functions. You can follow the outline from the previous exercise, if you like:
   - Create a `constexpr` function `is_prime (int p, int k = p-1)` that checks whether p is not divisible by any number larger than one and smaller than or equal to k (recursive function).
   - Write `constexpr` functions `next_prime(int n)` and `prime(int n)` that return the next prime after n and the n-th prime, respectively.
   - Note that there is a fundamental difference to the previous exercise: the ternary operator wasn't suitable for the template metaprogram, since both "branches" would be instantiated in any case (infinite recursion), but it is actually guaranteed that only one of them will be executed in the end. This means you can use the ternary operator for this version, as mentioned above.
3. **Comparisons.** Compare the implementation effort and code complexity of the template metaprogramming and `constexpr` versions for both sequences. Which version would you prefer? Measure the compilation times of all four codes (Fibonacci and prime numbers, template metaprogramming and `constexpr`) for different inputs. What do you observe? Discuss.

Linked List with Smart Pointers

Revisit the linked list exercise: we used raw pointers back then, but we might just as well use smart pointers instead. The behavior of the linked list class will change depending on the actual choice of smart pointer. Note that one doesn't assign `nullptr` to an invalid smart pointer, or compare against this value to check validity. Instead, invalid smart pointers are simply those that don't currently store a raw pointer and evaluate to `false` when used as a Boolean expression.

1. Try replacing the raw pointers with `std::unique_ptr<Node>`. Is this straight-forward? If not, what exactly is problematic about the chosen interface of the linked list? Remember what happened when the list was copied. What happens / would happen, if unique pointers are used instead?

2. Replace the raw pointers with `std::shared_ptr<Node>`. What happens in this case if we copy the list, and then modify one of the copies? And what happens if one of them or both go out of scope?

3. Turn the shared pointer version of the linked list into a doubly linked list: introduce an additional method `previous` that returns the previous node instead of the next one, and modify the `Node` class accordingly. Note that simply using shared pointers for this would produce a silent memory leak, since the resulting loops of shared pointers would artificially keep the list alive after it goes out of scope. This means you have to use a `std::weak_ptr<Node>` instead.

4. Discuss the benefits and drawbacks of smart pointers for this application. How would you personally implement a doubly linked list, and what would be your design decisions?

Lambda Expressions for Integration

Introduced in C++11, lambda expressions are a very convenient way to define function objects (functors). We will have a second look at numerical integration, so that you can decide for yourself whether using lambdas makes the resulting code more readable or not.

In a previous exercise you had to implement several functors to write a program for numerical integration (quadrature):

- scalar real functions (polynomials and trigonometric functions)
- quadrature rules (trapezoidal rule and Simpson's rule)
- composite rules (only the equidistant one in our case)
- ...

Each of these functors was derived from some abstract base class that defined its interface.

Your task is the modification of the program you already have, so that it uses lambda expressions instead of the classes you created (or to write the program from scratch if you skipped that exercise). Proceed as follows:

1. Remove all abstract base classes. Lambda expressions introduce anonymous auto-generated classes that are not derived from anything[38], so these base classes become irrelevant. The usual replacement for these abstract base classes would be templates, to model the polymorphism at compile-time. However, you need to pass lambdas as arguments to lambdas, i.e., these lambda expressions themselves would need to be templates. This isn't possible in C++11, since a lambda expression defines a functor class, not a functor class template, and the type of function arguments is therefore fixed. C++14 introduces generic lambdas, as we will see, but for now another solution has to be used.

[38] https://en.cppreference.com/w/cpp/language/lambda

2. C++11 provides a class template called `std::function` (in header `<functional>`). This template is an instance of *type erasure*: you can assign any type of function or function object to it, and it will forward the function calls to it, while its type is independent of the exact type of its contents. This means you can capture lambdas in `std::function` objects and ignore the fact that each lambda defines its own type. Inform yourself how this class is used and what its drawbacks are[39].

3. Replace each functor object with an equivalent lambda expression, using `std::function` wherever you have to pass a lambda to another lambda. Note how the `std::function` argument serves as documentation of the expected function interface. Wherever appropriate, capture local variables instead of handing them over as arguments, and make sure that the right capture type is used (by-value vs. by-reference).

4. Run the test problems as described in the previous exercise, and make sure that your new version produces the same results. Use the `time` command to compare the compile time for the two versions, as well as the runtime for a large number of subintervals. Which version is faster, if any, and which is easier to read resp. implement? Discuss.

Variadic Templates: Type-Safe Printing

The concept of variadic functions is nothing new: C has them as well, e.g., the `printf` function that creates formatted output. However, these C-style variadic functions[40] have two drawbacks:

- They need special macros, like `va_start`, `va_arg`, `va_end`, etc., which are not very intuitive to use and incur a certain runtime cost.
- They are not type-safe, since there is no way to know what data types are going to be passed as arguments.

Variadic functions based on the variadic templates introduced in C++11, however, have full access to the type information of their arguments. Use this to write a flexible `print()` function:

1. Start by writing several variants of this function for a single argument, using a combination of, e.g., SFINAE and simple function overloads:
   - Any class that has a `print()` method should be printed by simply calling this method. Create a simple numerical vector class with such a method that prints its components as comma-separated values enclosed by brackets, e.g., `[2.72,3.12,6.59]`, and test your function with it. There is no need to implement any of the usual methods of vector classes, just an appropriate constructor and the `print()` method.

[39] https://en.cppreference.com/w/cpp/utility/functional/function
[40] https://en.cppreference.com/w/cpp/utility/variadic

- Strings (`std::string`) and string literals (`const char*`) should be placed between quotation marks (you can ignore the fact that internal quotation marks in the strings would have to be escaped in practice).
- Floating-point numbers should be printed with six-digit precision in scientific notation. Use the appropriate I/O manipulator[41] and the `precision()` method of the stream object, and don't forget to restore the original precision after printing.
- Any other single argument should simply be printed by passing it to `std::cout` .

2. Write a variadic `print()` function that is able to print an arbitrary number of arguments using the rules above. Base your implementation on the single-argument versions you have already produced, and note that the order of function definitions is important. Test your function with some sequences of mixed arguments. There is no need to specify any types, like you would for `printf` : the compiler can deduce those types from the function argument. Note that this also means that there is no way to accidentally specify the wrong type, of course.

Variadic Templates: Tuples

The lecture discussed a custom implementation of arrays using recursive data types:

- a zero-element array is simply an empty struct
- an $(N+1)$-element array is an N-element array (public inheritance) storing an additional element
- the number of elements `N` and the type of elements `T` are template parameters
- a templatized `entry` method returns the M-th entry, $0 \leq M < N$, of the array

Back then we noted that tuples could also be implemented this way, but only if variadic templates are available for the arbitrarily long list of stored types. With variadic templates having been introduced in the lecture, you are tasked with implementing such a recursive variadic class template that defines tuples.

Proceed as follows:

1. Define a class template `Tuple` that accepts an arbitrary number of types, but *don't* provide an implementation. Every possible case is handled through a specialization, but you will see that you need this general declaration, because otherwise you will not be able to match the template parameter signatures.
2. A tuple containing $N + 1$ variables of up to $N + 1$ different types consists of a variable of the first type and a tuple of the remaining N variables with associated types (again via public inheritance). Create a template specialization for the case $N > 0$, with an appropriate data member, base class specification, and constructor.

[41] https://en.cppreference.com/w/cpp/io/manip

3. Write a method function template `entry<int M>()` that returns the M-th entry ($0 \leq M < N$). Two cases are possible:

   - $M = 0$: The zeroth entry is the data member, so we can simply return that.
   - $M \neq 0$: The requested entry is part of the base class, so we forward the call and return what the base class gives us.

   Use SFINAE to differentiate between these two cases, and handle them separately. Note that we don't actually know the returned type in the second case. Even knowing the return type of the direct base class is not enough, because the actual data member might be in some base class of the base class. One possible solution would be an internal template metaprogram that extracts the correct type. Is there a simpler solution?

4. The class `std::tuple` has two different access methods: via index as above, or via requested type, if the latter is unique across the tuple. Provide this functionality for the custom tuple class, i.e., write a method function template `entry<U>()` that returns the entry of type `U`. There are two possibilities:

   - The data member has type `U` and that type doesn't appear in the base class: simply return the data member as above.
   - The data member doesn't have type `U`: hand the request over to the base class.

   Use SFINAE to differentiate between these two cases, and handle them separately. Note that we don't cover the case where `U` appears more than once explicitly — it's okay if this just results in a compilation error. You will need information about contained types: write an internal class template `struct contains<U>` that exports `true` if the tuple or its base classes contain a data member of type `U`, else `false`. Use normal logic operators (`&&` and `||`) in the SFINAE and internal struct, or the C++17 class templates `std::conjunction` and `std::disjunction` if you want.

5. Provide a base case, which is an empty tuple. Just as with the custom arrays, this is essentially an empty struct, but you will have to provide a base case for the internal `contains` struct.

The complete implementation contains four different `entry` methods, and the SFINAEs make sure that exactly one of them matches in any situation, whether an index is passed as parameter or a type. Note that in contrast to our custom implementation, the class `std::tuple` uses a free function template named `std::get` for access. The main reason is that our version becomes slightly awkward to use within templates, where one has to specify that the method is, indeed, also a template: `t.template entry<2>()` or similar.

Concurrency with Threads

Use threads to implement a parallelized scalar product of two vectors. You may use any vector class: the numerical ones of the lecture, a `std::vector`, or even a plain old C-style

array. Alternatively, you may provide a function template to handle all these separate cases simultaneously.

Create the following four versions, each operating on a subset of the vector components:

1. A version using mutexes and locks, directly adding the occuring products to the result.
2. A second version using mutexes and locks, computing the local scalar product of the indices belonging to the thread, and then adding those local products to the result.
3. A variant of the first version, using atomics instead of mutexes.
4. A variant of the second version, using atomics instead of mutexes.

   The main program should divide any given pair of two vectors into segments of more or less equal size and hand them two a matching number of worker threads. Test your four versions on large vectors, and measure the required time. What happens for larger numbers of threads, or what do you expect would happen, in case the number of parallel threads you can start is very limited?

   Assume for a moment that the number of threads is so large that even the second and the fourth version suffer from congestion (this is a real problem, albeit in the context of message passing on very large super clusters). What could be done to alleviate the problem? You don't need to implement the solution.

   In a real numerical program, the scalar product would be used for subsequent computations. An example is the Conjugate Gradients method, where the expression for the step direction of the scheme contains a scalar product. In a parallelized version of such a program, the threads would not just compute the scalar product, but perform other operations before and after. It is obviously very important to make sure that the scalar product has been fully computed before using the result in other computations.
5. Create a synchronization point (barrier) for the first two versions, e.g., using counters and condition variables or something similar. After this point, have each thread print the scalar product as a stand-in for further computations, and check that all threads report the same value.
6. Inform yourself about memory order models and their consequences, and try to create a similar barrier for the atomics versions, e.g., using atomic counter variables and a Boolean flag that uses `load` and `store`. Print and check the results as above.

C++ Quiz 5

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/219 (variadic function template)

Question 2: https://cppquiz.org/quiz/question/131 (types of initialization)

Question 3: https://cppquiz.org/quiz/question/109 (most vexing parse)

Question 4: https://cppquiz.org/quiz/question/42 (initializer list constructor)

Question 5: https://cppquiz.org/quiz/question/48 (async and futures)

Question 1 revisits variadic function templates. Questions 2 through 4 are about the different kinds of initialization and construction, and certain details that might be surprising. Question 5 is about the futures returned by `std::async`, and how their behavior differs from that of other futures.

Message Passing Interface (MPI)

In this exercise we will train using external libraries and performing parallel computations. Use the *Message Passing Interface* (MPI)[42] to calculate matrix-vector products in parallel.

We will use OpenMPI, but you are free to use any other available MPI software package you like. If you do not have MPI already installed on your machine, you can:

- install it using your package manager (it should be called `openmpi` or similar)
- manually download OpenMPI from https://www.open-mpi.org/software/ompi/v4.1/

In the latter case, extracting and installing is done by the commands:

```
shell$ tar -xvzf openmpi-X.Y.Z.tar.gz
shell$ cd openmpi-X.Y.Z
shell$ mkdir build
shell$ cd build
shell$ ../configure --prefix=/where/to/install
shell$ make all install
```

For more details about the installation process, you can have a look at the provided `INSTALL` file or online at https://www.open-mpi.org/faq/.

If you have never used MPI before, it might be helpful to compile and run a hello world program first, like this C-style example on https://mpitutorial.com/tutorials/mpi-hello-world/.

MPI-based programs have to include a header file called `mpi.h`. They also have to initialize the MPI framework before it can be used, and shut it down before the program finishes. The MPI interface is C code, and may therefore look unfamiliar to you. There are named constants, all uppercase, and functions, with uppercase initial letter:

- Startup/shutdown functions: `MPI_Init`, `MPI_Finalize`
- Communication functions: `MPI_Send`, `MPI_Receive`, `MPI_Alltoall`, `MPI_Barrier`, ...
- Communicators: `MPI_COMM_WORLD`, `MPI_COMM_SELF`

[42] https://en.wikipedia.org/wiki/Message_Passing_Interface

- Data types: `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, ...
- Reduction operations: `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, ...

Here are some resources you might find helpful, an example program, a condensed overview of commands, and the documentation page:

- https://en.wikipedia.org/wiki/Message_Passing_Interface#Example_program
- https://www.boost.org/doc/libs/1_75_0/doc/html/mpi/c_mapping.html
- https://www.mpi-forum.org/

To compile your code use

```
mpic++ -o application application.cc
```

This is a wrapper around the C++ compiler that takes care of MPI-related stuff. All flags you otherwise would pass to your C++ compiler (like warnings or optimization), can be be passed to `mpic++` as well.

To run your code use

```
mpirun -n 4 ./application
```

where the number 4 stands for the number of processes you would like to start. Take care that you don't accidentally crash your system by invoking too many processes.

In thread-based applications, all the threads share the same memory space, and the main issue is mutual exclusion. In applications based on message passing, however, each process has its own memory space. This means values that are needed for some local computation may need to be sent by one process and received by another.

Write one of the following parallel matrix-vector products, assuming an $n \times n$ matrix with n divisible by the number of processos, and the input and result vectors each distributed in chunks of equal size:

1. *Horizontally distributed matrix (whole rows).* The i-th entry of the result vector is on the same process as the i-th row of the matrix, so assembling the output is easy, but the required components of the input vector are mostly non-local and have to be communicated.
2. *Vertically distributed matrix (whole columns).* The j-th entry of the input vector is on the same process as the j-th column of the matrix, so local multiplication is easy, but the components of the output vector have to be accumulated from across all processes.

Which of these two versions is easier to implement? Which, do you think, is more efficient? Choose carefully. Note that there is no need to communicate individual entries when you want to send a whole subvector: the memory layout of `std::vector` is contiguous, and there is even a `data` method that provides access to the underlying raw memory. Make use of that. Receive anything you communicate in a local scope and discard it after using it: in real

applications you often don't have the space to keep all intermediate data around (that's one of the reasons to parallelize, after all).

Message Passing, Part II

Implement the version of parallel matrix-vector multiplication you did not choose in the previous exercise. Determine which of your implementations is faster (for large dimension n), and how the required time scales with the dimension in each version.

Variadic Templates: Matrix Polynomials

In the lecture we discussed the evaluation of polynomials using variadic templates. The recursive function definition allowed using a different type for each of the coefficients, but was restricted to `double` as the type of the variable x and the resulting function value $p(x)$. We would like to make this function more general step by step, until we are able to evaluate matrix polynomials and polynomial matrices.

A *matrix polynomial*[43] is a polynomial where the scalar variable x is replaced with a square matrix A:

$$p(A) = a_0 I + a_1 A + a_2 A^2 + a_3 A^3 + \dots$$

Any square matrix A commutes with itself, of course, so that integer matrix powers are well-defined. The neutral element of multiplication I (the identity matrix) has to be specified explicitly in this case, while it is normally omitted in ordinary, scalar-valued polynomials.

A *polynomial matrix*[44] is a matrix with polynomials as entries, which can also be written as a (scalar) polynomial with matrices $A_0, A_1, \dots$ as coefficients:

$$p(x) = A_0 + A_1 x + A_2 x^2 + A_3 x^3 + \dots$$

These matrices don't have to be square, but they need to be of compatible dimensions, i.e., each an $n \times m$ matrix, with the same n and m across all the matrices.

Proceed as follows:

1. Take the function definition of the lecture and introduce a template parameter `X` for the type of the first argument x. The resulting return type is now a function of `X`, `T`, and the remaining types `Ts...`, and is not known a priori: it is certainly not simply `X`, because a polynomial with integer argument and real coefficients has to produce real function values. How can you specify the correct return type?
2. The function should now be able to evaluate polynomial matrices. Test your implementation with $n \times n$ matrices of small size, say $n = 3$, and make sure that the results are what you expect. Each entry of the resulting matrix is the evaluation of a separate ordinary, scalar-valued polynomial, so this should be straight-forward.

[43] https://en.wikipedia.org/wiki/Matrix_polynomial
[44] https://en.wikipedia.org/wiki/Polynomial_matrix

3. Expand the implementation so that it can also handle matrix polynomials. In this case the function has to compute matrix powers, and it becomes especially important to use Horner's method. Handling matrices as arguments requires the explicit creation of identity matrices, see the definition above. How can you make sure that your function remains usable for ordinary polynomials with real x? Use a matrix class where the dimensions are defined through template parameters, so that you don't have to consider the correct size when creating identity matrices.

4. Test your implementation on matrix polynomials to make sure that everything works correctly. How does the computation time depend on the size of the matrices, and how on the number of coefficients? Create a variant that uses the less efficient evaluation strategy from the lecture. How does that influence the time that is needed for the computations?

Custom Range-Based Loops

Create your own range-based `for` loops for numerical vectors. Take one of our vector classes, and write code that provides the additional functionality. The vector class itself should not be modified (treat it as if it was from some external library). Instead, use the public interface of the class.

1. Write an iterator class with the following functionality:

   - A constructor taking a vector object. The resulting iterator belongs to this object and iterates over it.
   - Pre-increment and pre-decrement operators (`operator++` and `operator--`). These shift the iterator by one position.
   - A dereference operator `operator*`, which returns the entry the iterator refers to.
   - A not-equal-to operator `operator!=`, for comparison against the end of the vector.

   The iterator has to store its position, the current index is a natural fit. You also need to encode iterators with invalid state, e.g., by using the length of the vector as stored index.

2. Provide free `begin` and `end` functions that receive a vector object and return an iterator object, pointing to the first entry for the former, and having an invalid state for the latter.

3. Test your implementation to make sure that it is working, e.g., by printing the vector components in order. Then, provide a `const` version of the class and functions, so that `const` vectors work as well. Use the appropriate standard version of the range-based for loop in each case.

C++ Quiz 6

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/126 (unqualified lookup)

Question 2: https://cppquiz.org/quiz/question/184 (method shadowing)

Question 3: https://cppquiz.org/quiz/question/18 (private virtual function)

Question 4: https://cppquiz.org/quiz/question/264 (consequences of defaulted constr.)

Question 5: https://cppquiz.org/quiz/question/112 (default arg of virtual function)

Question 6: https://cppquiz.org/quiz/question/250 (overloaded variadic functions)

Question 7: https://cppquiz.org/quiz/question/109 (deduced and nondeduced context)

These are questions that shine some light on the hidden depths of C++. Neither the questions nor their answers are in any way relevant for the exam, and the correct answer may range from surprising to confusing. This means you don't have to look at them, unless you are interested, of course. Here be dragons.