Object-Oriented Programming for Scientific Computing

Dr. Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

Winter Semester 2020/21

Prerequisites and Objectives

Prerequisites

- Familiarity with at least one programming language
- At least procedural programming in C / C++
- Willingness to program in practice

Objectives

- Improved programming skills
- Introduction of modern programming techniques
- Strong focus on topics of relevance to Scientific Computing

Course Outline

General course outline:

Short recapitulation of basics	data types, functions, templates, classes, etc.
The C++ Standard Library	input/output, containers, iterators, algorithms, exceptions, etc.
Advanced topics	dynamic polymorphism, RAII, template meta programming, static polymorphism, SFINAE, etc.
C++11 features	smart pointers, lambda expressions, variadic templates, random numbers, chrono library, threads, etc.
C++14 features	generic lambdas, variable templates
C++17 features	guaranteed copy elision, structured bindings, fold expressions
Upcoming C++20 features	modules, concepts, ranges, coroutines

The baseline for the lecture is the C++11 standard, with changes due to C++14 and C++17 taken into account. The new features of the upcoming C++20 standard will be introduced where appropriate.

Lecture Website

Link: conan.iwr.uni-heidelberg.de/teaching/oopfsc_ws2020/

Content:

- Link to draft of official standard
- Literature recommendations and quick reference
- GitLab instructions (for exercises)
- Lecture slides (updated after each lecture)
- Exercise sheets (same as above)

MUESLI links (also listed on the lecture website):

- Registration: muesli.mathi.uni-heidelberg.de/user/register
- Course page: muesli.mathi.uni-heidelberg.de/lecture/view/1288

Exercises and Exam

Exercises:

- Will be completely online this semester (see homepage)
- Assistant: Stefan Meggendorfer ⟨stefan.meggendorfer@iwr.uni-heidelberg.de⟩
- New exercises every week: on the website after the lecture
- To be handed in right before the lecture on Wednesday
- Geared towards g++ and Linux
- Correction, grading etc. depends on course size

Exam:

- Probably in the last week of the semester (TBA)
- Mandatory for all participants
- 50% of the points in the exercises required for admission, plus active participation in discussion

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)

Why C++?

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

Why C++?

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 C++14 to support some architectures, see for example DUNE [1].

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

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

Fundamental Concepts

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.

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.

Fundamental Types

`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.

Fundamental Types

`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;};
```

Introducing New Types

`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[2], preferably from the PDELab subproject[3].

```
// 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] https://www.dune-project.org

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

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

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`).

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[4], 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.

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

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:

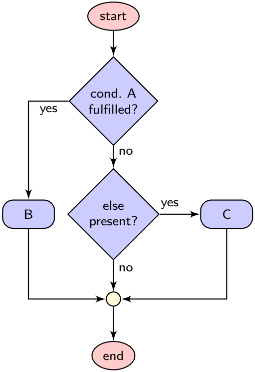
```cpp
for (int i = 0; i < 100; i++)
{
  if (i % 2 == 0)
    continue;
  if (i > 10)
    break;
  // prints 1, 3, 5, 7, 9
  std::cout << i << std::endl;
}
```

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).

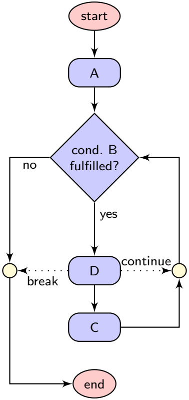
Diagram: Loops and Branches

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;
```

and

```
do A while (B);
```

is equivalent to

```
A; while (B) A;
```

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:

```cpp
// 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[5]:

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

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

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

Call-by-Reference

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):

```cpp
// 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[6]. 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.

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

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).

```cpp
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 `struct PairOfInts {int a; int b;};` . 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).

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:

```cpp
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 `*`:

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

Templates

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:

```cpp
// 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:

```cpp
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[7]: 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.

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

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;
    };
  };
```

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();
```

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).

Constructors

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[8]. Assume we have defined

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

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
```

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

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:

```
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;
  }
};
```

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).

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.

Default Methods

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[9] 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.

[9] 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)

The Singleton Pattern

Realization of the singleton pattern in C++:

```
class Singleton
{
  public:
    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).

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 maxtrix 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

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[10].

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`	—	—	—

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

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.

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[11].

Covariance of return types The method may return a subtype of the original return type[12].

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

[11] not available in C++

[12] in C++ only in the context of → dynamic polymorphism

The Liskov Substitution Principle

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.
- 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.

```cpp
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

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.

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.

```cpp
#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
```

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.

The Standard Library

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`.

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`	specialization for strings as data sources/sinks

Input / Output

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.

Input / Output

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[13]), 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);
};
```

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

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<<"[";
  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;
```

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).

Sequential Containers

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.

Associative Containers

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[14] $\mathcal{O}(1)$, 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[15], and complexity guarantees are linked from there.

[14] averaged over many method calls

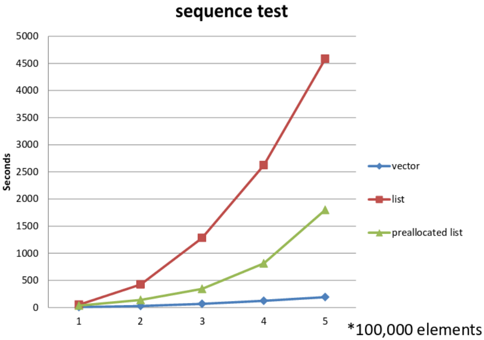
[15] https://en.cppreference.com/w/cpp/container

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
- 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`?

Sequence Test: Vector vs. List



Source: B. Stroustrup

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

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.).

Iterators

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.

Iterators

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`[16]

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.

[16] mainly because their order is arbitrary anyways

Algorithms

The Standard Library provides a large number of algorithms[17] 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

[17] full list: `https://en.cppreference.com/w/cpp/algorithm`

Algorithms

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`:

```cpp
// 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;
    }
  }
```

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

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.

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. . .

Exceptions

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`.

```cpp
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;
}
```

Exceptions

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.

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)

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[18], 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*>`*?*

[18] and variable templates since C++14

Template Specialization

```cpp
// 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

Template Specialization

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++.

Template Specialization

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[19]:

```
// (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[20]! *Why?*

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

[19] see http://www.gotw.ca/publications/mill17.htm

[20] interactive version: link to godbolt.org

Dimov/Abrahams Example

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 → `constexpr if` 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”

Template Specialization (cont.)

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);
  }
```

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.

Classes and Templates: Interactions

```
#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[a]
- `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)

[a] link to godbolt.org

Classes and Templates: Interactions

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.

Classes and Templates: Interactions

```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.

Classes and Templates: Interactions

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;
    ...
  };
```

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

- If acquiring `r_k` fails, `r_1,...,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,...,r_(k-1)`?

Multiple Resource Allocation

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.

Example Implementation

```
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)

Example Implementation

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)[21], 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

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

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:

```cpp
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[a]

[a] Link: gitlab.dune-project.org/staging/dune-typetree

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[22] (only depends on the object itself, not the method arguments).

[22] 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
```

Polymorphic Types

- 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 (re-)declare 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.

Implementation Detail: vtables

```
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

Copying Polymorphic Types

```
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

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.

Example from DUNE

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

// 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.

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.

Static Polymorphism

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.

Static Polymorphism

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

struct Derived : public Base<Derived>
{ // derived class: provide implementation
  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); }
};
```

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

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.

Picking Numerical Solvers

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[23], 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

[23] e.g., some precomputed matrix decomposition, a specialized spectral method, etc.

Solving Linear Systems

Traits class for existance of method, adapted from Jean Guegant's example[24]:

```
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))};
  };
```

[24] https://jguegant.github.io/blogs/tech/sfinae-introduction.html

Solving Linear Systems

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

```
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)

SFINAE (cont.)

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):

```cpp
//! 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.");
  ...
};
```

Example from DUNE

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

```cpp
//! 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)
  { }
};
```

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

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.

Automatic Type Deduction

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;}
```

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;
  }
```

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

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[25].

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.

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

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))
  {}

  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)

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

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 Pointers

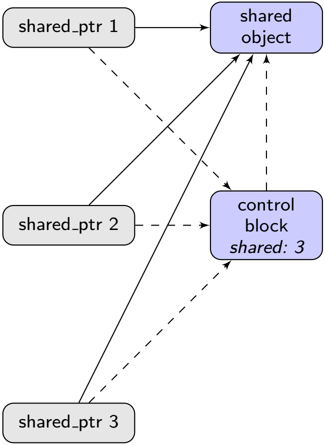
Shared pointer provide the same methods as unique pointers, and additionally have a `use_count()` method that reports the total number of pointers sharing the resource. There is also again an `operator[]` for the array case, but only from C++17 onwards.

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).

Shared Pointers



- 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

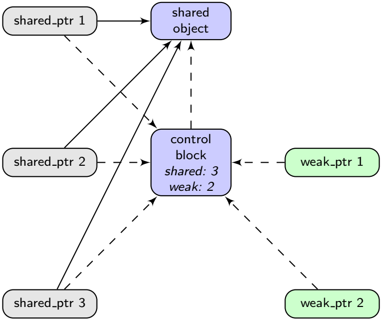
Weak Pointers

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
  ...
}
```

Weak Pointers



- 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 `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.

Smart Pointer Use Cases

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.

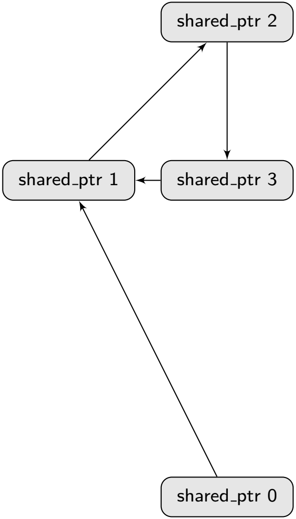
Pitfalls When Using Shared Pointers

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.

Pitfalls When Using Shared Pointers



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

Pitfalls When Using Shared Pointers

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();}
}
```

Pitfalls When Using Shared Pointers

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.

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).

Lambda Expressions (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
```

Lambda Expressions (Closures)

The capture list may:

- be empty, thereby defining an anonymous function that is not a closure (i.e., nothing is captured[26])
- 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 {...};
```

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

Lambda Expressions (Closures)

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()` .

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);
                                 }));
  }
```

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`.

Variadic Templates

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.

Variadic Templates

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

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

```
// 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);
```

Variadic Templates

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...);
  }
```

Variadic Templates

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;
```

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.

Concurrency and Parallel Computing

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

Concurrency with Threads

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.

Concurrency with Threads

Thread-based programs have a special thread-local namespace named `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.

Concurrency with Threads

```cpp
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.

Mutexes and Locks

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 and Locks

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:

```cpp
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.

Mutexes and Locks

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.

Atomics

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.

Atomics

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[27].

[27] See https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange

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:

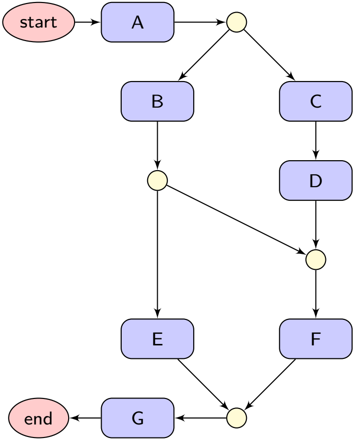
- there is extensive communication and exchange of data between the threads
- the threads operate collaboratively, maybe even executing the same code on different sets of data, exchanging intermediate results

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.

Concurrency with Tasks



- 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

Promises and Futures

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[28].

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.

[28] This makes it possible to return large objects from tasks using move semantics.

Promises and Futures

A matrix-vector product, handed over to a new thread:

```cpp
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.

```
// 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

Asynchronous Execution

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[29].

[29] A void future `std::future<void>` can be used in this case.

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[30]

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!

[30] See https://en.cppreference.com/w/cpp/language/aggregate_initialization for exact specification.

Types of Initialization

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.

Types of Initialization

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.

```
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 Loops

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.

Range-Based Loops

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.

Range-Based Loops

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'})
{ ... }
```

Template Aliases

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.

```
// classical typedefs
using Number = int; // replaces ``typedef int Number''

// partial specialization or instantiation
template<typename T>
  using Vector3D = Vector<T,3>; // defines ``Vector3D<T>''

// template alias
template<typename U, typename V>
  using Matrix = Long::Namespace::SpecialMatrix<U,V>;

// variadic template alias
// (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, e.g., `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

Time measurements using the library tend to be a bit verbose, because durations have to be explicitly cast to time units:

```cpp
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

Random Number Generation

```
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

```cpp
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, …).

Custom Literals

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:

```cpp
// 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[31].

[31] See https://en.cppreference.com/w/cpp/regex.

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

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.

Improvements to Constant Expressions

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.

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 `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.

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.

```cpp
// 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;
```

Variable Templates

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.

Return Type Deduction

In many cases the trailing return type of lambda expressions is optional, and can be deduced from the return statement instead:

```cpp
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).

Return Type Deduction

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};
}
```

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 `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.