C++20

The Complete Guide

Nicolai M. Josuttis

C++20 - The Complete Guide

First Edition

Nicolai M. Josuttis

This version was published on **2021-12-31**.

This book was typeset by Nicolai M. Josuttis using the LaTeX document processing system.

This book is for sale at **http://leanpub.com/cpp20**.

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book, and build traction once you do.

Contents

Preface

C++20 is the next evolution in modern C++ programming, which is already (partially) supported by the latest version of gcc, clang, and Visual C++. The move to C++20 is at least as big a step as the move to C++11. C++20 contains a significant number of new language features and libraries that again will change the way we program in C++. This applies to both application programmers and programmers who provide foundation libraries.

An Experiment

This book is an experiment in two ways:

- I am writing an in-depth book covering a complex new features provided by different authors and C++ working groups. However, I can ask questions and I do.
- I am publishing the book myself on Leanpub and for printing on demand. That is, this book is written step by step and I will publish new versions as soon there is a significant improvement that makes the publication of a new version worthwhile.

The good thing is:

- You get the view of the language features from an experienced application programmer—somebody who feels the pain a feature might cause and asks the relevant questions to be able to motivate and explain the design and its consequences for programming in practice.
- You can benefit from my experience with C++20 while I am still learning and writing.
- This book and all readers can benefit from your early feedback.

This means that you are also part of the experiment. So help me out: give feedback about flaws, errors, features that are not explained well, or gaps, so that we all can benefit from these improvements.

Versions of This Book

Because this book is written incrementally, the following is a history of the major updates (newest first):

- **2021-12-31**: Describe where `typename` is no longer required for type members of template parameters
- **2021-12-30**: Describe new bit operations (including `bit_cast<>()`)

- **2021-12-29**: Describe all improvements for string types (including `std::u8string` and using strings at compile time)
- **2021-12-28**: Describe the `unseq` execution policy for algorithms
- **2021-12-25**: Describe all other missing lambda features
- **2021-12-11**: Describe `consteval` lambdas
- **2021-12-06**: Describe `std::endian`
- **2021-12-06**: Describe synchronized output streams
- **2021-12-04**: Describe header `<version>`
- **2021-11-21**: Describe compile-time use of vectors and `constexpr` extensions
- **2021-10-25**: Describe designated initializers
- **2021-10-14**: Describe severe compatibility issues with `operator==`
- **2021-10-14**: Describe mathematical constants
- **2021-10-12**: Describe `constinit`, `consteval`, and `std::is_constant_evaluated()`
- **2021-10-07**: Describe `char8_t` for UTF-8 (and its compatibility issues)
- **2021-10-02**: Clarify over which `const` views you cannot iterate
- **2021-10-01**: Details about the formatting library
- **2021-09-21**: Updates and fixes according to several reviews
- **2021-09-20**: First description of features for modules
- **2021-09-11**: `const` issues of ranges and views
- **2021-08-30**: All concepts documented
- **2021-08-30**: All concepts documented
- **2021-08-28**: Document all new type traits
- **2021-08-27**: Document iterator utility functions for ranges.
- **2021-08-26**: Document `lazy_split_view<>` added to C++20 with a defect.
- **2021-08-25**: Document all missing views of C++20 and `all_t<>`
- **2021-08-20**: Document all new iterator types (`counted_iterator`, `common_iterator`, `default_sentinel`, `unreachable_sentinel`)
- **2021-08-20**: Document all primary range adaptors (`counted()`, `all()`, and `common()`)
- **2021-08-19**: Document `common_view`
- **2021-08-16**: Document how to deal with semantic constraints of concepts
- **2021-08-15**: Document `type_identity`
- **2021-07-31**: Document `empty_view`, `ref_view`, and `view_interface`
- **2021-07-29**: Generic code for range has to use universal/forwarding references
- **2021-07-28**: Document `iota_view` and `unreached_sentinel`
- **2021-07-27**: Extend and reorganize chapters about ranges and views
- **2021-07-17**: Document `subrange`
- **2021-07-15**: Fix wrong website for the book
- **2021-07-09**: Document `std::source_location`
- **2021-06-29**: The initial published version of the book. The following features are more or less completely described or provide at least a full conceptual introduction:

- Comparisons and the spaceship operator `<=>`
- Generic functions
- Concepts and requirements (details open)
- Ranges and views (details open)
- Spans
- Non-type template parameter extensions
- Formatted output (overview only)
- Dates and time zones for `<chrono>`
- Coroutines (first examples)
- `std::thread` and stop tokens
- New concurrency features

Acknowledgments

First of all, I would like to thank you, the C++ community, for making this book possible. The incredible design of all the features of C++20, the helpful feedback, and your curiosity are the basis for the evolution of a successful language. In particular, thanks for all the issues you told me about and explained and for the feedback you gave.

I would especially like to thank everyone who reviewed drafts of this book or corresponding slides and provided valuable feedback and clarification. These reviews increased the quality of the book significantly, again proving that good things are usually the result of many people. Therefore, so far (this list is still growing) huge thanks to Carlos Buchart, Javier Estrada, Howard Hinnant, Paul Ranson, Thomas Symalla, Steve Vinoski and Victor Zverovich.

******* This chapter/section is at work *******

In addition, I would like to thank everyone on the C++ standards committee. In addition to all the work involved in adding new language and library features, these experts spent many, many hours explaining and discussing their work with me, and they did so with patience and enthusiasm. Special thanks here go to Howard Hinnant, Tomasz Kaminski, Arthur O'Dwyer, Peter Sommerlad, Tim Song, Barry Revzin, Ville Voutilainen, and Jonathan Wakely.

Special thanks go to the LaTeX community for a great text system and to Frank Mittelbach for solving my LaTeX issues (it was almost always my fault).

This page is intentionally left blank

About This Book

This book will present all the new language and library features of C++20. It will cover the motivation and context of each new feature with examples and background information.

As usual for my books, the focus lies on the application of the new features in practice and the book will demonstrate how features impact day-to-day programming and how you can benefit from them in projects. This applies to both application programmers and programmers who provide generic frameworks and foundation libraries.

What You Should Know Before Reading This Book

To get the most from this book, you should already be familiar with C++, ideally C++11, C++14, and/or C++17. You should be familiar with the concepts of classes and references in general, and you should be able to write C++ programs using components such as IOStreams and containers from the C++ standard library. You should also be familiar with the basic features of "Modern C++," such as `auto` or the range-based `for` loop.

However, you do not have to be an expert. My goal is to make the content understandable for the average C++ programmer who does not necessarily know all the details of the latest features. I will discuss basic features and review more subtle issues as the need arises.

This ensures that the text is accessible to experts and intermediate programmers alike.

Overall Structure of the Book

This book covers *all* changes to C++ introduced with C++20. This applies to both language and library features as well as both features that affect day-to-day application programming and features for the sophisticated implementation of (foundation) libraries. However, the more general cases and examples usually come first.

The different chapters are grouped, but the grouping has no deeper didactic reasoning other than that it makes sense to first introduce language features because they might be used by the following library features. In principle, you can read the chapters in any order. If features from different chapters are combined, there are corresponding cross-references.

The book therefore contains the following parts:

******* This chapter/section is at work *******

How to Read This Book

Do not be afraid by the number of pages in this book. As always with C++, things can become pretty complicated when you look into details. For a basic understanding of the new features, the introducing motivations and examples are often sufficient to understand the basic idea of a feature.

In my experience, the best way to learn something new is to look at examples. Therefore, you will find a lot of examples throughout the book. Some are just a few lines of code illustrating an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples will be introduced by a C++ comment describing the file containing the program code. You can find these files on the website for this book at `http://www.cppstd20.com`.

The Way I Implement

Note the following hints about the way I write code and comments.

Initializations

I usually use the modern form of initialization (introduced in C++11 as *uniform initialization*) with curly braces:

```
int i{42};
std::string s{"hello"};
```

This form of initialization, which is called *brace initialization*, has the following advantages:

- It can be used with fundamental types, class types, aggregates, enumeration types, and `auto`
- It can be used to initialize containers with multiple values
- It can detect narrowing errors (e.g., initialization of an `int` by a floating-point value)
- It cannot be confused with function declarations or calls

If the braces are empty, the default constructors of (sub)objects are called and fundamental data types are guaranteed to be initialized with `0`/`false`/`nullptr`.

Error Terminology

I often talk about programming errors. If there is no special hint, the term *error* or a comment such as

```
...     // ERROR
```

means a compile-time error. The corresponding code should not compile (with a conforming compiler).

If I use the term *runtime error*, the program might compile but not behave correctly or result in undefined behavior (thus, it might or might not do what is expected).

Code Simplifications

I try to explain all features with helpful examples. However, to concentrate on the key aspects to be taught, I might often skip other details that should be part of the code.

- Most of the time I use an ellipsis ("...") to signal additional code that is missing. Note that I do not use code font here. If you see an ellipsis with code font, code must have these three dots as a language feature (such as for "`typename...`").
- In header files I usually skip the preprocessor guards. All header files should have something like the following:

```
#ifndef MYFILE_HPP
#define MYFILE_HPP
...
#endif  // MYFILE_HPP
```

  So, please beware and fix the code when using these header files in your projects.

The C++ Standards

C++ has different versions defined by different C++ standards.

The original C++ standard was published in 1998 and was subsequently amended by a *technical corrigendum* in 2003, which provided minor corrections and clarifications to the original standard. This "old C++ standard" is known as C++98 or C++03.

The world of "Modern C++" began with C++11 and was extended with C++14 and C++17. The international C++ standards committee now aims to issue a new standard every three years. Clearly, that leaves less time for massive additions, but it brings the changes to the broader programming community more quickly. The development of larger features, therefore, takes time and might cover multiple standards.

C++20 is now the beginning of the next "Even more Modern C++" evolution. Again, several ways to program will probably change. However, as usual, compilers need some time to provide the latest language features. At the time of writing this book, C++20 is already at least partially supported by major compilers. However, as usual, compilers differ greatly in their support of new different language features. Some will compile most or even all of the code in this book, while others may only be able to handle a significant subset. I expect that this problem will soon be resolved as programmers everywhere demand standard support from their vendors.

Example Code and Additional Information

You can access all example programs and find more information about this book from its website, which has the following URL:

http://www.cppstd20.com

Feedback

I welcome your constructive input—both negative and positive. I have worked very hard to bring you what I hope you will find to be an excellent book. However, at some point I had to stop writing, reviewing, and tweaking to "release the new revision." You may therefore find errors, inconsistencies, presentations that could be improved, or topics that are missing altogether. Your feedback gives me a chance to fix these issues, inform all readers about the changes through the book's website, and improve any subsequent revisions or editions.

The best way to reach me is by email. You will find the email address at the website for this book:

`http://www.cppstd20.com`

If you use the ebook, you might want to ensure to have the latest version of this book available (remember it is written and published incrementally). You should also check the book's website for the currently known errata before submitting reports. In any case, refer to the publishing date of this version when giving feedback. The current publishing date is **2021-12-31** (you can also find it on page ii, the page directly after the cover).

Many thanks.

Chapter 1

Comparisons and Operator <=>

C++20 simplifies the definition of comparisons for user-defined types and introduces better ways to deal with them. For this purpose, the new operator <=> (also called the *spaceship* operator) was introduced.

This chapter introduces the how to define and deal with comparisons since C++20 using these new features.

1.1 Motivation for Operator<=>

Let us first motivate the new way comparisons are handled since C++20 and the new operator <=>.

1.1.1 Defining Comparison Operators Before C++20

Before C++20 you had to define six operators for a type to provide full support for all possible comparisons of its objects.

For example, if you wanted to compare objects of a type `Value` (having an integral ID), you had to implement the following:

```
class Value {
 private:
  long id;
  ...
 public:
  ...
  // equality operators:
  bool operator== (const Value& rhs) const {
    return id == rhs.id;       // basic check for equality
  }
  bool operator!= (const Value& rhs) const {
    return !(*this == rhs);    // derived check
  }
```

```
  // relational operators:
  bool operator< (const Value& rhs)  const {
    return id < rhs.id;        // basic check for ordering
  }
  bool operator<= (const Value& rhs) const {
    return !(rhs < *this);     // derived check
  }
  bool operator> (const Value& rhs)  const {
    return rhs < *this;        // derived check
  }
  bool operator>= (const Value& rhs) const {
    return !(*this < rhs);     // derived check
  }
};
```

This enables you to call any of the six comparison operators for a `Value` (the object the operator is defined for) with another `Value` (passed as parameter `rhs`). For example:

```
Value v1, v2;
... ;
if (v1 <= v2) {    // calls v1.operator<=(v2)
  ...
}
```

The operators might also be called indirectly (e.g., by calling `sort()`):

```
std::vector<Value> coll;
... ;
std::sort(coll.begin(), coll.end());   // uses operator < to sort
```

Since C++20 , the call might alternatively use ranges:

```
std::ranges::sort(coll);               // uses operator < to sort
```

The problem is that even though most of the operators are defined in terms of others (they are all based on either operator == or operator <), the definitions are tedious and they add a lot of visual clutter.

In addition, for a well implemented type, you might need more:

- Declare the operators with `noexcept` if they cannot throw
- Declare the operators with `constexpr` if they can be used at compile time
- Declare the operators as "hidden friends" (declare them with `friend` inside the class structure so that both operands become parameters and support implicit type conversions) if the constructors are not `explicit`
- Declare the operators with `[[nodiscard]]` to warn if the return value is not used

For example:

lang/valueold.hpp

```
class Value {
 private:
  long id;
  ...
```

```
public:
 constexpr Value(long i) noexcept  // supports implicit type conversion
  : id{i} {
 }
 ...
 // equality operators:
 [[nodiscard]] friend constexpr
 bool operator== (const Value& lhs, const Value& rhs) noexcept {
   return lhs.id == rhs.id;       // basic check for equality
 }
 [[nodiscard]] friend constexpr
 bool operator!= (const Value& lhs, const Value& rhs) noexcept {
   return !(lhs == rhs);          // derived check for inequality
 }

 // relational operators:
 [[nodiscard]] friend constexpr
 bool operator< (const Value& lhs, const Value& rhs) noexcept {
   return lhs.id < rhs.id;        // basic check for ordering
 }
 [[nodiscard]] friend constexpr
 bool operator<= (const Value& lhs, const Value& rhs) noexcept {
   return !(rhs < lhs);           // derived check
 }
 [[nodiscard]] friend constexpr
 bool operator> (const Value& lhs, const Value& rhs) noexcept {
   return rhs < lhs;              // derived check
 }
 [[nodiscard]] friend constexpr
 bool operator>= (const Value& lhs, const Value& rhs) noexcept {
   return !(lhs < rhs);           // derived check
 }
};
```

1.1.2 Defining Comparison Operators Since C++20

Since C++20, a couple of things regarding comparison operators changed.

Operator == Implies Operator !=

To check for equality it is now enough to define operator `==`.

The reason is that if for an expression `a!=b` no implementation can be found, the compiler *rewrites* the expressions and looks for `!(a==b)`. If that does not work, the compiler also tries to exchange the order of the operands, so that it also tries `!(b==a)`:

```
a != b   // tries: a!=b, !(a==b), and !(b==a)
```

Therefore, for `a` of type `TypeA` and `b` of `TypeB`, the compiler will be able to compile

```
a != b
```

if there is

- a freestanding `operator!=(TypeA, TypeB)`
- a freestanding `operator==(TypeA, TypeB)`
- a freestanding `operator==(TypeB, TypeA)`
- a member function `TypeA::operator!=(TypeB)`
- a member function `TypeA::operator==(TypeB)`
- a member function `TypeB::operator==(TypeA)`

Directly calling a defined operator `!=` is preferred (but the order of the types has to fit). Reordering the operands has the lowest priority. Having both a freestanding and a member function is an ambiguity error.

Thus, with

```
bool operator==(const TypeA&, const TypeB&);
```

or

```
class TypeA {
 public:
  ...
  bool operator==(const TypeB&) const;
};
```

the compiler will be able to compile:

```
MyType a;
MyType b;
...
a == b;   // OK: fits perfectly
b == a;   // OK, rewritten as: a == b
a != b;   // OK, rewritten as: !(a == b)
b != a;   // OK, rewritten as: !(a == b)
```

Note that thanks to rewriting, implicit type conversions for the first operand are also possible when rewriting converts it to become the parameter of the defined member function.

See the example `sentinel1.cpp` for how to benefit from this feature by only defining a member operator == for an iterator for which usually `!=` with a different order or operand is called.

Operator <=>

There is no equivalent rule that for all relational operators it is enough to have operator < defined. However, you only have to define the new operator <=>.

In fact, the following is enough to enable programmers to use all possible comparators:

lang/value20.hpp

```
#include <compare>
```

```
class Value {
 private:
  long id;
  ...
 public:
  constexpr Value(long i) noexcept
   : id{i} {
  }
  ...
  // enable use of all equality and relational operators:
  auto operator<=> (const Value& rhs) const  = default;
};
```

In general, operator == cares for the equality operators, while operator <=> cares for the relational operators. However, by declaring an `operator<=>` with `=default`, we use a special rule that a ***defaulted*** member `operator<=>`:

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const  = default;
};
```

generates a corresponding member `operator==`, so that we effectively get:

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const  = default;
  auto operator== (const Value& rhs) const  = default;  // implicitly generated
};
```

The effect is that both operators use their default implementation, which compares objects member by member. This means that the order of the members in the class matters.

Thus, with

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const  = default;
};
```

we get all we need to be able to use all six comparison operators.

In addition, even when declaring the operator as a member function, the generated operators

- are `noexcept` if comparing the members never throws
- are `constexpr` if comparing the members is possible at compile time
- thanks to *rewriting*, implicit type conversions for the first operand are also supported (although it is not passed as a parameter) if a corresponding implicit type conversion is defined
- may warn if the result of a comparison is not used (however, this is up to the compiler)

This reflects that in general `operator==` and `operator<=>` care for different but related things:

- `operator==` defines equality and can be used by the *equality operators* `==` and `!=`.
- `operator<=>` defines the ordering and can be used by the *relational operators* `<`, `<=`, `>`, and `>=`.

To have more control over the generated comparison operators, you can define `operator==` and `operator<=>` yourself. For example:

lang/value20def.hpp

```
#include <compare>

class Value {
 private:
  long id;
  ...
 public:
  constexpr Value(long i) noexcept
   : id{i} {
  }
  ...
  // for equality operators:
  bool operator== (const Value& rhs) const {
    return id == rhs.id;       // defines equality (== and !=)
  }
  // for relational operators:
  auto operator<=> (const Value& rhs) const {
    return id <=> rhs.id;      // defines ordering (<, <=, >, and >=)
  }
};
```

That way you can specify which members in which order matter or implement special behavior.

The way these basic operators work is that if an expression uses one of the comparison operators and does not find a matching direct definition, the expression gets *rewritten* so that it can use these operators. Corresponding to rewriting calls of equality operators, rewriting might also change the order of relational operands, which might enable implicit type conversion for the first operand. For example, if

```
x <= y
```

does not find a matching definition of `operator<=`, it might be rewritten as

```
(x <=> y) <= 0
```

or even

```
0 <= (y <=> x)
```

As you can see by this rewriting, the new `operator<=>` performs a three-way comparison, which yields a value you can compare with `0`:

- If the value of `x<=>y` compares equal to `0`, `x` and `y` are equal or equivalent.
- If the value of `x<=>y` compares less than `0`, `x` is less than `y`.
- If the value of `x<=>y` compares greater than `0`, `x` is greater than `y`.

However, note that the return type of `operator<=>` is not an integral value. The return type is a type that signals the *comparison category*, which could be *strong ordering*, *weak ordering*, or *partial ordering*. These types support the comparison with `0` to deal with the result.

1.2 Defining and Using Comparisons

The following sections will explain the details of the handling of the comparison operators since C++20.

1.2.1 Using Operator<=>

Operator `<=>` is a new binary operator. It is defined for all fundamental data types, for which the relational operators are defined. As usual, it can be user-defined as `operator<=>()`.

Operator `<=>` has precedence over all other comparison operators so that you need parentheses to use it in an output statement but not to compare its result with another value:

```
std::cout << (0 < x <=> y) << '\n';  // calls 0 < (x <=> y)
```

Please note that you have to include a specific header file to deal with the result of operator `<=>`:

```
#include <compare>
```

However, most header files for standard types (strings, containers, `<utility>`) include this header anyway. To call the operator on values or types that do not require this header, you have to include `<compare>`. For example:

```
#include <compare>    // for calling <=>

auto x = 3 <=> 4;     // does not compile without header <compare>
```

1.2.2 Comparison Category Types

The new operator `<=>` does not return a Boolean value. Instead, it acts similar to three-way-comparisons yielding a negative value to signal *less*, a positive value to signal *greater*, and `0` to signal *equal* or *equivalent*. This behavior is similar to the return value of the C function `strcmp()`; however, there is an important difference: the return value is not an integral value. Instead, the C++ standard library provides three possible return types, which reflect the category of the comparison.

Comparison Categories

When comparing two values to put them in an order, we have different categories of behavior that could happen:

- With **strong ordering** (also called *total ordering*) any value of a given type is either *less* than or *equal* to or *greater* than any other value of this type (including itself).

  Typical examples of this category are integral values or common string types. A string `s1` is either less or equal or greater than a string `s2`.

  If a value is neither less nor greater is has to be equal. If you have multiple objects you can sort them in ascending or descending order (with equal values having any order among each other).

- With **weak ordering** any value of a given type is either *less* than or *equivalent* to or *greater* than any other value of this type (including itself). However, equivalent values do not have to be equal (have the same value).

  A typical example of this category is a type for case-insensitive strings. A string `"hello"` is less than `"hello1"` and greater than `"hell"`. However, `"hello"` is equivalent to `"HELLO"` although these two strings are not equal.

  If a value is neither less nor greater is has to be equivalent. If you have multiple objects you can sort them in ascending or descending order (with equivalent values having any order among each other).
- With **partial ordering** any value of a given type *could* either be *less* than or *equivalent* to or *greater* than any other value of this type (including itself). However, it could also happen that you cannot specify a specific order between two values.

  A typical example of this category is a floating-point type, because they might have the special value NaN ("not a number"). Any comparison with NaN yields `false`. So in this case a comparison might yield that two values are *unordered* and the comparison operator might return one of four values.

  If you have multiple objects you might not be able to sort them in ascending or descending order (unless you ensure that values that cannot be ordered are not there).

Comparison Categories Types in the Standard Library

For the different comparison categories, the C++20 standard introduces the following types:

- `std::`**`strong_ordering`** with the values:
  - `std::`**`strong_ordering::less`**
  - `std::`**`strong_ordering::equal`**
    (also available as `std::`**`strong_ordering::equivalent`**)
  - `std::`**`strong_ordering::greater`**
- `std::`**`weak_ordering`** with the values:
  - `std::`**`weak_ordering::less`**
  - `std::`**`weak_ordering::equivalent`**
  - `std::`**`weak_ordering::greater`**
- `std::`**`partial_ordering`** with the values:
  - `std::`**`partial_ordering::less`**
  - `std::`**`partial_ordering::equivalent`**
  - `std::`**`partial_ordering::greater`**
  - `std::`**`partial_ordering::unordered`**

Note that all types have the values `less`, `greater`, and `equivalent`. However, `strong_ordering` also has `equal`, which is the same as `equivalent` there, and `partial_ordering` has the value `unordered`, representing neither less nor equal nor greater.

Stronger comparison types have implicit type conversions to weaker comparison types. That means you can use any `strong_ordering` value as `weak_ordering` or `partial_ordering` value (equal becomes `equivalent`, then).

1.2.3 Using Comparison Categories with `operator<=>`

The new operator `<=>` should return a value of one of the comparison category types, representing the result of the comparison combined with the information whether this result is able to create a strong/total, weak, or partial ordering.

For example, this is how a free-standing `operator<=>` might be defined for a type `MyType`:

```
std::strong_ordering operator<=> (MyType x, MyOtherType y)
{
  if (xIsEqualToY)  return std::strong_ordering::equal;
  if (xIsLessThanY)  return std::strong_ordering::less;
  return std::strong_ordering::greater;
}
```

Or as a more concrete example, defining `operator<=>` for a type `MyType`:

```
class MyType {
  ...
  std::strong_ordering operator<=> (const MyType& rhs) const {
    return value == rhs.value ? std::strong_ordering::equal :
           value < rhs.value ? std::strong_ordering::less :
                               std::strong_ordering::greater;
  }
};
```

However, it is usually easier to define the operator by mapping it to results of underlying types. So, the member `operator<=>` above would better just yield the value and category of its member `value`:

```
class MyType {
  ...
  auto operator<=> (const MyType& rhs) const {
    return value <=> rhs.value;
  }
};
```

This does not only return the right value; it also ensures that the return value has the right comparison category type depending on the type of the member `value`.

1.2.4 Calling `operator<=>`

You can call any defined operator `<=>` directly:

```
MyType x, y;
...
x <=> y          // yields a value of the resulting comparison category type
```

As written, the `operator<=>` is predefined for all fundamental types, for which the relational operators are defined. For example:

```
int x = 17, y = 42;
x <=> y                // yields std::strong_ordering::less
```

```
x <=> 17.0          // yields std::partial_ordering::equivalent
&x <=> &x           // yields std::strong_ordering::equal
&x <=> nullptr      // ERROR: relational comparison with nullptr not supported
```

In addition, all types of the C++ standard library that provide relational operators also provide `operator<=>` now. For example:

```
std::string{"hi"} <=> "hi"                  // yields std::strong_ordering::equal;
std::pair{42, 0.0} <=> std::pair{42, 7.7}   // yields std::partial_ordering::less
```

For your own type(s), you only have to define `operator<=>` as a member or free-standing function.

Remember that the return type depends on the comparison category. You can check against a specific return value:

```
if (x <=> y == std::partial_ordering::equivalent)   // always OK
```

Due to the implicit type conversions to weaker ordering types, this will even compile if the `operator<=>` yields a `strong_ordering` or `weak_ordering` value.

However, the other way around does not work. If the comparison yields a `weak_ordering` or `partial_ordering` value, you cannot compare it with a `strong_ordering` value.

```
if (x <=> y == std::strong_ordering::equal)    // might not compile
```

However, a comparison with `0` is always possible and usually easier:

```
if (x <=> y == 0)        // always OK
```

In addition, `operator<=>` might be called indirectly due to the new rewriting of relational operator calls:

```
if (!(x < y || y < x))   // might call operator<=> to check for equality
```

Or:

```
if (x <= y && y <= x)    // might call operator<=> to check for equality
```

Note that `operator!=` never gets rewritten to call `operator<=>`. However, it might might call an `operator==` member that is implicitly generated due to a defaulted `operator<=>` member.

1.2.5 Dealing with Multiple Ordering Criteria

To compute the result of `operator<=>` based on multiple attributes, you usually can implement just a chain of sub-comparisons until the result is not equal/equivalent or you reach the final attribute to compare:

```
class Person {
  ...
  auto operator<=> (const Person& rhs) const {
    auto cmp1 = lastname <=> rhs.lastname;     // primary member for ordering
    if (cmp1 != 0) return cmp1;                // return result if not equal
    auto cmp2 = firstname <=> rhs.firstname;   // secondary member for ordering
    if (cmp2 != 0) return cmp2;                // return result if not equal
    return value <=> rhs.value;                // final member for ordering
  }
};
```

However, the return type does not compile if the attributes have different comparison categories. For example, if a member `name` is a string and a member `value` is a `double`, we have conflicting return types:

```
class Person {
  std::string name;
  double value;
  ...
  auto operator<=> (const Person& rhs) const {  // ERROR: different return types deduced
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;     // return strong_ordering for std::string
    return value <=> rhs.value;     // return partial_ordering for double
  }
};
```

In that case you can define converting to the weakest comparison type. If you know the weakest comparison type, you can just use it as the return type:

```
class Person {
  std::string name;
  double value;
  ...
  std::partial_ordering operator<=> (const Person& rhs) const {  // OK
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;     // strong_ordering converted to return type
    return value <=> rhs.value;     // partial_ordering used as the return type
  }
};
```

If you do not know the comparison types (e.g., their type is a template parameter), you can use a new type trait `std::common_comparison_category<>` that computes the strongest comparison category:

```
class Person {
  std::string name;
  double value;
  ...
  auto operator<=> (const Person& rhs) const    // OK
   -> std::common_comparison_category_t<decltype(name <=> rhs.name),
                                        decltype(value <=> rhs.value)> {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;     // used as or converted to common comparison type
    return value <=> rhs.value;     // used as or converted to common comparison type
  }
};
```

By using the trailing return type syntax (with `auto` in front and the return type after `->`), we can use the parameters to compute the comparison types. Although in this case you could just use `name` instead of `rhs.name`, this approach works in general (e.g., also for freestanding functions).

If you want to provide a stronger category than used internally, you have to map all possible values of the internal comparsions to values of the return type. This might include some error handling if you cannot map some values. For example:

```
class Person {
  std::string name;
  double value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = x.name <=> y.name;
    if (cmp1 != 0) return cmp1;       // return strong_ordering for std::string
    auto cmp2 = x.value <=> y.value; // might be partial_ordering for double
    // map partial_ordering to strong_ordering:
    assert(cmp2 != std::partial_ordering::unordered);   // RUNTIME ERROR if unordered
    return cmp2 == 0 ? std::strong_ordering::equal
                     : cmp2 > 0 ? std::strong_ordering::greater
                                : std::strong_ordering::less;
  }
};
```

1.3 Defining `operator<=>` and `operator==`

Both `operator<=>` and `operator==` can be defined for your data types:

- either as a member function taking one parameter
- or a free-standing function taking two parameters

Inside a class or data structure (as a member or `friend` function), both operators can be declared as defaulted with `=default`. The member function has to take the second parameter as `const` lvalue reference (`const &`). A friend function might also take both parameters by value.

For defaulted definitions of both `operator<=>` and `operator==`:

- The operator is `noexcept` if comparing the members guarantees not to throw.
- The operator is `constexpr` if comparing the members is possible at compile time.

1.3.1 Defaulted `operator<=>`

If and only if an `operator<=>` member is defined as defaulted, then by definition a corresponding `operator==` member is also defined. All aspects (visibility, `virtual`, attributes, requirements, etc.) are adopted. For example:

```
template<typename T>
class Type {
  ...
 public:
  [[nodiscard]] virtual std::strong_ordering
     operator<=>(const Type&) const requires(!std::same_as<T,bool>) = default;
```

```
};
```

is equivalent to the following:

```
template<typename T>
class Type {
  ...
 public:
  [[nodiscard]] virtual std::strong_ordering
     operator<=> (const Type&) const requires(!std::same_as<T,bool>) = default;

  [[nodiscard]] virtual bool
     operator== (const Type&) const requires(!std::same_as<T,bool>) = default;
};
```

For example, the following is enough to support all 6 comparison operators for objects of type `Coord`:

lang/coord.hpp

```
#include <compare>

struct Coord {
  double x{};
  double y{};
  double z{};
  auto operator<=>(const Coord&) const = default;
};
```

Note again that the member function must be `const` and that the parameter must be declared to be a `const` lvalue reference (`const &`).

You can use this data structure as follows:

lang/coord.cpp

```
#include "coord.hpp"
#include <iostream>
#include <algorithm>

int main()
{
  std::vector<Coord> coll{ {0, 5, 5}, {5, 0, 0}, {3, 5, 5},
                           {3, 0, 0}, {3, 5, 7} };

  std::sort(coll.begin(), coll.end());
  for (const auto& elem : coll) {
    std::cout << elem.x << '/' << elem.y << '/' << elem.z << '\n';
  }
}
```

The program has the following output:

```
0/5/5
3/0/0
3/5/5
3/5/7
5/0/0
```

1.3.2 Defaulted `operator<=>` and Inheritance

If `operator<=>` is defaulted, you have a base class, and you call one of the relational operators, then the following happens:

- If for the base class `operator<=>` is defined, that operator is called.
- Otherwise, `operator==` and `operator<` are called to decide whether (from the base class point of view)
  - the objects are `equal`/`equivalent` (`operator==` yields `true`)
  - the objects are `less` or `greater`
  - the objects are `unordered` (only when partial ordering is checked)

  In that case, the return type of the defaulted `operator<=>` calling these operators cannot be `auto`.

For example, consider the following declarations:

```
struct Base {
  bool operator==(const Base&) const;
  bool operator<(const Base&) const;
};

struct Derived : public Base {
  std::strong_ordering operator<=> (const Derived&) const = default;
};
```

Then:

```
Derived d1, d2;
d1 > d2;   // calls Base::operator== and possibly Base::operator<
```

If `operator==` yields `true`, we know that the result of `>` is `false` and that is it. Otherwise, `operator<` is called to find out whether the expression is `true` or `false`.

With

```
struct Derived : public Base {
  std::partial_ordering operator<=> (const Derived&) const = default;
};
```

the compiler might call `operator<` even twice to find out whether there is any order at all.

With

```
struct Base {
  bool operator==(const Base&) const;
  bool operator<(const Base&) const;
```

```
};

struct Derived : public Base {
  auto operator<=> (const Derived&) const = default;
};
```

the compiler does not compile any call with relational operators because it cannot decide which ordering category the base class has. In that case, you need `operator<=>` in the base class, too.

However, checks for equality work, because in `Derived` `operator==` is automatically declared equivalent to the following:

```
struct Derived : public Base {
  auto operator<=> (const Derived&) const = default;
  bool operator== (const Derived&) const = default;
};
```

so we have the following behavior:

```
Derived d1, d2;
d1 > d2;    // ERROR: cannot deduce comparison category of operator<=>
d1 != d2;   // OK (note: only tries operator<=> and Base::operator== of a base class)
```

Equality checks are always only using `operator==` of a base class (which might be generated according to a defaulted `operator<=>`, though). Any `operator<` or `operator!=` in the base class are ignored.

1.4 Overload Resolution with Rewritten Expressions

Let us finally elaborate on the evaluation of expressions with comparison operators with the support of rewritten calls.

Calling Equality Operators

To compile

```
x != y
```

the compiler now might try all of the following:

```
x.operator!=(y)          // calling member operator!= for x
operator!=(x, y)         // calling a free-standing operator!= for x and y

!x.operator==(y)         // calling member operator== for x
!operator==(x, y)        // calling a free-standing operator== for x and y

!x.operator==(y)         // calling member operator== generated by operator<=> for x

!y.operator==(x)         // calling member operator== generated by operator<=> for y
```

The last form is tried to support an implicit type conversion for the first operand, which requires that the operand is a parameter.

In general, the compiler tries to call:

- a free-standing operator `!=`: `operator!=(x, y)`
  or a member operator `!=`: `x.operator!=(y)`
  Having both operators `!=` defined is an ambiguity error.
- a free-standing operator `==`: `!operator==(x, y)`
  or a member operator `==`: `!x.operator==(y)`
  Note that the member operator `==` may be generated from a defaulted `operator<=>` member.
  Again, having both operators `==` defined is an ambiguity error. This also applies if the member `operator==` is generated due to a defaulted `operator<=>`.

When an implicit type conversion for the first operand `v` is necessary, the compiler also tries to reorder the operands. Consider:

```
v != y      // v converts to the type of y
```

In that case the compiler tries to call:

- a free-standing operator `!=`: `operator!=(v, y)`
- a free-standing operator `==`: `!operator==(v, y)`
- a member operator `==`: `!y.operator==(v)`
  (note that the a member operator `==` may be generated from a defaulted `operator<=>` member)

So in that case a free-standing function is preferred over a member function because the order of the operands does not have to change.

Note that a rewritten expression never tries to call a member operator `!=`.

Calling Relational Operators

For the relational operators we have similar behavior except that the rewritten statements fall back on the new operator `<=>` and compare the result with `0`. The operator behaves like a three-way comparison function returning a negative value for *less*, `0` for *equal*, and a positive value for *greater* (it does *not* return a numeric value, but values that behave like that).

For example, to compile

```
x <= y
```

the compiler now might try all of the following:

```
x.operator<=(y)           // calling member operator<= for x
operator<=(x, y)          // calling a free-standing operator<= for x and y

x.operator<=>(y) <= 0     // calling member operator<=> for x
operator<=>(x, y) <= 0    // calling a free-standing operator<=> for x and y

0 <= y.operator<=>(x)     // calling member operator<=> for y
```

Again, the last form is tried to support an implicit type conversion for the first operand, for which it has to become a parameter.

1.5 Compatibility Issues with the Comparison Operators

After introducing the new rules for comparisons, it turned out that C++20 introduces some issues that may be a problem when switching from older C++ versions.

Here is the essence of the most typical problem:

lang/spacecompat.cpp

```
#include <iostream>

class MyType {
 private:
  int value;
 public:
  MyType(int i)     // implicit constructor from int:
   : value{i} {
  }

  bool operator==(const MyType& rhs) const {
    return value == rhs.value;
  }
};

bool operator==(int i, const MyType& t) {
  return t == i;    // OK with C++17
}

int main()
{
  MyType x = 42;
  if (0 == x) {       // C++17: true
    std::cout << "'0 == MyType{42}' works\n";
  }
}
```

We have trivial class that stores an integral value and has an implicit constructor to initialize the object (the implicit constructor is necessary to support using initialization using the =):

```
class MyType {
  ...
 public:
  MyType(int i);    // implicit constructor from int:
  ...
};

MyType x = 42;      // OK
```

The class also declared a member to be able to compare objects:

```
class MyType {
  ...
  bool operator==(const MyType& rhs) const;
  ...
};
```

However, this would only enable implicit type conversions for the second operand. So, the class or some other code might introduce a global operator that swaps the order or the arguments:

```
bool operator==(int i, const MyType& t) {
  return t == i;    // OK with C++17
}
```

Usually, the class should better define the `operator==()` as "hidden friend" (declare them with `friend` inside the class structure so that both operators become parameters and support implicit type conversions). However, this is a valid approach to have the same effect.

Unfortunately, this code no longer works in C++20.[1] It results in an endless recursion. The reason is that inside the global function the expression `t == i` can also call the global `operator==()` itself, because the compiler also tries to rewrite the call as `t == i`:

```
bool operator==(int i, const MyType& t) {
  return t == i;    // finds operator==(i,t) in addition to t.operator(MyType{i})
}
```

Unfortunately, the rewritten statement is a better match, because it does not need the implicit type conversion. We have no solution yet to support backward compatibility here; however, compilers start to warn about code like this already.

if the constructors are not `explicit`

1.6 Afternotes

The request for implicitly defined comparison operators was first proposed by Oleg Smolsky in http://wg21.link/n3950. The issue was raised then again by Bjarne Stroustrup in http://wg21.link/n4175. However, the final proposed wording, formulated by Jens Maurer in http://wg21.link/p0221r2, was finally rejected because it was an opt-out feature (thus, existing types would automatically have comparison operators unless they declared that they did not want this). Various proposals were then under consideration, which Herb Sutter brought together in http://wg21.link/p0515r0.

The finally accepted wording was formulated by Herb Sutter, Jens Maurer, and Walter E. Brown in http://wg21.link/p0515r3 and http://wg21.link/p0768r1. However, significant modifications were accepted as formulated by Barry Revzin in http://wg21.link/p1185r2 and http://wg21.link/p1630r1.

[1] Thanks to Peter Dimov and Barry Revzin for pointing out the problem and discussing it in http://stackoverflow.com/questions/65648897.

Chapter 2

Placeholder Types for Function Parameters

Generic programming is a key paradigm of C++. Therefore, C++20 also provides several new features for generic programming. However, there is one basic extension, which we will need more or less throughout this book: you can use now `auto` and other placeholder types to declare parameters of ordinary functions.

Later chapters will introduce more generic extensions:

- Extensions for non-type template parameters
- Lambda templates

2.1 `auto` for Parameters of Ordinary Functions

Since C++14, lambdas can use placeholders such as `auto` to declare/define their parameters:

```
auto printColl = [] (const auto& coll) {  // generic lambda
                   for (const auto& elem : coll) {
                     std::cout << elem << '\n';
                   }
                 }
```

They allow us to pass arguments of any type, provided the operations inside the lambda are supported:

```
std::vector coll{1, 2, 4, 5};
...
printColl(coll);                    // compiles the lambda for vector<int>

printColl(std::string{"hello"});    // compiles the lambda for std::string
```

Since C++20, you can use placeholders such as `auto` for all functions (including member functions and operators):

```
void printColl(const auto& coll)  // generic function
{
```

```
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

Such a declaration is just a shortcut to declare/define a template such as the following:

```
template<typename T>
void printColl(const T& coll)      // equivalent generic function
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

The only difference is that by using `auto` you no longer have a name for the template parameter `T`. For this reason, this feature is also called the *abbreviated function template* syntax.

Because functions with `auto` are function templates, all rules of using function templates apply. That especially has the consequence that you cannot implement a function with `auto` parameters in one translation unit (CPP file) while calling it in a different translation unit. Functions with `auto` parameters belong in a header file to be usable in multiple CPP files (or you have to explicitly instantiate the function in one translation unit). However, it does not need to be declared as `inline`, because function templates are always inline.

2.1.1 `auto` for Parameters of Member Functions

You can also use this feature to define member functions:

```
class MyType {
  ...
  void assign(const auto& newVal);
};
```

That declaration is equivalent to (with the difference that there is no type `T` defined):

```
class MyType {
  ...
  template<typename T>
  void assign(const T& newVal);
};
```

However, note that templates may not be declared inside functions. Therefore, with that feature you can no longer define the class or data structure locally inside a function:

```
void foo()
{
  struct Data {
    void mem(auto);   // ERROR can't declare templates insides functions
  };
}
```

See sentinel1.cpp for an example of a member operator == with auto.

2.1.2 auto Functions versus Lambdas

A function with auto parameters is different from a lambda. For example, you still cannot pass a function with auto as a parameter without specifying the generic parameter:

```
bool lessByNameFunc(const auto& c1, const auto& c2) {  // sorting criterion
  return c1.getName() < c2.getName();  // compare by name
}
...
std::sort(persons.begin(), persons.end(),
          lessByNameFunc);      // ERROR: can't deduce type of parameters in sorting criterion
```

Remember that the declaration of lessByName() is equivalent to:

```
template<typename T1, typename T2>
bool lessByName(const T1& c1, const T1& c2) {  // sorting criterion
  return c1.getName() < c2.getName();        // compare by name
}
```

Because the function template is not directly called, the compiler cannot deduce the template parameters to compile the call. Therefore, you have to explicitly specify the template parameters:

```
std::sort(persons.begin(), persons.end(),
          lessByName<Customer, Customer>);  // OK
```

By using a lambda, you do not have to specify the types of the template parameters when passing the lambda as argument:

```
lessByNameLambda = [] (const auto& c1, const auto& c2) {  // sorting criterion
                     return c1.getName() < c2.getName();  // compare by name
                   };
...
std::sort(persons.begin(), persons.end(),
          lessByNameLambda);   // OK
```

On the other hand, a function template parameter is easy to specify:

```
void printFunc(const auto& arg) {
  ...
}

printFunc<std::string>("hello");  // call function template compiled for std::string
```

Because a generic lambda is a function object with a generic function call operator operator(), to specify the template parameter explicitly, you have to pass it as argument to operator():

```
auto printFunc = [] (const auto& arg) {
                   ...
                 };
```

```
printFunc.operator()<std::string>("hello");   // call lambda compiled for std::string
```

2.1.3 `auto` for Parameters in Detail

Using `auto` to declare a function parameter follows the same rules as using it to declare a parameter of a lambda:

- For each parameter declared with `auto`, the function has an implicit template parameter.
- The parameters can be a parameter pack:

  ```
  void foo(auto... args);
  ```

  This is equivalent to the following (without introducing `Types`):

  ```
  template<typename... Types>
  void foo(Types... args);
  ```

- Using `decltype(auto)` is not allowed.

2.2 Other Placeholder Types for Parameters of Ordinary Functions

******* This chapter/section is at work *******

2.3 Afternotes

`auto` and other placeholder types for parameters of ordinary functions was first proposed together with the option to use type constraints for them by Ville Voutilainen, Thomas Köppe, Andrew Sutton, Herb Sutter, Gabriel Dos Reis, Bjarne Stroustrup, Jason Merrill, Hubert Tong, Eric Niebler, Casey Carter, Tom Honermann, and Erich Keane in `http://wg21.link/p1141r0`. The finally accepted wording was formulated by Ville Voutilainen, Thomas Köppe, Andrew Sutton, Herb Sutter, Gabriel Dos Reis, Bjarne Stroustrup, Jason Merrill, Hubert Tong, Eric Niebler, Casey Carter, Tom Honermann, Erich Keane, Walter E. Brown, Michael Spertus, and Richard Smith in `http://wg21.link/p1141r2`.

Chapter 3

Concepts and Requirements

This chapter introduces the new C++ feature *concepts* with the keywords `concept` and `requires`.

For sure, *concepts* are a milestone for C++, because concepts provide a language feature for something we need a lot when writing generic code: specifying *requirements*. Although there have been workarounds, we now have an easy and readable way to specify the requirements for generic code, get better diagnostics when requirements are broken, disable generic code when it does not work (although it might compile), and switch between different generic code for different types.

3.1 Motivating Example of Concepts and Requirements

Consider the following function template returning the maximum of two values:

```
template<typename T>
T maxValue(T a, T b) {
  return b < a ? a : b;
}
```

This function template can be called for two arguments that have the same type, provided the operations performed for the parameters (comparing with `operator<` and copying) are valid.

However, passing two pointers would compare their addresses instead of the values they refer to.

3.1.1 Improving the Template Step-by-Step

Using a `requires` Clause

To fix that, we can equip the template with a ***constraint*** so that it is not available if raw pointers are passed:

```
template<typename T>
requires (!std::is_pointer_v<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
```

```
}
```

Here, the constraint is formulated in a ***requires clause***, which is introduced with the keyword `requires` (there are other ways to formulate constraints).

To specify the constraint that the template cannot be used for raw pointers we use the standard *type trait* `std::is_pointer_v<>` (which yields the `value` member of the standard type trait `std::is_pointer<>`).[1] With this constraint we can no longer use the function template for raw pointers:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n';       // OK: maximum value of ints
std::cout << maxValue(&x, &y) << '\n';     // ERROR: constraint not met
```

The requirement is a compile-time check and has no impact on the performance of the compiled code. It only means that the template simply cannot be used for raw pointers. When raw pointers are passed, the compiler behaves as if the template were not there.

Defining and Using a `concept`

It might be pretty likely that we need a constraint for pointers more than once. For this reason, we can introduce a ***concept*** for it:

```
template<typename T>
concept IsPointer = std::is_pointer_v<T>;
```

A *concept* is a template where we introduce a name for one or more constraints on the passed template parameters. After the equal sign we have to specify the constraints as a Boolean expression that is evaluated at compile-time. In this case, we require that the template argument passed to `IsPointer<>` has to be a raw pointer.

Now, we can use this concept to constraint the `maxValue()` template:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

Overloading with Concepts

By using constraints and concepts we can even overload the `maxValue()` template to have one implementation for pointers and one for other types:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)           // maxValue() not for pointers
```

[1] Type traits were introduced as standard type functions with C++11 and the ability to call them with a `_v` suffix was introduced with C++17.

```
{
  return b < a ? a : b;        // compare values
}

template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b)        // maxValue() for pointers
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

Note that a requires clause that just constrains a template with a concept no longer needs parentheses.

Now, we have two function templates with the same name, but only one of them is available for each type:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n';       // calls maxValue() not for pointers
std::cout << maxValue(&x, &y) << '\n';     // calls maxValue() for pointers
```

Because the implementation for pointers delegate the computations of the return value to the objects the pointers refer to, the second calls uses both `maxValue()` templates. When passing pointers to `int`, we instantiate the template for pointers with `T` being `int*` and the basic `maxValue()` template that is not for pointers for `T` being `int`.

This even works recursively now. We can ask for the maximum value of a pointer to a pointer to an `int`:

```
int* xp = &x;
int* yp = &y;
std::cout << maxValue(&xp, &yp) << '\n';   // calls maxValue() for int**
```

Overload Resolution with Concepts

Overload resolution considers templates with constraints as more special than templates without. Therefore, it is enough to only constrain the implementation for pointers:

```
template<typename T>
T maxValue(T a, T b)           // maxValue() for a value of type T
{
  return b < a ? a : b;        // compare values
}

template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b)        // maxValue() for pointers (higher priority)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

However, be careful: overloading once using references and once not using references might cause ambiguities.

By using concepts it is even possible to prefer some constraints over others. However, this requires that concepts that *subsume* other concepts are used.

Type Constraints

If a constraint is a concept applied to a single parameter, there are ways to shortcut the specification of the constraint. First, you can specify it directly as a ***type constraint*** when declaring the template parameter:

```
template<IsPointer T>    // only for pointers
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

In addition, you can use the concept as a *type constraint* when declaring parameters with `auto`:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

This also works for parameters passed by reference:

```
auto maxValue3(const IsPointer auto& a, const IsPointer auto& b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

Note that by constraining both parameters directly, we changed the specification of the template: we no longer require that `a` and `b` have to have the same type. We only require that both are pointer-like objects of arbitrary type.

When using the template syntax, this looks as follows:

```
template<IsPointer T1, IsPointer T2>    // only for pointers
auto maxValue(T1 a, T2 b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

We probably should also allow different types types for the basic function template comparing values. One way to do that is to specify two template parameters:

```
template<typename T1, typename T2>
auto maxValue(T1 a, T2 b)       // maxValue() for values
{
  return b < a ? a : b;         // compare values
}
```

The other option would be to also use `auto` parameters:

```
auto maxValue(auto a, auto b)   // maxValue() for values
{
  return b < a ? a : b;         // compare values
}
```

Now, we could pass a pointer to an `int` and a pointer to a `double`.

Trailing `requires` Clauses

Consider the pointer version of `maxValue()`:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

There is still an implicit requirement that is not obvious: finally the values have to be comparable.

Compilers detect that requirement when (recursively) instantiating the `maxValue()` templates. However, the error message might be a problem, because the error occurs late and the requirement is not visible in the declaration of `maxValue()` for pointers.

Standard Concepts

To let the pointer version already declare that the values the pointers point to are comparable, we can add another constraint to the function:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::three_way_comparable_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

What we use here to declare the constraint is a ***trailing requires clause***, which can be specified ***after*** the parameter list. It has the benefit that it can use the parameters to formulate constraints.

In this case we use the parameters to pass the values they refer to a standard concept that checks whether the values are comparable. `three_way_comparable_with` is one of many standard concepts the C++ standard library provides in namespace `std`. Its name comes from the new way to define relational operators by using `operator<=>`.

`requires` Expressions

So far, the `maxValue()` templates do not work for pointer-like types that are not raw pointers, such as smart pointers. If the code should also compile for those types, we better define pointers as objects, for which we can call `operator*`.

Such a requirement is easy to specify since C++20:

```
template<typename T>
concept IsPointer = requires(T p) { *p; };   // constraint that calling operator * has to be valid
```

Instead of using the type trait for raw pointers, the concept formulates a simple requirement: The expression `*p` has to be valid for an object `p` of the passed type `T`.

Here, we are using the `requires` keyword again to introduce a ***requires expression***, which can define one or more ***requirements*** for types and parameters. By declaring a `p` of type `T`, we can simply specify which operations for such an object have to be supported.

We can also require multiple operations, type members, and that expressions yield specify types. For example:

```
template<typename T>
concept IsPointer = requires(T p) {
                      *p;              // operator * has to be valid
                      {p < p} -> std::convertible_to<bool>;   // < yields bool
                      p == nullptr;    // can compare with nullptr
                    };
```

Now we specify three requirements, which all apply to a parameter `p` of the type `T` we define this concept for

- The type has to support operator `*`.
- The type has to support operator `<`, which has to yield a Boolean value.
- Objects of that type have to be comparable with `nullptr`.

Note that we do not need two parameters of type `T` to check whether `<` can be called. The runtime value does not matter. However, note that there are some restrictions for how to specify what an expression yields (e.g., you cannot specify just `bool` there).

Also note that we do ***not*** require for `p` to be the `nullptr` here (that we can only check at runtime). We require only that we ***can*** compare `p` with the `nullptr`. However, that rules out iterators, because in general they cannot be compared with `nullptr` (except they are implemented as raw pointers as for `std::array<>`'s).

Again, this is a compile-time constraint that has no impact on the generated code; we only decide whether and for which types the code compiles. Therefore, it does not matter that we declare parameter `p` as a value instead of a reference.

You could also use the *requires expression* as an ad-hoc constraint directly in the *requires clause* (which looks a bit funny but makes total sense once you understand the difference of a requires clause and a requires expression):

```
template<typename T>
requires requires(T p) { *p; }  // constrain template with ad-hoc requirement
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);
}
```

3.1.2 The Whole Resulting Program

Now, we have everything to compute the maximum value for plain values and pointer-like objects. Here is a complete program:

lang/maxvalue.cpp

```
#include <iostream>

// concept for pointer-like objects:
template<typename T>
concept IsPointer = requires(T p) {
                      *p;              // operator * has to be valid
                      p == nullptr;    // can compare with nullptr
                      {p < p} -> std::convertible_to<bool>;   // < yields bool
                    };

// maxValue() for plain values:
auto maxValue(auto a, auto b)
{
  return b < a ? a : b;
}

// maxValue() for pointers:
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::three_way_comparable_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);  // return maximum value of where the pointers refer to
}

int main()
{
  int x = 42;
  int y = 77;
  std::cout << maxValue(x, y) << '\n';       // maximum value of ints
  std::cout << maxValue(&x, &y) << '\n';     // maximum value of where the pointers point to

  int* xp = &x;
  int* yp = &y;
  std::cout << maxValue(&xp, &yp) << '\n';   // maximum value of pointer to pointer

  double d = 49.9;
  std::cout << maxValue(xp, &d) << '\n';     // maximum value of int and double pointer
}
```

Note that we cannot check for the maximum of two iterator values:

```
std::vector coll{0, 8, 15, 11, 47};
auto pos = std::find(coll.begin(), coll.end(), 11);    // find specific value
if (pos != coll.end()) {
  // maximum of first and found value:
```

```
  auto val = maxValue(coll.begin(), pos);                // ERROR
}
```

The reason is that we require the parameters to be comparable with `nullptr`, which in general does not work. That demonstrates that it is important to think carefully about the definition of general concepts.

3.2 Typical Application of Concepts and Requirements in Practice

There are multiple reason why using constraints and requirements can be useful.

- Constraints help to *understand* the restrictions on templates and to get more understandable error messages when these requirements are broken.
- Constraints can be used to *disable generic code* for cases when it does not make sense:
  - Specify generic types might compile, but not do the right thing.
  - We might have to fix overload resolution, which decides which of multiple callable functions to call.
- Constraints can be used to *overload* or *specialize* generic code so that for different types different code is compiled.

Let us have a closer look to these reasons by developing another example step by step. This way we also introduce a couple of additional details about constraints, requirements, and concepts.

3.2.1 Requirements to Understand Code and Error Messages

Assume we want to write generic code that inserts the value of an object into a collection. Thanks to templates we can implement it once as code that is compiled for the types of passed objects once we know them:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

This code does not always compile. There is a hidden requirement for the types of the arguments: the call to `push_back()` for the value of type `T` has to be supported for the container of type `Coll`.

You could also argue that this is a combination of multiple basic requirements:

- Type `Coll` has to support `push_back()`.
- There has to be a conversion from type `T` to the type of the elements of `Coll`.
- If the passed argument has the elements type of `Coll`, that type has to support copying (because a new element is created initialized with the passed value).

If any of these requirements is broken, the code does not compile. For example:

```
std::vector<int> vec;
add(vec, 42);        // OK
add(vec, "hello");   // ERROR: no conversion from string literal to int

std::set<int> coll;
```

```
add(coll, 42);        // ERROR: no push_back() supported by std::set<>

std::vector<std::atomic<int>> aiVec;
std::atomic<int> ai{42};
add(aiVec, ai);       // ERROR: cannot copy/move atomics
```

When compilation fails, error messages can be very clear, such as when on the top level of a template the member `push_back()` is not found:

```
prog.cpp: In instantiation of 'void add(Coll&, const T&)
          [with Coll = std::__debug::set<int>; T = int]':
prog.cpp:17:18:   required from here
prog.cpp:11:8: error: 'class std::set<int>' has no member named 'push_back'
   11 |   coll.push_back(val);
      |   ~~~~~~~~~~~~~~
```

However, they can also be very tough to read and understand. For example, when the compiler has to deal with the broken requirement that copying is not supported, the problem is detected in the deep darkness of the implementation of `std::vector<>`. We get between 40 and 90 lines of error messages, where you have to carefully look for the broken requirement:

```
...
prog.cpp:11:17:   required from 'void add(Coll&, const T&)
                  [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]'
prog.cpp:25:18:   required from here
.../gcc/include/bits/stl_construct.h:96:17:
  error: use of deleted function
         'std::atomic<int>::atomic(const std::atomic<int>&)'
    96 |     -> decltype(::new((void*)0) _Tp(std::declval<_Args>()...))
       |                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

You might think you improve the situation by defining a concept that checks whether you can perform the `push_back()` call:

```
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
    c.push_back(v);
};
```

However, the check for being copyable still is detected in the deep darkness of the code for `std::vector<>` (this time when the concept is checked instead of when the code is compiled).

The situation gets better when we specify the basic requirement that is used by `push_back()` instead:

```
template<typename Coll, typename T>
requires std::convertible_to<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

Here, we use the standard concept `std::convertible_to` to ensure that the type of the passed argument `T` can be (implicitly and explicitly) converted to the element type of the collection.

Now, if the requirement is broken, we should get an error message with the broken concept and the location where it is broken. For example:[2]

```
...
prog.cpp:11:17:   In substitution of 'template<class Coll, class T>
                  requires  convertible_to<T, typename Coll::value_type>
                  void add(Coll&, const T&)
                  [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]':
prog.cpp:25:18:   required from here
.../gcc/include/concepts:72:13: required for the satisfaction of
                  'convertible_to<T, typename Coll::value_type>
                  [with T = std::atomic<int>;
                        Coll = std::vector<std::atomic<int>,
                                           std::allocator<std::atomic<int> > >]'
.../gcc/include/concepts:72:30: note: the expression 'is_convertible_v<_From, _To>
                  [with _From = std::atomic<int>; _To = std::atomic<int>]'
                  evaluated to 'false'
   72 |     concept convertible_to = is_convertible_v<_From, _To>
      |                              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

See rangessort.cpp for another example of algorithms checking the constraints of their parameters.

3.2.2 Requirements to Disable Generic Code

Assume we want to provide a special implementation for an `add()` function template like the one introduced above. When dealing with floating-point values, something different or in addition should happen.

A naive approach might be to overload the function template for `double`:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)     // for generic value types
{
  coll.push_back(val);
}

template<typename Coll>
void add(Coll& coll, double val)       // for floating-point value types
{
  ...    // special stuff for floating-point values
  coll.push_back(val);
}
```

As expected, when we pass a `double` as a second argument, the second function is called; otherwise, the generic argument is used:

```
std::vector<int> iVec;
add(iVec, 42);        // OK: calls add() for T being int
```

[2] This is just one example of a possible message, not necessarily matching the situation of a specific compiler.

```
std::vector<double> dVec;
add(dVec, 0.7);        // OK: calls 2nd add() for double
```

When passing a `double`, both function overloads match. The second overload is preferred, because it is a perfect match for the second argument.

However, if we pass a `float`, we have the following effect:

```
float f = 0.7;
add(dVec, f);          // OOPS: calls 1st add() for T being float
```

The reason lies in the sometimes subtle details of overload resolution. Again, both function could be called. Overload resolution has some general rules, such as:

- No type conversion is better than a type conversion.
- No template parameter is better than a template parameter.

Here, however, overload resolution has to decide between a type conversion and using a template parameter. Unfortunately, the version with the template parameter is preferred.

Fixing Overload Resolution

The fix for the wrong overload resolution is pretty simple. Instead of declaring the second parameter with a specific type, we should just require that the value to insert has a floating-point type. For this, we can constrain the function template by using the new standard concept `std::floating_point`:

```
template<typename Coll, typename T>
requires std::floating_point<T>
void add(Coll& coll, const T& val)
{
  ...    // special stuff for floating-point values
  coll.push_back(val);
}
```

Because we use a concept that applies to a single template parameters, we can also use the shorthand notation:

```
template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)
{
  ...    // special stuff for floating-point values
  coll.push_back(val);
}
```

For floating-point values we now have two function templates that can be called, one without and one with a specific requirement:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)     // for generic value types
{
  coll.push_back(val);
}
```

```
template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)     // for floating-point value types
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}
```

This is enough because overload resolution also prefers overloads or specializations that have constraints over those that have less or no constraints:

```
std::vector<int> iVec;
add(iVec, 42);       // OK: calls add() for generic value types

std::vector<double> dVec;
add(dVec, 0.7);      // OK: calls add() for floating-point types
```

Restrictions on Different Signatures

If two overloads or specializations have constraints, it is important that overload resolution can decide which one is better. For that, overloaded functions have to have the same signature.

If the signatures differ, the more constrained overload is not preferred. For example, if we declare the overload for floating-point values to take the argument by value, passing a floating-point value becomes ambiguous:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)  // note: pass by const reference
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, T val)         // note: pass by value
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}

std::vector<double> dVec;
add(dVec, 0.7);       // ERROR: both templates match and no preference
```

The latter declaration is no longer a special case of the former declaration. We have just two different function templates that both could be called.

If you really want to have different signatures, you have to constrain the first function template not to be available for floating-point values.

Restrictions on Narrowing

We have another interesting issue here: Both function templates allow us to pass a `double` to add it into a collection of `int`:

```
std::vector<int> iVec;

add(dVec, 1.9);        // OOPS: add 1
```

The reason is that we have implicit type conversions from `double` to `int` (due to being compatible to the programming language C). Such an implicit conversion where we might lose parts of the value is called *narrowing*. It means that the code above compiles and converts the value `1.9` to `1` before it gets inserted.

If you do not want to support narrowing, you have multiple options: One option is to disable type conversions at all by requiring that the passed value type matches the element type of the collection:

```
requires std::same_as<typename Coll::value_type, T>
```

However, this would also disable useful and safe type conversions.

For that reason, it is better to define a concept that yields whether a type can be converted to another type without narrowing, which is possible in a short tricky requirement:[3]

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
    std::convertible_to<From, To> &&
    requires (From&& x) {
      { std::type_identity_t<To[]>{std::forward<From>(x)} }
         -> std::same_as<To[1]>;
    };
```

Then we can use this concept to formulate a corresponding constraint:

```
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  ...
}
```

Subsuming Constraints

It would be probably enough to define the concept for narrowing conversions without requiring the concept `std::convertible_to`, because the rest checks this implicitly:

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing = requires (From&& x) {
    { std::type_identity_t<To[]>{std::forward<From>(x)} } -> std::same_as<To[1]>;
};
```

[3] Thanks to Giuseppe D'Angelo and Zhihao Yuan for the trick to check for narrowing conversions introduced in http://wg21.link/p0870.

However, there is an important benefit if the concept `ConvertsWithoutNarrowing` also checks for the concept `std::convertible_to`. In that case the compiler can detect that `ConvertsWithoutNarrowing` is more constrained than `std::convertible_to`. The terminology is that `ConvertsWithoutNarrowing` ***subsumes*** `std::convertible_to`.

That allows a programmer to do the following:

```
template<typename F, typename T>
requires std::convertible_to<F, T>
void foo(F, T)
{
   std::cout << "may be narrowing\n";
}

template<typename F, typename T>
requires ConvertsWithoutNarrowing<F, T>
void foo(F, T)
{
   std::cout << "without narrowing\n";
}
```

Without specifying that `ConvertsWithoutNarrowing` ***subsumes*** `std::convertible_to`, the compiler would raise an ambiguity error here when calling `foo()` with two parameters that convert to each other without narrowing.

3.2.3 Requirements to Use Different Statements

Finally, we might want to make our `add()` function template more flexible:

- We might also want to support collections that only provide `insert()` instead of `push_back()` to insert new elements.
- We might want to support passing a collection (container or range) to insert multiple values.

Yes, you can argue that these are different functions and they should have different names and if that works it is often better to use different names. However, the C++ standard library is a good example of the benefits you can get if different API's are harmonized. For example, you can use the same generic code to iterate over all containers, although internally containers use very different ways to go to the next element and access its value.

Calling Different Member Functions

To switch between using `push_back()` and `insert()` in the implementation of a function, we have several options. There are two issues to solve:

- We have to detect whether `push_back()` or `insert()` is supported.
- We have to ensure that we compile only calls that are supported.

Because the switch happens inside a function, we could use the `if constexpr` feature introduced with C++17. We can even use it directly with a `requires` expression as its condition:[4]

```
if constexpr (requires { coll.push_back(val); }) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

The other option is to introduce a name for the support of `push_back()` and `insert()`. Here, we also have multiple options:

- We can define a variable template of type `bool` (like type traits do):

```
template<typename T>
inline constexpr bool SupportsPushBack = requires(T coll) {
  coll.push_back(std::declval<typename T::value_type>());
};
```

- We can define a concept:

```
template<typename T>
concept SupportsPushBack = requires(T coll) {
  coll.push_back(std::declval<typename T::value_type>());
};
```

In both cases we only use one template argument by trying to insert an object of the element type of the container. Here, you can see that the definition of concepts and requirements does not create code. It is an unevaluated context where we can use `std::declval<>()` for "assume we would have an object of this type" and it does not matter whether we declare `coll` as a value or non-`const` reference.

The question whether to prefer defining a concept or a variable template of type `bool` is an interesting one discussed later in detail. However, let us look at what is technically possible here.

We can use any form of a named requirement in the `if constexpr` condition:

```
if constexpr (SupportsPushBack<coll>) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

The other option is to overload two implementations:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
```

[4] Thanks to Arthur O'Dwyer for pointing this out.

```
}

template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.insert(val);
}
```

With a `requires` clause, this works for both a concept and a named `bool`.

Note that we do not need a named requirement `SupportsInsert` here because the `add()` with the additional requirement is more special so that overload resolution prefers it.

If we define the named requirement as a concept, we can use it even as a constrained template parameter:

```
template<SupportsPushBack Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

This would not be possible with a named requirement defined as a variable template.

Before C++20, we had to use tricky declarations to "SFINAE" out the templates that did not fit.

Inserting Single and Multiple Values

To provide an overload that deals with multiple values passed as one collection, we can simply add constraints for them. The standard concept `std::ranges::input_range` can be used for that:

```
template<PushBackContainer Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), val.begin(), val.end());
}

template<InsertContainer Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(val.begin(), val.end());
}
```

Again, as long as the overload has this as an additional constraint, these functions will be preferred.

The concept `std::ranges::input_range` is a concept introduced to deal with ranges, which are collections you can iterate over with `begin()` and `end()`. However, ranges are not required to have `begin()` and `end()` as member functions. Code dealing with ranges should therefore use the helpers `std::ranges::begin()` and `std::ranges::end()` that the ranges library provides:

```
template<PushBackContainer Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
```

```
  coll.insert(coll.end(), std::ranges::begin(val), std::ranges::end(val));
}

template<InsertContainer Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(std::ranges::begin(val), std::ranges::end(val));
}
```

These helpers are in fact function objects, which avoids ADL problems.

Dealing with Multiple Constraints

By bringing together all useful concepts and requirements, we could place them all in one function at different locations.

```
template<PushBackContainer Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
              std::ranges::begin(val), std::ranges::end(val));
}
```

To disable narrowing conversions, we use `std::ranges::range_value_t` to pass the element type of the ranges to `ConvertsWithoutNarrowing`. `std::ranges::range_value_t` is another ranges utility to get the element type of ranges when iterating over them.

We could also formulate them together in the requires clause:

```
template<typename Coll, typename T>
requires PushBackContainer<Coll> &&
         std::ranges::input_range<T> &&
         ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
              std::ranges::begin(val), std::ranges::end(val));
}
```

Both ways to declare the function template are equivalent.

3.2.4 The Example as a Whole

Here is the final example as a whole:

lang/add.cpp

```
#include <iostream>
#include <vector>
#include <set>
#include <ranges>
#include <atomic>

// concept to disable narrowing conversions:
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
    std::convertible_to<From, To> &&
    requires (From&& x) {
      { std::type_identity_t<To[]>{std::forward<From>(x)} }
          -> std::same_as<To[1]>;
    };

// concept for container with push_back():
template<typename T>
concept SupportsPushBack = requires(T coll) {
  coll.push_back(std::declval<typename T::value_type>());
};


// add() for single values:
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}

// add() for multiple values:
template<typename Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.insert(coll.end(),
                std::ranges::begin(val), std::ranges::end(val));
  }
```

```
  else {
    coll.insert(std::ranges::begin(val), std::ranges::end(val));
  }
}

int main()
{
  std::vector<int> iVec;
  add(iVec, 42);       // OK: calls push_back() for T being int

  std::set<int> iSet;
  add(iSet, 42);       // OK: calls insert() for T being int

  short s = 42;
  add(iVec, s);        // OK: calls push_back() for T being short

  long long ll = 42;
  //add(iVec, ll);     // ERROR: narrowing
  //add(iVec, 7.7);    // ERROR: narrowing

  std::vector<double> dVec;
  add(dVec, 0.7);      // OK: calls push_back() for floating-point types
  add(dVec, 0.7f);     // OK: calls push_back() for floating-point types
  //add(dVec, 7);      // ERROR: narrowing

  // insert collections:
  add(iVec, iSet);     // OK: insert set elements into a vector
  add(iSet, iVec);     // OK: insert vector elements into a set

  // can even insert raw array:
  int vals[] = {0, 8, 18};
  add(iVec, vals);     // OK
  //add(dVec, vals);   // ERROR: narrowing
}
```

3.2.5 Former Workarounds

Constraining templates was already possible before C++20. However, the way this was possible often was not easy to use and could provide significant drawbacks.

SFINAE

The major approach to disable the availability of templates before C++20 was SFINAE. The term "SFINAE" (pronounced like *sfee-nay*) stands for "*substitution failure is not an error*" and means the rule that we simply ignore generic code if its declaration is not well-formed instead of raising a compile-time error.

For example, to switch between `push_back()` and `insert()`, we could declare the function templates before C++20 as follows:

```
template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.push_back(val))
{
  return coll.push_back(val);
}

template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.insert(val))
{
  return coll.insert(val);
}
```

Because we make the call to `push_back()` here part of the first template **declaration**, this template is ignored if `push_back()` is not supported. The corresponding declaration is necessary in the second template for `insert()` (overload resolution does not understand that one template is more constraint than the other).

However, that is a very subtle way to place a requirement and programmers might overlook it easily.

`std::enable_if`

For more complicated cases of disabling generic code the C++ standard library since C++11 provides `std::enable_if<>`.

This is a type trait that takes a Boolean condition which yields invalid code if `false`. By using the `std::enable_if<>` somewhere inside a declaration, this could also "SFINAE out" generic code.

For example, you could use the `std::enable_if<>` type trait to exclude types from calling the `add()` function template in the following way:

```
// disable the template for floating-point values:
template<typename Coll, typename T,
         typename = std::enable_if_t<!std::is_floating_point_v<T>>>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

The trick is to insert an additional template parameter to be able to use the trait `std::enable_if<>`. The trait yields `void` if it does not disable the template (you can specify that it yields another type as optional second template parameter).

Code like this is also hard to write and read and has subtle drawbacks. For example, for a second requirement you would need yet another template parameter and you have to be very careful that at each time exactly one of the different overloads is available.

Concepts provide a far more readable way of formulating constraints, including the important benefit that compilers can detect whether a constraint subsumes another constraint.

3.3 Constraints and Requirements in Detail

Let us discuss how to specify constraints and requirements in detail.

3.3.1 Constraints

To specify requirements for template parameters you need ***constraints***, which are Boolean expressions that evaluate to `true` or `false` at compile time to decide whether to instantiate and compile a template or not.

A constraint can be:

- an ad-hoc Boolean compile-time expression (with a restricted syntax so that it might need parentheses around to avoid ambiguities)
- a `requires` expressions specifying required operations and types
- a concept (defined before, by using the keyword `concept`)

The basic application of a constraint is a *requires clause*, which uses the keyword `requires` and the constraint(s) to restrict the availability of the template. All constraints could also be used wherever a Boolean expression can be used (especially as an `if constexpr` condition).

To combine multiple constraints, we can use `&&`. For example:

```
template<typename T>
requires (sizeof(T) > 4)                          // ad-hoc Boolean expression
         && requires { typename T::value_type; }  // requires expression
         && std::input_iterator<T>                // concept
```

The order of the constraints does not matter.

It is also possible to express "alternative" constraints using `||`. For example:

```
template<typename T>
requires std::integral<T> || std::floating_point<T>
T power(T b, T p);
```

Specifying alternative constraints is rarely needed and should not be done too casually because excessive use of the `||` operator in requires clauses may potentially tax compilation resources (i.e., make compilation noticeably slower).

A single constraint can also involve multiple template parameters: For example:

```
template<typename T, typename U>
requires std::convertible_to<T, U>
auto f(T x, U y) {
  ...
}
```

That way, constraints can impose a relationship between type parameters.

3.3.2 Ad hoc Boolean Expressions

The first way to formulate constraints for templates is to use compile-time expressions that convert to `true` or `false`. These expressions may especially use:

- Type predicates, such as type traits

- Compile-time variables (defined with `constexpr` or `constinit const`)
- Compile-time functions (defined with `constexpr` or `consteval`)

Let us look at some examples of ad hoc Boolean expressions to restrict the availability for a template:

- Available only if `int` and `long` have a different size:

```
template<typename T>
requires (sizeof(int) != sizeof(long))
...
```

- Available only if `sizeof(T)` is not too large:

```
template<typename T>
requires (sizeof(T) <= 64)
...
```

- Available only if the non-type template parameter `Sz` is greater than zero:

```
template<typename T, std::size_t Sz>
requires (Sz > 0)
...
```

- Available only for raw pointers and the `nullptr`:

```
template<typename T>
requires (std::is_pointer_v<T> || std::same_as<T, std::nullptr_t>)
...
```

  `std::same_as` is a new standard concept. You could also use the type trait `std::is_same<>`:

```
template<typename T>
requires (std::is_pointer_v<T> || std::is_same_v<T, std::nullptr_t>)
...
```

- Available only if the argument cannot be used as a string:

```
template<typename T>
requires (!std::convertible_to<T, std::string>)
...
```

  `std::convertible_to` is a new standard concept. You could also use the type trait `std::is_convertible<>`:

```
template<typename T>
requires (!std::is_convertible_v<T, std::string>)
...
```

- Available only if the argument is a pointer (or pointer-like object) to an integral value:

```
template<typename T>
requires std::integral<std::remove_reference_t<decltype(*std::declval<T>())>>
...
```

  Note that `operator*` usually yields a reference, which is not an integral type. Therefore, we do the following:

  – Assume we have an object of type `T`: `std::declval<T>()`

  - Call `operator*` for it: `*`
  - Ask for its type: `decltype()`
  - Remove referenceness: `std::remove_reference_v<>`
  - Check of integral type: `std::integral<>`

  `std::integral` is a new standard concept. You could also use the type trait `std::is_integral<>`. This constraint would also be satisfied by a `std::optional<int>`.
- Available only if the non-type template parameters `Min` and `Max` have a greatest common divisor that is greater than one:

```
template<typename T>
constexpr bool gcd(T a, T b);   // greatest common divisor (forward declaration)

template<typename T, int Min, int Max>
requires (gcd(Min, Max) > 1)    // available if there is a GCD greater than 1
...
```

- Disable a template (temporarily):

```
template<typename T>
requires false  // disable the template
...
```

Usually, you need parentheses around the whole expression. The only exception are constraints using identifiers, `::`, and `<...>` (optionally combined with `&&` and `||`):

```
requires std::convertible_to<T, int>      // no parentheses needed here
          &&
         (!std::convertible_to<int, T>)   // ! forces the need of parentheses
```

3.3.3 `requires` Expressions

Requires expressions (which are distinct from *requires clauses*) provide a simple but flexible syntax to specify multiple requirements on one or multiple template parameters. You can specify:

- Required type definitions
- Expressions that have to be valid
- Requirements on the types that expressions yield

A *requires expression* starts with `requires` followed by an optional parameter list and then a block of requirements (all ending with semicolons). For example:

```
template<typename Coll>
... requires {
      typename Coll::value_type::first_type;    // elements/values have first_type
      typename Coll::value_type::second_type;   // elements/values have second_type
    }
```

The optional parameter list allows you to introduce a set of "dummy variables" usable to express requirements in the body of the requires expression:

```
template<typename T>
```

```
... requires(T x, T y) {
      x + y;    // supports +
      x - y;    // supports -
    }
```

These parameters are never replaced by arguments. Therefore, it usually does not matter whether you declare them by value or by reference.

The parameters also allow us to introduce (parameters of) sub-types:

```
template<typename Coll>
... requires(typename Coll::value_type v) {
      std::cout << v;    // supports output operator
    }
```

This requirements checks whether `Coll::value_type` is valid and whether objects of this type support the output operator.

When using this to check only whether `Coll::value_type` is valid, you do not need anything in the body of the block of the requirements. However, the block cannot be empty. So, you might simply use `true` then:

```
template<typename Coll>
... requires(typename Coll::value_type v) {
      true;              // dummy requirement because the block cannot be empty
    }
```

Simple Requirements

Simple requirements are just expressions that have to be well-formed. That means the calls have to compile. The calls are not performed, so it does not matter whether the operations have defined behavior or yield `true`.

For example:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
      *p;              // operator* has to be supported for T2
      p[0];            // operator[] has to be supported for int as index
      p->value();      // calling a member function value() without arguments has to be possible
      *p > val;        // support that we can compare the result of operator* with T1
      p == nullptr;    // support that we can compare a T2 with a nullptr
    }
```

The last call does not require that `p` ***is*** the `nullptr` (to require that, you have to check whether `T2` is type `std::nullptr_t`). Instead, we require that we ***can*** compare an object of type `T2` with an object of type `std::nullptr_t` (the type of `nullptr`).

It usually does not make sense to use operator `||`. A simple requirement such as

```
*p > val || p == nullptr;
```

does ***not*** require that either the left or the right sub-expression is possible. It formulates the requirement that we can combine the results of both sub-expressions with `operator||`.

To require either one of the two sub-expressions, you have to use:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
      *p > val;           // support that we can compare the result of operator* with T1
    }
    || requires(T2 p) {   // OR
      p == nullptr;       // support that we can compare a T2 with a nullptr
    }
```

Also note that this concept does ***not*** require that `T` to be an integral type:

```
template<typename T>
... requires {
      std::integral<T>;             // OOPS: does not require T to be integral
      ...
    };
```

It only requires that the expression `std::integral<T>` is valid, which is the case for all types. Instead, you have to formulate it as follows:

```
template<typename T>
... std::integral<T> &&             // OK, does require T to be integral
    requires {
      ...
    };
```

or as follows:

```
template<typename T>
... requires {
      requires std::integral<T>;    // OK, does require T to be integral
      ...
    };
```

Type Requirements

Type requirements are expressions that have to be well-formed when using a name of a type. That means the specified name has to be defined as a valid type.

For example:

```
template<typename T1, typename T2>
... requires {
  typename T1::value_type;                // type member value_type required for T1
  typename std::ranges::iterator_t<T1>;   // iterator type required for T1
  typename std::ranges::iterator_t<std::vector<T1>>;
  typename std::common_type_t<T1, T2>;    // T1 and T2 have to have a common type
}
```

For all type requirements, if the type exists but is `void` then the requirement is met.

Note that you can only check for names given to types (names of classes, enumeration types, from `typedef` or `using`). You cannot check for other type declarations using the type:

```
template<typename T>
... requires {
  typename int;              // ERROR: invalid type requirement
  typename T&;               // ERROR: invalid type requirement
}
```

The way to test the latter is to declare a corresponding parameter:

```
template<typename T>
... requires(T&) {
  true;       // some dummy requirement
};
```

Again, the requirements checks whether using the passed type(s) to define another type is valid. For example:

```
template<std::integral T>
class MyType1 {
  ...
};

template<typename T>
requires requires {
  typename MyType1<T>;      // instantiation of MyType1 for T would be valid
}
void mytype1(T) {
  ...
}

mytype1(42);     // OK
mytype1(7.7);    // ERROR
```

Therefore, the following requirement does ***not*** check whether there is a standard hash function for type `T`:

```
template<typename T>
concept StdHash = requires {
  typename std::hash<T>;   // does not check whether std::hash<> is defined for T
};
```

The way to do that is to try to create or use it:

```
template<typename T>
concept StdHash = requires {
  std::hash<T>{};   // OK, checks whether we can create a standard hasher for T
};
```

It does not make sense to use type functions that always yield a type:

```
template<typename T>
... requires {
```

```
  typename std::remove_const_t<T>;     // not useful as type requirement (always valid)
}
```

The requirement only checks whether the type expression yields a type, which is always the case here. To check for constness, use:

```
template<typename T>
... std::is_const<T>   // ensure that T is const
```

It also does not make sense to use type functions may have undefined behavior. For example, the type trait `std::make_unsigned<>` requires that the passed argument is an integral type other than `bool`. If this is not the case, you get undefined behavior. If you used it as a requirement:

```
std::make_unsigned_r<T>::type     // not useful as type requirement (valid or undefined behavior)
```

the requirement can only be fulfilled or results in undefined behavior (which might mean that the requirement is still fulfilled).

Compound Requirements

Compound requirements allow us to combine the abilities of simple and type requirements. In this case you can specify an expression (inside a block of braces) and then add one or both of the following:

- `noexcept` to require that the expression guarantees not to throw
- `->` *type-constraint* to apply a concept on what the expression evaluates to

Here are some examples:

```
template<typename T>
... requires(T x) {
  { &x } -> std::input_or_output_iterator;
  { x == x }
  { x == x } -> std::convertible_to<bool>;
  { x == x }noexcept
  { x == x }noexcept -> std::convertible_to<bool>;
}
```

Note that the type constraint after the `->` takes the resulting type as its first template argument. That means:

- In the first requirement we require that the concept `std::input_or_output_iterator` is satisfied when using `operator&` for an object of type `T` (`std::input_or_output_iterator<decltype(&x)>` yields `true`).

  You could also specify this as follows:

  ```
  { &x } -> std::is_pointer_v<>;
  ```

- In the last requirement we require that we can use the result of `operator==` for two objects of type `T` as `bool` (the concept `std::convertible_to` is satisfied when passing the result of `operator==` for two objects of type `T` and `bool` as arguments).

Requires expressions can also express the need for associated types. For example:

```
template<typename T>
... requires(T coll) {
  { *coll.begin() } -> std::convertible_to<T::value_type>;
```

```
}
```

However, you cannot specify type requirement using nested types. For example, you cannot use it to require that the return value of `operator*` yields an integral value. The problem is that the return value is a reference you have to dereference first:

```
std::integral<std::remove_reference_t<T>>;
```

However, you can use such a nested expression here:

```
template<typename T>
concept Check = requires(T p) {
  { *p } -> std::integral<std::remove_reference_t<>>;   // ERROR
  { *p } -> std::integral<std::remove_reference_t>;     // ERROR
};
```

You either have to define a corresponding concept first:

```
template<typename T>
concept UnrefIntegral = std::integral<std::remove_reference_t<T>>;

template<typename T>
concept Check = requires(T p) {
  { *p } -> UnrefIntegral;  // OK
};
```

Or you have to use a nested requirement.

Nested Requirements

Nested requirements allow us to specify additional constraints by using local parameters.

For example, we can use a nested requirement to solve the problem just introduced to specify a complex type requirement on an expression:[5]

```
template<typename T>
concept Check = requires(T p) {
  requires std::integral<std::remove_cvref_t<decltype(*p)>>;
};
```

3.4 Concepts in Detail

By defining a `concept` you can introduce a name for one or more *constraints*.

Templates (function, class, and variable templates) can use concepts to constrain their ability (via a requires clause or as a direct type constraint for a template parameter). However, concepts are also Boolean compile-time expressions (type predicates) you can use wherever you have to check something for a type (such as in an `if constexpr` condition).

[5] Thanks to Hannes Hauswedell for pointing this out.

3.4.1 Defining Concepts

Concepts are defined as follows:

```
template<...>
concept name = ... ;
```

The equal sign is required (you cannot declare a concept without defining it). Behind the equal sign you can specify any compile-time expression that converts to `true` or `false`.

Concepts are much like `constexpr` variable templates of type `bool`, but the type is not explicitly specified:

```
template<typename T>
concept MyConcept = ... ;

std::is_same<MyConcept<...>, bool>   // yields true
```

That means, at compile-time or runtime you can always use a concept where the value of a Boolean expression is needed. However, you cannot take the address because there is no object behind it (it is a prvalue).

The template parameters may not have constraints (you cannot use a concept to define a concept).

You cannot define concepts inside a function (as is the case for all templates).

3.4.2 Special Abilities of Concepts

Concepts have special abilities. If you compare the usual definition of a Boolean variable template (as it is especially used to define type traits):

```
template<typename T>
inline constexpr bool IsOrHasThisOrThat = ... ;
```

with a definition of a concept:

```
template<typename T>
concept IsOrHasThisOrThat = ... ;
```

we have the following differences:

- Concepts do not represent code. They have no type, storage, lifetime, or any other properties associated with objects.

  By instantiating them at compile time for specific template parameters, their instantiation just becomes `true` or `false`. Therefore, you can use them wherever you can use `true` or `false` and you get all properties of these literals.
- Concepts do not have to be declared as `inline`, they implicitly are.
- Concepts can be used as type constraints:

  ```
  template<IsOrHasThisOrThat T>
  ...
  ```

  Variable templates cannot be used that way.
- Concepts are the only way to give *constraints* a name, which means that you need them to decide whether a constraint is a special case of another constraint. To decide whether a constraint subsumes you need concepts to compare the constraint with.

3.4.3 Using Concepts as Type Constraints

As introduced, concepts can be used as type constraints. There are different places where type constraints can be used:

- In the declaration of a template type parameter
- In the declaration of a call parameter declared with `auto`
- As requirement in a compound requirements

For example:

```
template<std::integral T>                          // type constraint for a template parameter
class MyClass {
  ...
};

auto myFunc(const std::integral auto& val) {       // type constraint for an auto parameter
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::integral;                    // type constraint for return type
  };
```

Here, we use unary constraints that are called for a single parameter or type returned by an expression.

Type Constraints with Multiple Parameters

You can also use constraints with multiple parameters, for which the parameter type or return value is then used as the first argument:

```
template<std::convertible_to<int> T>               // conversion to int required
class MyClass {
  ...
};

auto myFunc(const std::convertible_to<int> auto& val) {  // conversion to int required
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::convertible_to<int>;         // conversion to int required
  };
```

Another example often used is to constrain the type of a callable (function, function object, lambda) to require that you can pass a certain number of arguments of certain types using the concepts std::invocable or

std::regular_invocable: For example, to require to pass an operation that takes an `int` and a `std::string`, you have to declare:

```
template<std::invocable<int, std::string> Callable>
void call(Callable op);
```

or:

```
void call(std::invocable<int, std::string> auto op);
```

The difference between `std::invocable` and `std::regular_invocable` is that the latter guarantees not to modify the passed operation and arguments. That is a semantic difference which only helps to document the intention. Often just `std::invocable` is used.

Type Constraints and `auto`

Type constraints can be used in all places where `auto` can be used:

- To constrain declarations:

```
std::integral auto val1 = 42;    // OK
std::integral auto val2 = true;  // ERROR

for (const std::integral auto& elem : coll) {
  ...
}
```

- To constrain return types:

```
std::integral auto foo() {
  ...
}
```

- to constrain non-type template parameters:

```
template<typename T, std::integral auto max>
class SizedColl {
  ...
};
```

This also works with concepts taking multiple parameters:

```
template<typename T, std::convertible_to<T> auto default>
class MyType {
  ...
};
```

For another example, see the support for lambdas as non-type template parameters.

3.5 Subsuming Constraints and Concepts

Two concepts can have a subsuming relation. That is, one concept can be specified that it restricts one or more other concepts. The benefit is that overload resolution then prefers the more constrained generic code over the less constrained generic code when both constraints are satisfied.

For example, consider we introduce the following two concepts:

```
template<typename T>
concept GeoObject = requires(T obj) {
    { obj.width() } -> std::integral;
    { obj.height() } -> std::integral;
    obj.draw();
  };

template<typename T>
concept ColoredGeoObject =
  GeoObject<T> &&              // subsumes concept GeoObject
  requires(T obj) {            // additional constraints
    obj.setColor(Color{});
    { obj.getColor() } -> std::convertible_to<Color>;
  };
```

The concept `ColoredGeoObject` explicitly *subsumes* the concept `GeoObject` because it explicitly formulates the constraint that type `T` also has to satisfy the concept `GeoObject`.

As a consequence, when overloading templates for both concepts and both are satisfied, we do not get an ambiguity. Overload resolution prefers the concept that subsumes the other(s):

```
template<GeoObject T>
void process(T)          // called for objects not providing setColor() and getColor()
{
  ...
}

template<ColoredGeoObject T>
void process(T)          // called for objects providing setColor() and getColor()
{
  ...
}
```

Constraint subsumption only works when concepts are used. There is no automatic subsumption when one concept/constraint is more special than the other.

Constraints and concepts do *not* subsume based only on requirements. Consider the following example:[6]

```
// declared in some header:
```

[6] That was in fact the example discussed regarding this feature during standardization. Thanks to Ville Voutilainen for pointing that out.

```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

// declared in another header:
template<typename T>
concept Cowboy = requires(T obj) {
                   obj.draw();
                   obj = obj;
                 };
```

Consider we overload a function template for both, `GeoObject`'s and `Cowboys`'s:

```
template<GeoObject T>
void print(T) {
  ...
}

template<Cowboy T>
void print(T) {
  ...
}
```

We do not want that for a `Circle` or `Rectangle`, which have a `draw()` member function, the call to `print()` prefers the `print()` for cowboys, just because the `Cowboy` concept is more special. We want to see that there are two possible `print()` functions that in this case collide.

The effort to check for subsumptions is only evaluated for concepts. Overloading with different constraints is ambiguous if no concepts are used:

```
template<typename T>
requires std::is_convertible_v<T, int>
void print(T) {
  ...
}

template<typename T>
requires (std::is_convertible_v<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42);   // ERROR: ambiguous (if both constraints are true)
```

When using concepts instead, this code works:

```
template<typename T>
requires std::convertible_to<T, int>
void print(T) {
```

```
  ...
}

template<typename T>
requires (std::convertible_to<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42);  // OK
```

One reason for this behavior is that it takes compile time to process dependencies between concepts in detail.

The concepts provided by the standard library are carefully designed to subsume other concepts when it makes sense. For example:

- `std::random_access_range` subsumes `std::bidirectional_range`, both subsume the concept `std::forward_range`, all three subsume `std::input_range`, and all of them subsume `std::range`. However, `std::sized_range` does only subsume `std::range` and none of the others.
- `std::regular` subsumes `std::semiregular` and both subsume both `std::copyable` and `std::default_initializable` (which subsume several other concepts such as `std::movable`, `std::copy_constructible`, and `std::destructible`).
- `std::sortable` subsumes `std::permutable` and both subsume `std::indirectly_swappable` for both parameters being the same type.

3.5.1 Indirect Subsumptions

Constraints can even subsume indirectly.[7] That means, overload resolution can still prefer one overload or specialization over the other although their constraints are not defined in terms of each other.

For example, consider you have defined the following two concepts:

```
template<typename T>
concept RgSwap = std::ranges::input_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

Now when we overload two functions for these two concepts and pass an object that fits both concepts, this is no ambiguity:

```
template<RgSwap T>
void foo1(T) {
  std::cout << "foo1(RgSwap)\n";
}

template<ContCopy T>
```

[7] Thanks to Arthur O'Dwyer for pointing this out.

```
void foo1(T) {
  std::cout << "foo1(ContCopy)\n";
}

foo1(std::vector<int>{});     // OK: both fit, ContCopy is more constraint
```

The reason is that `ContCopy` subsumes `RgSwap` because

- Concept `contiguous_range` is defined in terms of concept `input_range` (it implies `random_access_range`, which implies `bidirectional_range`, which implies `forward_range`, which implies `input_range`).
- Concept `copyable` is defined in terms of concept `swappable` (it implies `movable`, which implies `swappable`).

However, with the following declarations we get an ambiguity when both concepts fit:

```
template<typename T>
concept RgSwap = std::ranges::sized_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

The reason is that neither the concept `contiguous_range` implies `sized_range` nor the concept `sized_range` implies `contiguous_range`.

Also, for the following declarations, no concept subsumes the other:

```
template<typename T>
concept RgCopy = std::ranges::input_range<T> && std::copyable<T>;

template<typename T>
concept ContMove = std::ranges::contiguous_range<T> && std::movable<T>;
```

On one hand, `ContMove` is more constrained because `contiguous_range` implies `input_range`; however, on the other hand, `RgCopy` is more constrained because `copyable` implies `movable`.

To avoid confusion, do not make too many assumptions about concepts subsuming each other. When in doubt, specify all the concepts you require.

3.6 Semantic Constraints

Concepts might check both syntactical and semantic constraints:

- **Syntactic constraints** mean that we can check at compile-time whether certain requirements are satisfied (“*Is a specific operation supported?*” or “*Does a specific operation yield a specific type?*”).
- **Semantic constraints** mean that certain requirements are satisfied, which can only be checked at runtime (“*Does an operation have the same effect?*” or “*Does the same operation performed for a specific value always yield the same result?*”).

A simple example of a semantic constraint is the difference between the concepts std::invocable and std::regular_invocable. The latter guarantees not to modify the state of the passed operation and the passed arguments. We cannot check that with a compiler, so the difference document the ***intention*** of the specified API. Often, for simplicity, just `std::invocable` is used.

As another example, besides certain syntactic differences there is a semantic difference between the concepts `weakly_incrementable` and `incrementable`:

- `weakly_incrementable` only requires that a type supports the increment operators. Incrementing the same value may yield different results.
- `incrementable` requires each increment of the same value gives the same result.

Therefore:

- When `incrementable` is satisfied, you can iterate multiple times from a starting value over a range.
- When only `weakly_incrementable` is satisfied, you can iterate over a range only once. A second iteration with the same staring value might yield different results.

This difference matters for iterators: Input stream iterators (iterators that read values from a stream) can only iterate once because the next iteration yields different values. Consequently, input stream iterators satisfy the `weakly_incrementable` concept but not the `incrementable` concept. However, the concepts cannot be used to check for this difference:

```
std::weakly_incrementable<std::istream_iterator<int>>  // yields true
std::incrementable<std::istream_iterator<int>>         // OOPS: also yields true
```

The reason is that the difference is a semantic constraint that cannot be checked at compile time. So the concepts can be used to *document* the constraints:

```
template<std::weakly_incrementable T>
void algo1(T beg, T end);                    // single-pass algorithm

template<std::incrementable T>
void algo2(T beg, T end);                    // multi-pass algorithm
```

Note that we use different names for the algorithms here. Due to the fact that we cannot check the semantic difference of the constraints, it is up to the programmer to not pass an input stream iterator:

```
algo1(std::istream_iterator<int>{std::cin},  // OK
      std::istream_iterator<int>{});

algo2(std::istream_iterator<int>{std::cin},  // OOPS: violates constraint
      std::istream_iterator<int>{});
```

However, you cannot dispatch between two different implementations based on this difference:

```
template<std::weakly_incrementable T>
void algo(T beg, T end);                     // single-pass implementation

template<std::incrementable T>
void algo(T beg, T end);                     // multi-pass implementation
```

If you pass an input stream iterator here, the compiler will wrongly use the multi-pass implementation:

```
algo(std::istream_iterator<int>{std::cin},   // OOPS: calls the wrong overload
     std::istream_iterator<int>{});
```

Fortunately, there is a solution here, because for this semantic difference C++98 already had introduced iterator traits, which are used by the iterator concepts. If you use these concepts (or the corresponding range concepts), everything works fine:

```
template<std::input_iterator T>
void algo(T beg, T end);                        // single-pass implementation

template<std::forward_iterator T>
void algo(T beg, T end);                        // multi-pass implementation

algo(std::istream_iterator<int>{std::cin},      // OK: calls the right overload
     std::istream_iterator<int>{});
```

So, in this case you should prefer the more specific concepts for iterators and ranges.

3.7 Design Guidelines for Concepts

******* This chapter/section is at work *******

3.7.1 Dealing with Multiple Requirements

******* This chapter/section is at work *******

For example, a concept checking for the requirements of iterators might look as follows:[8]

```
template<typename I>
concept IsIterator =
  default_initializable<T> &&
  std::copyable<I> &&
  requires(I i) {
    typename iter_difference_t<I>;
    *i;
    { ++i } -> same_as<I&>;
    { i++ } -> same_as<I>;
};
```

3.7.2 Concepts versus Traits and Expressions

Concepts are more than just expressions that evaluate Boolean results at compile time. You should usually prefer them over type traits and other compile-time expressions.

[8] The requirements in the C++ standard are even a bit more complicated.

Standard Concepts Support Overload Resolution Better

Compilers are for example able to detect whether a concept is part of another concept.

Consider the following example, where we overload a function `foo()` with two requirements defined as type traits:

```
template<typename T, typename U>
requires std::is_same_v<T, U>              // using traits
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::is_same_v<T, U> && std::is_integral_v<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // ERROR: ambiguity: both requirements are true
```

The problem is that if both requirements evaluate to `true` both overloads fit and there is no rule that one of them has priority over the other. Therefore, the compiler stops compiling will an ambiguity error.

If we use the corresponding concepts instead, the compiler finds out that the second requirement is a specialization and prefers it if both requirements are met:

```
template<typename T, typename U>
requires std::same_as<T, U>                // using concepts
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<T, U> && std::integral<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

Define Concepts with Care

To detect sub-concepts, the compiler has to be able detect that concepts and the way they are called match. This requires careful definition of concepts. For example, consider we define our own concept `SameAs` and use it as follows:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;        // define SameAs

template<typename T, typename U>
requires SameAs<T, U>                         // use SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<T, U> && std::integral<T>     // use SameAs again
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

However, note what happens if we exchange the parameters when we call `SameAs<>` in the requirement for the second `foo()`:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;        // define SameAs

template<typename T, typename U>
requires SameAs<T, U>                         // use SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T>     // use SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // ERROR: ambiguity: both true but no sub-concept
```

The problem is that the compiler cannot detect that `SameAs<>` is commutative. The order might matter and therefore the first requirement is not necessarily a subset of the second requirement.

However, we can solve that problem by making clear that for `SameAs<>` the order of the elements does not matter. This requires a helper concept:

```
template<typename T, typename U>
concept SameAsHelper = std::is_same_v<T, U>;

template<typename T, typename U>
concept SameAs = SameAsHelper<T, U> && SameAsHelper<U, T>;      // commutative
```

Now, the order of the parameters does no longer matter for `IsSame<>`:

```
template<typename T, typename U>
requires SameAs<T, U>                           // use SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T>     // use SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

The compiler can find out that the first building block `SameAs<U,T>` is part of a sub-concept of the definition of `SameAs` so that the other building blocks `SameAs<T,U>` and `std::integral<T>` are an extension. Therefore, the second `foo()` now has preference.

The C++ standard knows problems like this and therefore defines the concept `std::same_as` accordingly, so that the order or the elements does not matter:

```
template<typename T, typename U>
requires std::same_as<T, U>                          // use same_as<>
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<U, T> && std::integral<T>    // use same_as<> with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}
```

```
foo(1, 2);    // OK: second foo() preferred
```

3.7.3 When to Use `if constexpr`

******* This chapter/section is at work *******

C++17 introduced a compile-time if which allows us to switch between code depending on certain compile-time conditions.

For example:

```
template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)      // for floating-point value types
{
  if constexpr(std::is_floating_point_v<T>) {
    ...   // special stuff for floating-point values
  }
  coll.push_back(val);
}
```

To behave different when using template parameters, this can be more readable than providing overloaded or specialized templates. However, you cannot use `if constexpr` to provide different API's, to allow others to add other overloads or specializations later on, or to disable template at all,

3.8 Other Stuff of Concepts

******* This chapter/section is at work *******

restrict constructors
requirements for static polymorphism
requirements for ranges

3.9 Afternotes

Since C++98, C++ language designers have explored how to constrain the parameters of templates with concepts. There were multiple approaches to introduce concepts in the C++ programming language (e.g., see http://wg21.link/n1510 by Bjarne Stroustrup). However, the C++ standards committee could not agree on an appropriate mechanism (there was even a very rich concept system adopted for the C++11 working draft, which was later dropped).

The finally accepted wording was formulated by Andrew Sutton in http://wg21.link/p0734r0.

******* This chapter/section is at work *******

Chapter 4

Standard Concepts in Detail

This chapter describes all *concepts* of the C++20 standard library in detail.

4.1 Overview of all Standard Concepts

Table *Basic Concepts for Types and Objects* lists the basic concepts for types and objects in general.

Table *Concepts for Ranges, Iterators, and Algorithms* lists the concepts for ranges, iterators, and algorithms. to create views and the corresponding adaptors.

Table *Auxiliary Concepts* lists the concepts that are mainly used a building blocks for other concepts and usually not directly used by application programmers.

4.1.1 Header Files and Namespaces

Almost all concepts are defined in header `<concepts>`, which is included by `<ranges>` and `<iterator>`. The only exceptions are

- `three_way_comparable` concepts, which are defined in `<compare>` (which is included by almost every other header file)
- `uniform_random_bit_generator`, which is defined in `<random>`

Almost all concepts are defined in namespace `std`. The only exception are the ranges concepts, which are defined in namespace `std::ranges`.

Concept	Constraint
integral	integral type
signed_integral	signed integral type
unsigned_integral	unsigned integral type
floating_point	floating-point type
movable	Supports move initialization/assignment and swaps
copyable	Supports move and copy initialization/assignment and swaps
semiregular	Supports default initialization, copies, moves, and swaps
regular	Supports default initialization, copies, moves, swaps, and equality comparisons
same_as	Same types
convertible_to	Type convertible to another
derived_from	Type convertible from another
constructible_from	Type constructible from others
assignable_from	Type assignable from another
swappable_with	Type swappable with another
common_with	Two types have a common type
common_reference_with	Two types have a common reference type
equality_comparable	Type supports checks for equality
equality_comparable_with	Can check two types for equality
totally_ordered	Types supports a strict weak ordering
totally_ordered_with	Can check two types for strict weak ordering
three_way_comparable	Can apply all comparison operators (including operator <=>)
three_way_comparable_with	Can compare two types with all comparison operators (including operator <=>)
invocable	Can call a callable type with specified arguments
regular_invocable	Can call a stateless callable type with specified arguments
predicate	Can call a predicate with specified arguments
relation	A callable type defines a relation between two types
equivalence_relation	A callable type defines an equality relation between two types
strict_weak_order	A callable type defines an ordering relation between two types
uniform_random_bit_generator	A callable type can be use as random-number generator

Table 4.1. Basic Concepts for Types and Objects

Concept	Constraint
destructible	type is destructible
default_initializable	type is default initializable
move_constructible	Type supports move initializations
copy_constructible	Type supports copy initializations
swappable	Type is swappable
weakly_incrementable	Type supports the increment operators
incrementable	Type supports the increment operators with preserving equality

Table 4.2. Auxiliary Concepts

Concept	Constraint
range	Type is a range
output_range	Type is a range to write to
input_range	Type is a range to read from
forward_range	Type is a range to read from multiple times
bidirectional_range	Type is a range to read forward and backwards from
random_access_range	Type is a range that support jumping around over elements
contiguous_range	Type is a range with elements in contiguous memory
sized_range	Type is a range with cheap size support
common_range	Type is a range with iterators and sentinels having the same type
borrowed_range	Type is a borrowed range
view	Type is a view
viewable_range	Type is or can be converted to a view
indirectly_writable	Type can be used to write to where it refers
indirectly_readable	Type can be used to read from where it refers
indirectly_movable	Type refers to movable objects
indirectly_movable_storable	Type refers to movable objects with support for temporaries
indirectly_copyable	Type refers to copyable objects
indirectly_copyable_storable	Type refers to copyable objects with support for temporaries
indirectly_swappable	Type refers to swappable objects
indirectly_comparable	Type refers to comparable objects
input_output_iterator	Type is an iterator
output_iterator	Type is an output iterator
input_iterator	Type is (at least) an input iterator
forward_iterator	Type is (at least) a forward iterator
bidirectional_iterator	Type is (at least) a bidirectional iterator
random_access_iterator	Type is (at least) a random-access iterator
contiguous_iterator	Type is an iterator to elements in contiguous memory
sentinel_for	Type can be used as sentinel for an iterator type
sized_sentinel_for	Type can be used as sentinel for an iterator type with cheap computation of distances
permutable	Type is a (at least) forward iterator that can reorder elements
mergeable	Two types can be used to merge sorted elements to a third type
sortable	A type is sortable (according to a comparison and projection)
indirectly_unary_invocable	Operation can be used to call it with the value of an iterator type
indirectly_regular_unary_invocable	Stateless operation can be used to call it with the value of an iterator type
indirect_unary_predicate	Unary predicate can be used to call it with the value of an an iterator type
indirect_binary_predicate	Binary predicate can be used to call it with the values of two iterators types
indirect_equivalence_relation	Predicate can be used to check two values of the passed iterator(s) for equality
indirect_strict_weak_order	Predicate can be used to order two values of the passed iterator(s)

Table 4.3. Concepts for Ranges, Iterators, and Algorithms

4.2 Language-Related Concepts

This section lists the concepts that apply to objects and types in general.

4.2.1 Arithmetic Concepts

`std::`**`integral`**`<T>`

- guarantees that type *T* is an integral type (including `bool` and all character types).
- Requires:
  - The type trait `std::is_integral_v<T>` yields `true`

`std::`**`signed_integral`**`<T>`

- guarantees that type *T* is a signed integral type (including signed character types, which `char` might be).
- Requires:
  - `std::integral<T>` is satisfied
  - The type trait `std::is_signed_v<T>` yields `true`

`std::`**`unsigned_integral`**`<T>`

- guarantees that type *T* is a signed integral type (including `bool` and unsigned character types, which `char` might be).
- Requires:
  - `std::integral<T>` is satisfied
  - `std::signed_integral<T>` is not satisfied

`std::`**`floating_point`**`<T>`

- guarantees that type *T* is a floating point type (`float`, `double`, or `long double`).
- The concept was introduced to be able to define the mathematical constants.
- Requires:
  - The type trait `std::is_floating_point_v<T>` yields `true`

4.2.2 Object Concepts

For objects (types that are neither references, functions, or `void`) there is a hierarchy of required basic operations.

`std::`**`movable`**`<T>`

- guarantees that type *T* is movable and swappable. That is, you can move construct, move assign, and swap with another object of the type.
- Requires:
  - `std::move_constructible<T>` is satisfied
  - `std::assignable_from<T&, T>` is satisfied

  - `std::swappable<T>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`copyable`**`<T>`

- guarantees that type *T* is copyable (which implies movable and swappable).
- Requires:
  - `std::movable<T>` is satisfied
  - `std::copy_constructible<T>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`.
  - `std::swappable<T>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`semiregular`**`<T>`

- guarantees that type *T* is semiregular (can default initialize, copy, move, and swap).
- Requires:
  - `std::copyable<T>` is satisfied
  - `std::default_initializable<T>` is satisfied
  - `std::movable<T>` is satisfied
  - `std::copy_constructible<T>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`.
  - `std::swappable<T>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`regular`**`<T>`

- guarantees that type *T* is regular. (can default initialize, copy, move, swap, and check for equality).
- Requires:
  - `std::semiregular<T>` is satisfied
  - `std::equality_comparable<T>` is satisfied
  - `std::copyable<T>` is satisfied
  - `std::default_initializable<T>` is satisfied
  - `std::movable<T>` is satisfied
  - `std::copy_constructible<T>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`.
  - `std::swappable<T>` is satisfied
  - *T* is neither a reference nor a function nor `void`

4.2.3 Concepts for Relations between Types

`std::`**`same_as`**`<T1, T2>`

- guarantees that the types *T1* and *T2* are the same.

- The concept calls the `std::is_same_v` type trait twice to ensure that the order of the parameters does not matter.
- Requires:
  - The type trait `std::is_same_v<`*T1*`,` *T2*`>` yields `true`
  - The order of *T1* and *T2* does not matter.

`std::`**`convertible_to`**`<`*From*`,` *To*`>`

- guarantees that objects of type *From* are both implicitly and explicitly convertible to objects of type *To*.
- Requires:
  - The type trait `std::is_convertible_v<`*From*`,` *To*`>` yields `true`
  - A `static_cast` from *From* to *To* is supported
  - You can return an object of type *From* as a *To*

`std::`**`derived_from`**`<`*D*`,` *B*`>`

- guarantees that type *D* is publicly derived from type *B* (or *D* and *B* are the same) so that any pointer of type *D* can be converted to a pointer of type *B*. In other words: the concept guarantees that a *D* reference/pointer can be used as a *B* reference/pointer.
- Requires:
  - The type trait `std::is_base_of_v<`*B*`,` *D*`>` yields `true`
  - The type trait `std::is_convertible_v` for a `const` pointer of type *D* to *B* yields `true`

`std::`**`constructible_from`**`<`*T*`,` *Args...*`>`

- guarantees that you can initialize object of types *T* with parameters of types *Args...*
- Requires:
  - `std::destructible<`*T*`>` is satisfied
  - The type trait `std::is_constructible_v<`*T*`,` *Args...*`>` yields true

`std::`**`assignable_from`**`<`*To*`,` *From*`>`

- guarantees that you can move or copy assign a value of type *From* to a value of type *To*.
  The assignment also has to yield the original *To* object.
- Requires:
  - *To* has to be an lvalue reference
  - `std::common_reference_with<`*To*`,` *From*`>` is satisfied for `const` lvalue references of the types
  - Operator = has to be supported and yield the same type as *To*

`std::`**`swappable_with`**`<`*T1*`,` *T2*`>`

- guarantees that values of types *T1* and *T2* can be swapped.
- Requires:
  - `std::common_reference_with<`*T1*`,` *T2*`>` is satisfied
  - Any two objects of types *T1* and *T2* can swap values with each other using `std::ranges::swap()`.

`std::`**`common_with`**`<`*T1*`,` *T2*`>`

- guarantees that types *T1* and *T2* share a common type to which they can be explicitly converted.
- Requires:
  - The type trait `std::common_type_t<`*T1*`, `*T2*`>` yields a type
  - A `static_cast` is supported to their common type
  - References of both types share a `common_reference` type
  - The order of *T1* and *T2* does not matter.

`std::`**`common_reference_with`**`<`*T1*`, `*T1*`>`

- guarantees that types *T1* and *T2* share a `common_reference` type to which they can be explicitly and implicitly converted.
- Requires:
  - The type trait `std::common_reference_t<`*T1*`, `*T2*`>` yields a type
  - Both type are `std::convertible_to` their common reference type
  - The order of *T1* and *T2* does not matter.

4.2.4 Comparison Concepts

`std::`**`equality_comparable`**`<`*T*`>`

- guarantees that objects of type *T* are comparable with operators `==` and `!=`. The order should not matter.
- Operator `==` for *T* should be symmetric and transitive:
  - `t1==t2` is `true` if and only if `t2==t1`.
  - If `t1==t2` and `t2==t3` are `true`, then `t1==t3` are `true`.

  However, this a semantic constraint that cannot be checked at compile-time.
- Requires:
  - Both `==` and `!=` are supported and yield a value convertible to `bool`.

`std::`**`equality_comparable_with`**`<`*T1*`, `*T2*`>`

- guarantees that objects of types *T1* and *T2* are comparable using operators `==` and `!=`.
- Requires:
  - `==` and `!=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`.

`std::`**`totally_ordered`**`<`*T*`>`

- guarantees that objects of type *T* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` so that two values are always either, equal, or less or greater.
- The values of *T* should have a total or weak order. However, this a semantic constraint that cannot be checked at compile-time.
- Requires:
  - `std::equality_comparable<`*T*`>` is satisfied
  - All comparisons with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`.

`std::totally_ordered_with<T1, T2>`

- guarantees that objects of types *T1* and *T2* are comparable operators `==`, `!=`, `<`, `<=`, `>`, and `>=` so that two values are always either equal or less or greater.
- Requires:
  - `==` and `!=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`.
  - `==`, `!=`, `<`, `<=`, `>`, and `>=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`.

`std::three_way_comparable<T>`
`std::three_way_comparable<T, Cat>`

- guarantees that objects of type *T* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and operator `<=>` (and have at least the comparison category type *Cat*). If no *Cat* is passed `std::partial_ordering` is required.
- This concept is defined in header file `<compare>`
- Note that this concept does not imply and subsume `std::equality_comparable` because the latter requires that operator `==` yields true only for two objects that are equal. With a weak or partial order this might not be the case.
- Note that this concept does now imply and subsume std::totally_ordered because the latter requires that the comparison category is `std::strong_ordering` or `std::weak_ordering`.
- Requires:
  - All comparisons with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`.
  - Any comparison with operators `<=>` yield a comparison category (which is at least *Cat*)

`std::three_way_comparable_with<T1, T2>`
`std::three_way_comparable_with<T1, T2, Cat>`

- guarantees that any two objects of types *T1* and *T2* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and operator `<=>` (and have at least the comparison category type *Cat*). If no *Cat* is passed `std::partial_ordering` is required.
- This concept is defined in header file `<compare>`
- Note that this concept does not imply and subsume `std::equality_comparable_with` because the latter requires that operator `==` yields true only for two objects that are equal. With a weak or partial order this might not be the case.
- Note that this concept does now imply and subsume std::totally_ordered_with because the latter requires that the comparison category is `std::strong_ordering` or `std::weak_ordering`.
- Requires:
  - `std::three_way_comparable` is satisfied for values and common references of *T1* and *T2* (and *Cat*)
  - All comparisons with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`.
  - Any comparison with operators `<=>` yield a comparison category (which is at least *Cat*)
  - The order of *T1* and *T2* does not matter.

4.3 Concepts for Iterators and Ranges

This section lists all basic concepts for iterators and ranges, which are useful to use in algorithms and similar functions.

Note that the concepts for ranges are in namespace `std::ranges` instead of `std`.

4.3.1 Concepts for Ranges and Views

Several concepts are defined to require parameters to be (specific) ranges. They correspond with the concepts for iterators.

`std::ranges::`**`range`**`<`*Rg*`>`

- guarantees that *Rg* is a valid range,
- This means that objects of type *Rg* support to iterate over the elements by using `std::ranges::begin()` and `std::begin::end()`.

  This is the case if the range either is an array, or provides `begin()` and `end()` members or can be used with free-standing `begin()` and `end()` functions.
- In addition, for `std::ranges::begin()` and `std::ranges::end()` the following constraints apply:
  - They have to operate in (amortized) constant time.
  - They do do not modify the range.
  - `begin()` yields the same position when called multiple times (unless the range does not provide at least forward iterators).

  That all means that we can iterate over all elements with good performance (even multiple times unless we have pure input iterators).
- Requires:
  - For an objects `rg` of types *Rg* `std::ranges::begin(rg)` is supported and `std::ranges::end(rg)` is supported

`std::ranges::`**`output_range`**`<`*Rg*`, `*T*`>`

- guarantees that *Rg* is a range that provides at least output iterators (iterators you can use to write) that accept values of type *T*.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::output_iterator` is satisfied for the iterator type and `T`

`std::ranges::`**`input_range`**`<`*Rg*`>`

- guarantees that *Rg* is a range that provides at least input iterators (iterators you can use to read).
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::input_iterator` is satisfied for the iterator type

`std::ranges::`**`forward_range`**`<`*Rg*`>`

- guarantees that *Rg* is a range that provides at least forward iterators (iterators you can use to read and write and to iterate over multiple times).
- Requires:
  - `std::input_range<Rg>` is satisfied
  - `std::forward_iterator` is satisfied for the iterator type

`std::ranges::`**`bidirectional_range`**`<Rg>`

- guarantees that *Rg* is a range that provides at least bidirectional iterators (iterators you can use to read and write and to iterator over also backwards).
- Requires:
  - `std::forward_range<Rg>` is satisfied
  - `std::bidirectional_iterator` is satisfied for the iterator type

`std::ranges::`**`random_access_range`**`<Rg>`

- guarantees that *Rg* is a range that provides random-access iterators (iterators you can use to read and write, jump back and forth, and compute the distance).
- Requires:
  - `std::bidirectional_range<Rg>` is satisfied
  - `std::random_access_iterator` is satisfied for the iterator type

`std::ranges::`**`contiguous_range`**`<Rg>`

- guarantees that *Rg* is a range that provides random access iterators with the additional constraint that the elements are stored in contiguous memory.
- Requires:
  - `std::random_access_range<Rg>` is satisfied
  - `std::contiguous_iterator` is satisfied for the iterator type
  - Calling `std::ranges::data()` yields a raw pointer to the first element

`std::ranges::`**`sized_range`**`<Rg>`

- guarantees that *Rg* is a range where the number of elements can be computed as the difference of its begin and end in constant time.
- That is, `std::ranges::size()` is well defined for objects of type *Rg*.
- To signal that a types does not satisfy this concept although it provides `size()`, you can define that `std::disable_sized_range<Rg>` yields `true`.
- Requires:
  - `std::range<Rg>` is satisfied
  - Calling `std::ranges::size()` is supported
  - `std::ranges::disable_sized_range<Rg>` is not defined to yield `true`

`std::ranges::`**`common_range`**`<Rg>`

- guarantees that *Rg* is a range where the begin iterator and the sentinel (end iterator) have the same type.

- The guarantee is always given by:
  - All standard containers (vector, list, etc.)
  - `empty_view`
  - `single_view`
  - `common_view`

  The guarantee is not given by
  - take views
  - `const` drop views
  - iota views with no end value or an end value of different type

  For other views, it depends on the type of the underlying ranges.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::ranges::iterator_t<`*Rg*`>` and `std::ranges::sentinel_t<`*Rg*`>` have the same type.

`std::ranges::`**`borrowed_range`**`<`*Rg*`>`

- guarantees that *Rg* is a *borrowed range*.
- That is, the iterators are not tied to the lifetime of the range. That means that iterators are more safe to use, because they cannot dangle when the range they were created from gets destroyed. However, they can still dangle if the iterators of the range refer to an underlying range and the underlying range is no longer there.
- The guarantee is given if *Rg* is an lvalue (such as an object with a name) or if the variable template `std::ranges::enable_borrowed_range<`*Rg*`>` is `true`, which is the case for the following views: `subrange`, `ref_view`, `string_view`, `span`, `iota_view`, and `empty_view`.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - *Rg* is an lvalue or `enable_borrowed_range<`*Rg*`>` yields `true`

`std::ranges::`**`view`**`<`*Rg*`>`

- guarantees that *Rg* is a view (a range that is cheap to copy, assign, and destroy).
- A view has the following requirements:[1]
  - It has to be a range (support to iterate over the elements).
  - It has to be movable.
  - Move constructor/assignment, copy constructor/assignment (if available), and destructor have to have constant complexity (the time they take does not depend on the number of elements).

  All but the last requirements are checked by corresponding concepts. The last requirement has to be guaranteed by the implementer of a type by deriving publicly from `std::ranges::view_interface` or from `std::ranges::view_base` or by specializing `std::ranges::enable_view<`*Rg*`>` to yield `true`.
- Requires:

[1] Initially C++20 required that views are also default constructible, but that requirement was removed with `http://wg21.link/p2325r3`.

  - `std::range<`*Rg*`>` is satisfied
  - `std::movable<`*Rg*`>` is satisfied
  - The variable template `std::enable_view<`*Rg*`>` is `true`

`std::ranges::`**`viewable_range`**`<`*Rg*`>`

- guarantees that *Rg* is a range that can be safely converted to a view with the `std::views::all()` adaptor.
- The concept is satisfied if *Rg* either is already a view or an lvalue of a range.[2]
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - Either `std::borrowed_range<`*Rg*`>` or `std::view<std::remove_cvref_t<`*Rg*`>>` is satisfied

4.3.2 Concepts for Pointers-Like Objects

This section lists all standard concepts for objects for which you can use operator `*` to deal with a value they point to. This usually applies to raw pointers, smart pointers, and iterators. Therefore, these concepts are usually sued as base constraints for concepts that deal with iterators and algorithms. Note that operator `->` is not required.

`std::`**`indirectly_writable`**`<`*P*`, `*Val*`>`

- guarantees that *P* is a pointer-like object supporting operator `*` to assign a *Val*.
- satisfied by non-const raw pointers, smart pointers, and iterators provided *Val* can be assigned to where *P* refers to.

`std::`**`indirectly_readable`**`<`*P*`>`

- guarantees that *P* is a pointer-like object supporting operator `*` for read access.
- satisfied by raw pointers, smart pointers, and iterators.
- Requires:
  - The resulting value has to the same reference type for both const and non-const objects (which rules out `std::optional<>`). That ensures that the constness of *P* does not propagate to where it points to (which is usually not the case when the operator returns a reference to a member).
  - `std::iter_value_t<`*P*`>` must be valid. The type does ***not*** have to support operator `->`.

Concepts for Indirectly Readable Objects

For pointer-like concept that are indirectly readable you can check additional constraints:

`std::`**`indirectly_movable`**`<`*InP*`,`*OutP*`>`

- guarantees that values of *InP* can be move assigned directly to values of *OutP*.
- With this concept, the following code is valid:

[2] For lvalues of move-only view types there is an issue that `all()` is ill-formed although `viewable_range` is satisfied.

```
void foo(InP inPos, OutP, outPos) {
  *outPos = std::move(*inPos);
}
```

- Requires:
  - `std::indirectly_readable<`*InP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for rvalue references of the values of *InP* to (the values of) *OutP*.

`std::`**`indirectly_movable_storable`**`<`*InP*`,`*OutP*`>`

- guarantees that values of *InP* can be move assigned indirectly to values of *OutP* even when using a (temporary) object of the type to where *OutP* points to.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP, outPos) {
  OutP::value_type tmp = std::move(*inPos);
  *outPos = std::move(tmp);
}
```

- Requires:
  - `std::indirectly_movable<`*InP*`, `*OutP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for the *InP* value to the objects *OutP* refers to.
  - `std::movable` is satisfied for the values *InP* refers to.
  - Rvalues *InP* refers to are copy/move constructible and assignable.

`std::`**`indirectly_copyable`**`<`*InP*`, `*OutP*`>`

- guarantees that values of *InP* can be assigned directly to values of *OutP*.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP outPos) {
  *outPos = *inPos;
}
```

- Requires:
  - `std::indirectly_readable<`*InP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for references of the values of *InP* to (the values of) *OutP*.

`std::`**`indirectly_copyable_storable`**`<`*InP*`, `*OutP*`>`

- guarantees that values of *InP* can be assigned indirectly to values of *OutP* even when using a (temporary) object of the type to where *OutP* points to.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP outPos) {
  OutP::value_type tmp = *inPos;
  *outPos = tmp;
}
```

- Requires:
  - `std::indirectly_copyable<`*InP*`, `*OutP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for `const` lvalue and rvalue references of the *InP* value to the objects *OutP* refers to.
  - `std::copyable` is satisfied for the values *InP* refers to.
  - The values *InP* refers to are copy/move constructible and assignable.

`std::`**`indirectly_swappable`**`<`*P*`>`
`std::`**`indirectly_swappable`**`<`*P1*`, `*P2*`>`

- guarantees that values of *P* or *P1* and *P2* can be swapped (using `std::ranges::iter_swap()`).
- Requires:
  - `std::indirectly_readable<`*P1*`>` (and `std::indirectly_readable<`*P2*`>`) are satisfied.
  - For any two objects of types *P1* and *P2* `std::ranges::iter_swap()` is supported.

`std::`**`indirectly_comparable`**`<`*P1*`, `*P2*`, `*Comp*`>`
`std::`**`indirectly_comparable`**`<`*P1*`, `*P2*`, `*Comp*`, `*Proj1*`>`
`std::`**`indirectly_comparable`**`<`*P1*`, `*P2*`, `*Comp*`, `*Proj1*`, `*Proj2*`>`

- guarantees that you can compare the elements (optionally transformed with *Proj1* and *proj2*) to where *P1* and *P2* refer
- Requires:
  - `std::indirect_binary_predicate` for *Comp*, `std::projected<`*P1*`, `*Proj1*`>`, and `std::projected<`*P2*`, `*Proj1*`>` is satisfied with `std::identity` as default projections

4.3.3 Concepts for Iterators

This section lists the concepts to require different types of iterators. They correspond with the concepts for ranges.

`std::`**`input_output_iterator`**`<`*Pos*`>`

- guarantees that *Pos* supports the basic interface of all iterators: `operator++` and `operator*`, where `operator*` has to refer to a value.

  The concept does not require that the iterator is copyable (thus, this is less than the basic requirements for iterators used by algorithms).
- Requires:
  - `std::weakly_incrementable<`*pos*`>` is satisfied
  - Operator `*` yields a reference

`std::`**`output_iterator`**`<`*Pos*`, `*T*`>`

- guarantees that *Pos* is an output iterator (an iterator where you can assign values to the elements) to which values you can assign values of type*T*.
- Iterators of type *Pos* can be used to assign a value `val` of type*T* with:

```
*i++ = val;
```

- These iterators are only good for single-pass iterations.
- Requires:
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_writable<`*Pos*`, I>` is satisfied

`std::`**`input_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is an input iterator (an iterator where you can read values from the elements).
- These iterators are only good for single-pass iterations if not also `std::forward_iterator` is satisfied.
- Requires:
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::input_iterator_tag`

`std::`**`forward_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is a forward iterator (a reading iterator with which you can iterate forward multiple times over elements).
- Requires:
  - `std::input_iterator<`*Pos*`>` is satisfied
  - `std::incrementable<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::forward_iterator_tag`

`std::`**`bidirectional_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is a bidirectional iterator (a reading iterator with which you can iterate forward and backward multiple times over elements).
- Requires:
  - `std::forward_iterator<`*Pos*`>` is satisfied
  - Support to iterate backward with operator `--`
  - *Pos* has an iterator category derived from `std::bidirectional_iterator_tag`

`std::`**`random_access_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is a random-access iterator (a reading iterator with which you can jump back and force over the elements).
- Requires:
  - `std::bidirectional_iterator<`*Pos*`>` is satisfied
  - `std::totally_ordered<`*Pos*`>` is satisfied
  - `std::sized_sentinel_for<`*Pos*`, `*Pos*`>` is satisfied
  - Support of `+`, `+=`, `-`, `-=`, `[]`
  - *Pos* has an iterator category derived from `std::random_access_iterator_tag` is satisfied

`std::`**`contiguous_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is an iterator iterating over element in contiguous memory.
- Requires:
  - `std::random_access_iterator<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::contiguous_iterator_tag`
  - `to_address()` for an element is a raw pointer to the element

`std::`**`sentinel_for`**`<`*S*`, `*Pos*`>`

- guarantees that *S* can be used as sentinel (end iterator of maybe different type) for *Pos*.
- Requires:
  - `std::semiregular<`*S*`>` is satisfied
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - Can compare *Pos* and *S* with operators `==` and `!=`

`std::`**`sized_sentinel_for`**`<`*S*`, `*Pos*`>`

- guarantees that *S* can be used as sentinel (end iterator of maybe different type) for *Pos* and you can compute the distance between them.
- To signal that that you can compute the distance but that does not have constant complexity, you can define that `std::disable_sized_sentinel_for<`*Rg*`>` yields `true`.
- Requires:
  - `std::sentinel_for<`*S*`, `*Pos*`>` is satisfied
  - Calling operator `-` for a *Pos* and *S* yields a value of the difference type of the iterator
  - `std::disable_sized_sentinel_for<`*S*`, `*Pos*`>` is not defined to yield `true`

4.3.4 Iterator Concepts for Algorithms

`std::`**`permutable`**`<`*Pos*`>`

- guarantees that you iterate forward using operator `++` and reorder elements by moving and swapping them
- Requires:
  - `std::forward_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_movable_storable<`*Pos*`>` is satisfied
  - `std::indirectly_swappable<`*Pos*`>` is satisfied

`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`, `*Proj1*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`, `*Proj1*`, `*Proj2*`>`

- guarantees that you can merge the elements of two sorted sequences to where *Pos1* and *Pos2* refer by copying them into a sequence to where *ToPos* refers. The order is defined by operator `<` or *Comp* (applied to the values optionally transformed with projections *Proj1* and *Proj2*).

- Requires:
  - `std::input_iterator` is satisfied for both *Pos1* and *Pos2*
  - `std::weakly_incrementable<`*ToPos*`>` is satisfied
  - `std::indirectly_copyable<`*PosN*`,` *ToPos*`>` is satisfied for both *Pos1* and *Pos2*
  - `std::indirect_strict_weak_order` of *Comp*, `std::projected<`*Pos1*`,` *Proj1*`>`, and `std::projected<`*Pos2*`,` *Proj2*`>` is satisfied (with `<` as default comparison and `std::identity` as default projections), which implies
    - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
    - `std::copy_constructible<`*Comp*`>` is satisfied
    - `std::strict_weak_order<`*Comp*`>` is satisfied for *Comp*`&` and the (projected) value/reference types

`std::`**`sortable`**`<`*Pos*`>`
`std::`**`sortable`**`<`*Pos*`,` *Comp*`>`
`std::`**`sortable`**`<`*Pos*`,` *Comp*`>`
`std::`**`sortable`**`<`*Pos*`,` *Comp*`,` *Proj*`>`

- guarantees that you can sort the elements iterator *Pos* refers to with operator `<` or *Comp* (after optionally applying the projection *Proj* to the values).
- Requires:
  - `std::permutable<`*Pos*`>` is satisfied
  - `std::indirect_strict_weak_order` of *Comp* and the (projected) values is satisfied (with `<` as default comparison and `std::identity` as default projection), which implies
    - `std::indirectly_readable<`*Pos*`>` is satisfied
    - `std::copy_constructible<`*Comp*`>` is satisfied
    - `std::strict_weak_order<`*Comp*`>` is satisfied for *Comp*`&` and the (projected) value/reference types

4.4 Concepts for Callables

This section lists all concepts for callables

- functions
- functions objects
- lambdas
- pointer to members (with the type of the object as additional parameter)

4.4.1 Basic Concepts for Callables

`std::`**`invocable`**`<`*Op*`,` *ArgTypes...*`>`

- guarantees that *Op* is callable for arguments of types *ArgTypes...*.
- *Op* can be a function, function object, lambda, or pointer-to-member.

- To document that neither the operation nor the passed arguments are modified you can use `std::regular_invocable` instead. However, note that there is only a semantic difference between these two concept, which cannot be checked at compile time. So the concept differences only document the intention.
- For example:

```
struct S {
  int member;
  int mfunc(int);
  void operator()(int i) const;
};

void testCallable()
{
  std::invocable<decltype(testCallable)>       // satisfied
  std::invocable<decltype([](int){}), char>    // satisfied (char converts to int)
  std::invocable<decltype(&S::mfunc), S, int> // satisfied (member-pointer, object, args)
  std::invocable<decltype(&S::member), S>;     // satisfied (member-pointer and object)
  std::invocable<S, int>;                      // satisfied due to operator()
}
```

  Note that as a type constraint, you only have to specify the parameter types:

```
void callWithIntAndString(std::invocable<int, std::string> auto op);
```

  For a complete example, see the use of a lambda as a non-type template parameter.
- Requires:
  - `std::invoke(op, args)` is valid for an `op` of type *Op* and `args` of type *ArgTypes...*

`std::`**`regular_invocable`**`<`*Op*`, `*ArgTypes...*`>`

- guarantees that *Op* is callable for arguments of types *ArgTypes...* and that the call does neither change the state of the passed operation nor of the passed arguments.
- *Op* can be a function, function object, lambda, or pointer-to-member.
- Note that the difference to the concept `std::invocable` is purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::invoke(op, args)` is valid for an `op` of type *Op* and `args` of type *ArgTypes...*

`std::`**`predicate`**`<`*Op*`, `*ArgTypes...*`>`

- guarantees that the callable (function, function object, lambda) *Op* is a predicate when called for the arguments of types *ArgTypes...*.
- Requires:
  - `std::regular_invocable<`*Op*`>` is satisfied
  - All calls of *Op* with *ArgTypes...* yield a value usable as Boolean value and convertible to `bool`

`std::`**`relation`**`<`*Pred*`, `*T1*`, `*T2*`>`

- guarantees that any two objects of types *T1* and *T2* have a binary relationship in that they can be passed as arguments to the binary predicate *Pred*
- Note that the differences to the concepts `std::equivalence_relation` and `std::strict_weak_order` are purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`equivalence_relation`**`<`*Pred*`, `*T1*`, `*T2*`>`

- guarantees that any two objects of types *T1* and *T2* have an equivalence relationship when compared with *Pred*.

  That is, they can be passed as arguments to the binary predicate *Pred* and the relationship is reflexive, symmetric, and transitive.
- Note that the differences to the concepts `std::relation` and `std::strict_weak_order` are purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`strict_weak_order`**`<`*Pred*`, `*T1*`, `*T2*`>`

- guarantees that any two objects of types *T1* and *T2* have a strict total ordering relationship when compared with *Pred*.

  That is, they can be passed as arguments to the binary predicate *Pred* and the relationship is irreflexive and transitive.
- Note that the differences to the concepts `std::relation` and `std::equivalence_relation` are purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`uniform_random_bit_generator`**`<`*Op*`>`

- guarantees that *Op* can be used as a random number generator returning unsigned integral values of (ideally) equal probability
- This concept is defined in header file `<random>`
- Requires:
  - `std::invocable<`*Op*`&>` is satisfied
  - `std::unsigned_integral` is satisfied for the result of the call
  - Expressions *Op*`::min()` and *Op*`::max()` are supported and yield the same type as the generator calls
  - *Op*`::min()` is less than *Op*`::max()`

4.4.2 Concepts for Callables Used by Iterators

`std::`**`indirectly_unary_invocable`**`<`*Op*`, `*Pos*`>`

- guarantees that *Op* can be called with the values *Pos* refers to.
- Note that the difference to the concept `indirectly_regular_unary_invocable` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Op*`>` is satisfied
  - `std::invocable` is satisfied for *Op*`&` and the value and (common) reference type of *Pos*
  - The result of calling *Op* with both a value and a reference have a common reference type.

`std::`**`indirectly_regular_unary_invocable`**`<`*Op*`, `*Pos*`>`

- guarantees that *Op* can be called with the values *Pos* refers to and the call does not change the state of *Op*
- Note that the difference to the concept `std::indirectly_unary_invocable` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Op*`>` is satisfied
  - `std::regular_invocable` is satisfied for *Op*`&` and the value and (common) reference type of *Pos*
  - The result of calling *Op* with both a value and a reference have a common reference type.

`std::`**`indirect_unary_predicate`**`<`*Pred*`, `*Pos*`>`

- guarantees that the unary predicate *Pred* can be called with the values *Pos* refers to.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::predicate` is satisfied for *Pred*`&` and the value and (common) reference type of *Pos*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

`std::`**`indirect_binary_predicate`**`<`*Pred*`, `*Pos1*`, `*Pos2*`>`

- guarantees that the binary predicate *Pred* can be called with the values *Pos1* and *Pos2* refer to.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::predicate` is satisfied for *Pred*`&`, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

`std::`**`indirect_equivalence_relation`**`<`*Pred*`, `*Pos1*`>`

`std::`**`indirect_equivalence_relation`**`<`*Pred*`, `*Pos1*`, `*Pos2*`>`

- guarantees that the binary predicate *Pred* can be called to check whether two values of *Pos1* and *Pos1*/*Pos2* are equivalent
- Note that the difference to the concept `std::indirectly_strict_weak_order` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::equivalence_relation` is satisfied for *Pred*`&`, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

`std::`**`indirect_strict_weak_order`**`<`*Pred*`, `*Pos1*`>`
`std::`**`indirect_strict_weak_order`**`<`*Pred*`, `*Pos1*`, `*Pos2*`>`

- guarantees that the binary predicate *Pred* can be called to check whether two values of *Pos1* and *Pos1*/*Pos2* have a strict total order
- Note that the difference to the concept `std::indirectly_equivalence_relation` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::strict_weak_order` is satisfied for *Pred*`&`, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

4.5 Auxiliary Concepts

This sections describes some standardized concepts that are mainly specified to implement other concepts. You should usually not use them in application code.

4.5.1 Concepts for Specific Type Attributes

`std::`**`destructible`**`<`*T*`>`

- guarantees that objects of type *T* are destructible without throwing an exception.
  Note that even implemented destructors automatically guarantee not to throw unless you explicitly mark them with `noexcept(false)`.
- Requires:
  - The type trait `std::is_nothrow_destructible_v<`*T*`>` yields `true`.

`std::`**`default_initializable`**`<`*T*`>`

- guarantees that the type *T* supports default construction
- Requires:
  - `std::constructible_from<T>` is satisfied
  - `std::destructible<T>` is satisfied
  - `T{};` is valid
  - `T x;` is valid

`std::`**`move_constructible`**`<T>`

- guarantees that objects of type *T* can be initialized with rvalues of their type.
- That means the following operation is valid (although it might copy instead of move):

  ```
  T t2{std::move(t1)}   // for any t1 of type T
  ```

  Afterwards, `t2` shall have the former value of `t1`, which is a semantic constraint that cannot be checked at compile-time.
- Requires:
  - `std::constructible_from<T, T>` is satisfied.
  - `std::convertible<T, T>` is satisfied.
  - `std::destructible<T>` is satisfied.

`std::`**`copy_constructible`**`<T>`

- guarantees that objects of type *T* can be initialized with an lvalues of their type.
- That means the following operation is valid:

  ```
  T t2{t1}   // for any t1 of type T
  ```

  Afterwards, `t2` shall be equal to `t1`, which is a semantic constraint that cannot be checked at compile-time.
- Requires:
  - `std::move_constructible<T>` is satisfied.
  - `std::constructible_from` and `std::convertible_to` are satisfied for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*.
  - `std::destructible<T>` is satisfied.

`std::`**`swappable`**`<T>`

- guarantees that you can swap the values of two objects of type *T*.
- Requires:
  - `std::ranges::swap()` can be called for two objects of type *T*

4.5.2 Concepts for Incrementable Types

`std::`**`weakly_incrementable`**`<T>`

- guarantees that type *T* supports increment operators.

- Note that this concept does ***not*** require that two increments of the same value yield equal results. Therefore, this is a requirement for single-pass iterations only.
- In contrast to `std::incrementable` the concept is also satisfied if
  - the type is not default constructible, not copyable, or not equality comparable
  - the postfix increment might return `void` (or any other type)
  - two increments of the same value might give different results
- Note that the differences to concept `std::incrementable` are purely semantic differences so that for types for which increments yield different results might still technically satisfy the concept `incrementable`. To implement different behavior for this semantic difference, you should use iterator concepts instead.
- Requires:
  - `std::default_initializable<`*T*`>` is satisfied
  - `std::movable<`*T*`>` is satisfied
  - `std::iter_difference_t<`*T*`>` is a valid signed integral type

`std::`**`incrementable`**`<`*T*`>`

- guarantees that type *T* is an incrementable type, so that you can iterate multiple times over the same sequence of values.
- In contrast to `std::weakly_incrementable` the concept
  - requires that two increments of the same value give different results (as it is the case for forward iterators).
  - requires that type *T* is default constructible, copyable, and equality comparable.
  - requires postfix increment to return a copy of the iterator (return type *T*).
- Note that the differences to concept `std::weakly_incrementable` are purely semantic differences so that for types for which increments yield different results might still technically satisfy the concept `incrementable`. To implement different behavior for this semantic difference, you should use iterator concepts instead.
- Requires:
  - `std::weakly_incrementable<`*T*`>` is satisfied
  - `std::regular<`*T*`>` is satisfied so that the type is default constructible, copyable, and equality comparable.

Open

4.6 Afternotes

This page is intentionally left blank

Chapter 5

Ranges and Views

Since the first C++ standard, the way to deal with the elements of containers and other sequences always has been to use iterators for the position of the first element (the *begin*) and the position behind the last element (the *end*). Therefore, algorithms that operate on ranges usually take two parameters to process all elements of a container and containers provide functions like `begin()` and `end()` to provide these parameters.

C++20 provides a new way to deal with ranges. You can specify objects that as a whole represent a *range* (a sequence of elements) and pass these objects to algorithms to process all elements of the sequence as a whole.

The change sounds pretty simple, but as you will see it has a lot of consequences. The way to deal algorithms changes dramatically both for the caller and for the implementor. Therefore, C++20 provides several new features to deal with ranges:

- New implementations of standard algorithms
- Several helper function and new type utilities
- Concepts that specifically deal with ranges
- Special handling for views (lightweight ranges that are cheap to create and use).

This chapter introduces the basic aspects and features of ranges and views. The following chapters discuss details.

5.1 A Tour of Ranges by Example

Let us understand ranges by looking at a few examples using them.

5.1.1 Passing Containers to Algorithms as Ranges

Since C++98, we are used to iterate over half-open ranges when dealing with collections of elements. By passing the begin and the end of a range (often coming from the `begin()` and `end()` member functions of a container), you could specify which elements have to be processed:

```
#include <vector>
#include <algorithm>
```

```
std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

std::sort(coll.begin(), coll.end());   // sort all elements of the collection
```

C++20 introduces the concept of a ***range***, which is an abstract description of a single object representing a sequence of values. Each container can be use as such a range and therefore you can now pass containers as a whole to algorithms:

```
#include <vector>
#include <algorithm>
#include <ranges>

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

std::ranges::sort(coll);               // sort all elements of the collection
```

Here, we pass the vector `coll` to the range algorithm `sort()` to sort all elements in the vector.

A few things are already remarkable in this little example:

- To deal with ranges you need the header file `<ranges>`.
- Almost all features of the ranges library are defined in the namespace `std::ranges`.

A new namespace was necessary, because we introduced several new API's and we wanted to keep the existing names (such as the name of the algorithm `sort()`). To avoid ambiguities and other conflicts with existing API's, the overloaded functionality was introduced in the sub-namespace `ranges`.

It is pretty common to introduce a shortcut for the namespace `std::ranges`, such as `rng`. So, in practice the code above might look as follows

```
#include <vector>
#include <algorithm>
#include <ranges>
namespace rng = std::ranges;           // define shortcut for std::ranges

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

rng::sort(coll);                       // sort all elements of the collection
```

In this book I might from time-to-time use namespace `rng` if otherwise symbols and code size becomes too large.

5.1.2 Algorithms with Requirements

The new standard algorithms for ranges declare range parameters as template parameters. This is necessary because ranges do not have a specific type (or are derived from a common base class). To specify and validate the necessary requirements when dealing with a range, C++20 introduces several concepts for ranges.

Consider, for example, the `sort()` algorithm for ranges. In principle, it is defined as follows (a few details are missing which we introduce later):

```
template<std::ranges::random_access_range R,
```

```
          typename Comp = std::ranges::less>
  requires std::sortable<std::ranges::iterator_t<R>, Comp>
  ... sort(R&& r, Comp comp = {});
```

This declaration already has multiple new ranges features:

- Two standard concepts specify requirements for the passed range `R`:
  - Concept `std::ranges::random_access_range` requires that `R` is a range that provides random-access iterators (iterators you can use to read and write, jump back and forth and compute the distance). The concept includes (subsumes) the basic concept for ranges: `std::range`, which requires that the passed argument is either an array or provides `begin()` and `end()` as member or free-standing functions.
  - Concept `std::sortable` requires that the elements in the range `R` are sortable with the sort criterion `Comp`.
- The new type utility `std::ranges::iterator_t` is used to pass the iterator type to `sortable`.
- As default comparison criterion `Comp` `std::ranges::less` is used, which defines that the sort algorithm sorts with operator `<`. `std::ranges::less` is kind of a concept-constrained `std::less` , which ensures that all comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`) are supported and the values have a total order.

That means, you can pass any range with random access iterators and sortable elements:

lib/rangessort.cpp

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

void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main(int argc, char** argv)
{
  std::vector<std::string> coll{"Rio", "Tokyo", "New York", "Berlin"};

  std::ranges::sort(coll);      // sort elements
  std::ranges::sort(coll[0]);   // sort character in first element
  print(coll);

  int arr[] = {42, 0, 8, 15, 7};
  std::ranges::sort(arr);       // sort values in array
  print(arr);
}
```

The program has the following output:

```
Beilnr New York Rio Tokyo
0 7 8 15 42
```

If you pass a container/range that has no random access iterators, you get an error message that the concept `std::ranges::random_access_range` is not satisfied:

```
std::list<std::string> coll2{"New York", "Rio", "Tokyo"};
std::ranges::sort(coll2);   // ERROR: random_access_range not satisfied
```

If you pass a container/range where you cannot compare the elements with operator `<`, `std::sortable` is not satisfied:

```
std::vector<std::complex<double>> coll3;
std::ranges::sort(coll3);   // ERROR: sortable<> not satisfied
```

5.1.3 Views

To deal with ranges, C++20 also introduces ***views***. Views are lightweight ranges that

- can refer to ranges and subranges
- can iterate over the elements of range by filtering out some elements or performing some transformations of their values
- can represent a sequence of values themselves

For example, you can use a view to iterate only over the first 5 elements of a range:

```
for (const auto& elem : std::views::take(coll, 5)) {
  ...
}
```

Here, we use the *range adaptor* `std::views::take()`. Range adaptors are helper functions defined in namespace `std::views` that creates views. `take()` creates a view to the first *n* elements of a passed range (if any). So with

```
std::views::take(coll, 5)
```

we pass a view to `coll` that ends with its sixth element (or the last element if there are fewer elements). The view provides the usual API of range so that `begin()`, `end()`, and operator `++` can be used to iterate over the elements and operators `*` and `->` can be used to deal with the values.

Such a view can also be used to sort only the first 5 elements:

```
std::ranges::sort(std::views::take(coll, 5));   // sort the first 5 elements of coll
```

Views can also generate a sequence of values themselves. For example, with the iota view, we can iterator over all values from 1 to 10 as follows:

```
for (int val : std::views::iota(1, 11)) {   // iterate from 1 to 10
  ...
}
```

Internally, adaptors use view types such as `std::ranges::take_view` or `std::ranges::iota_view`, which you could also use directly. However, the adaptors should be preferred because they are often smarter

and enable optimizations. For example, `take()` might yield a `string_view` instead of a `take_view` if the collection passed is already a `string_view`.

Combining Ranges and Views

Views that are not generating values can be used as building blocks.

Assume, you want to use the following three views:

```
// view with elements of coll that are multiples of 3:
std::views::filter(coll, [] (auto elem) {
                           return elem % 3 == 0;
                         })

// view with squared elements of coll:
std::views::transform(coll, [] (auto elem) {
                              return elem * elem;
                            })

// view with first 3 elements of coll:
std::views::take(coll, 3)
```

Because a view is a range, you can use a view as argument of another view:

```
// view with first 3 squared values of of the elements in coll that are multiples of 3:
auto v = std::views::take(
           std::views::transform(
             std::views::filter(coll,
                                [] (auto elem) { return elem % 3 == 0; }),
             [] (auto elem) { return elem * elem; }),
           3);
```

This nesting is hard to read and maintain. However, there is another way to combine ranges and views: views enable using operator | to pipe in the underlying range that should be processed:

```
// view with first 3 squared values of of the elements in coll that are multiples of 3:
auto v = coll
         | std::views::filter([] (auto elem) { return elem % 3 == 0; })
         | std::views::transform([] (auto elem) { return elem * elem; })
         | std::views::take(3);
```

This *pipeline* of ranges and views is easy to define and to understand.

For a collection such as

```
std::vector coll{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

the output would be:

```
9 36 81
```

Here is another full program that demonstrates the composability of view with the pipeline syntax:

lib/viewspipe.cpp

```
#include <iostream>
#include <vector>
#include <map>
#include <ranges>

int main()
{
  namespace vws = std::views;

  std::vector<int> vals{ 0, 1, 2, 3, 4 };

  // map of composers names and their years of birth:
  std::map<std::string, int> composers{
    {"Bach", 1685},
    {"Mozart", 1756},
    {"Beethoven", 1770},
    {"Tchaikovsky", 1840},
    {"Chopin", 1810},
    {"Vivaldi ", 1678},
  };

  // iterate over the names of the first 3 composers born since 1700:
  for (const auto& elem : composers
                        | vws::filter([](const auto& y) {   // since 1700
                                        return y.second >= 1700;
                                      })
                        | vws::take(3)                      // first 3
                        | vws::keys                         // keys/names only
                        ) {
    std::cout << "- " << elem << '\n';
  }
}
```

In this example, we apply a couple of views to a map of composers, where the elements have their name and their year of birth (note that we introduce `vws` as shortcut for `std::views`):

```
std::map<std::string, int> composers{ ... };
...
composers
 | vws::filter(...)
 | vws::take(3)
 | vws::keys
```

The composing pipeline is passed directly to the range-based `for` loop and produces the following output (remember that the elements are sorted according to their keys/names):

```
- Beethoven
- Chopin
- Mozart
```

Do not use a `using` declaration to skip even writing `vws::`:

```
using namespace std::views;    // NO good idea
```

You can easily run into problems as I did when writing this book, when I tried an example where I named a collection just `values` and used the adaptor `values()` instead of `keys()`.

5.1.4 Sentinels

To deal with ranges, we have to introduce a new term, ***sentinels***, which represent the end of a range.

In programming, a *sentinel* is a special value that marks an end or termination. Typical examples are

- The null terminator `'\0'` as the end of a character sequence (e.g., used in string literals)
- `nullptr` marking the end of a linked list
- `-1` to mark the end of a list of non-negative integers

In the ranges library, sentinels define the **end of a range**. In the traditional approach of the STL, sentinels would be the *end iterators*, which usually has the same type as the iterators iterating over a collection. However, with C++20, this is no longer necessary.

The requirement that end iterators have to have the same type as the iterators defining the begin of a range and used to iterate over the elements causes some drawbacks. Creating an end iterator may be expensive or might not even be possible:

- If we want to use a C string or string literal as a range, we first have to compute the end iterator by iterating over the characters until we find `'\0'`. Thus, before we use the string as a range, we already have done the first iteration. Dealing then with all the character would need a second iteration.
- In general, this principle applies if we define define the end of a range by a certain value. If we need an end iterator to deal with the range, we first have to iterate over the while range to find its end.
- Sometimes iterating twice (once to find the end and then once to process the elements of the range) is not possible. This applies to ranges that use pure input iterators, such as using input streams where we read from as ranges. To compute the end of the input (may be *EOF*), we already have to read the input. Reading the input again is not possible or will yield different values.

The generalization of end iterators as sentinels solves this dilemma. C++20 ranges support sentinels (end iterators) of different type. They can signal "until `'\0'`", "until EOF", or until any other value. They can even signal "there is no end" to define endless ranges and "hey, iterating iterator, check yourself whether there is the end."

Note that before C++20, we could also have these kind of sentinels, but they were required to have the same type as the iterators. One example were *input stream iterators*: A default constructed `istream_iterator<>` was used to create an *end-of-stream iterator* so that you can process input from a stream with an algorithm until end-of-file or an error occurs:

```
// print all int's read from standard input:
std::for_each(std::istream_iterator<int>{std::cin},   // read int's from cin
              std::istream_iterator<int>{},           // end is an end-of-file iterator
              [] (int val) {
```

```
    std::cout << val << '\n';
  });
```

So, by relaxing the requirement that sentinels (end iterators) now have to have the same type as iterating iterators we gain a couple of benefits:

- We may skip the need to find the end before we start to process (we process the values and find the end together while iterating).
- For the end iterator, we can sues types that disable operations that cause undefined behavior (such as calling operator `*`, because there is no value at the end). We can use that to signal an error at compile time when we try to dereference an end iterator.
- Defining an end iterator gets easier.

Let us look at a simple example of using a sentinel of different type to iterator over a "range" where the types of the iterators differ:

lib/sentinel1.cpp

```
#include <iostream>
#include <compare>
#include <algorithm>  // for for_each()

struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';  // end is where iterator points to '\0'
  }
};

int main()
{
  const char* rawString = "hello world";

  // iterate over the range of the begin of rawString and its end:
  for (auto pos = rawString; pos != NullTerm{}; ++pos) {
    std::cout << ' ' << *pos;
  }
  std::cout << '\n';

  // call range algorithm with iterator and sentinel:
  std::ranges::for_each(rawString,    // begin of range
                        NullTerm{},   // end is null terminator
                        [] (char c) {
                          std::cout << ' ' << c;
                        });
  std::cout << '\n';
}
```

The program has the following output:

```
h e l l o   w o r l d
h e l l o   w o r l d
```

In the program, we first define an end iterator that defines the end as having a value equal to `'\0'`:

```
struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';   // end is where iterator points to '\0'
  }
};
```

Note that we combine a few other new features of C++20 here:

- When defining `operator==`, we use `auto` as parameter type. That way we enable operator `==` to compare with an object `pos` of arbitrary type (provided comparing its value with a character in the body is valid).
- We only define operator `==` as a member function although algorithms usually compare iterators with sentinels with operator `!=`. Here we benefit from the fact that C++20 now can map operator `!=` to operator `==` with arbitrary order of the operands.

Using Sentinels Directly

Then we use a basic loop to iterate over the "range" of characters of the string `rawString`:

```
for (auto pos = rawString; pos != NullTerm{}; ++pos) {
  std::cout << ' ' << *pos;
}
```

We initialize `pos` as iterator iterating over the characters and printing out their values with `*pos`. The loop runs while its comparison with `NullTerm{}` yields that the value of `pos` does not equal `'\0'`. That is, `NullTerm{}` serves as end iterator. It does not have the same type as `pos` but it supports a comparison with `pos` in a way that it checks the current value `pos` refers to.

Here you can see how sentinels are the generalization of end iterators. They may have a different type than the iterator iterating over the elements, but support comparisons with it to decide whether we are at the end of a range.

Passing Sentinels to Algorithms

C++20 provides overloads for algorithm that no longer require that the begin end the end iterator have the same type. However, these overloads are provided in namespace `std::ranges`:

```
std::ranges::for_each(rawString,    // begin of range
                      NullTerm{},   // end is null terminator
                      ...);
```

The algorithms in namespace `std` still requires that begin and end iterator have the same type and cannot be used that way:

```
std::for_each(rawString,            // ERROR: begin and end have different types
              NullTerm{},
              ...);
```

If you have two iterators of the same type, type `std::common_iterator` provides a way to harmonize them for traditional algorithms. This can be useful because numeric algorithms, parallel algorithms, and containers still require to pass iterators of the same type.

5.1.5 Range Definitions with Sentinels and Counts

Ranges may not only be containers or a pair of iterators. Ranges can be defined by:

- A begin iterator and an end iterator of the same type
- A begin iterator and a sentinel (an end marker of a maybe different type)
- A begin iterator and a count
- Arrays

The ranges supports the definition and use of all of these ranges.

First, the algorithms are implemented in a way that the range can be arrays. For example:

```
int rawArray[] = {8, 6, 42, 1, 77};
...
std::ranges::sort(rawArray);   // sort elements in the raw array
```

In addition, there are several utilities to define ranges defined by iterators and sentinels or counts, which is introduced in the following subsections.

Subranges

To define ranges of iterators and sentinels the ranges library provides type `std::ranges::subrange<>`.

Let us look at a simple example of using subranges:

lib/sentinel2.cpp

```
#include <iostream>
#include <compare>
#include <algorithm>  // for for_each()

struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';  // end is where iterator points to '\0'
  }
};

int main()
{
  const char* rawString = "hello world";

  // define a range of a raw string and a null terminator:
  std::ranges::subrange rawStringRange{rawString, NullTerm{}};

  // use the range in an algorithm:
```

```
  std::ranges::for_each(rawStringRange,
                        [] (char c) {
                          std::cout << ' ' << c;
                        });
  std::cout << '\n';

  // range-based for loop also supports iterator/sentinel:
  for (char c : rawStringRange) {
    std::cout << ' ' << c;
  }
  std::cout << '\n';
}
```

As introduced as example of a sentinel we define type `NullTerm` as type for sentinels that check for the null terminator of a string as end of a range.

By using a `std::ranges::subrange`, the program defines a range object representing the begin of the string and the sentinel as its end:

```
std::ranges::subrange rawStringRange{rawString, NullTerm{}};
```

A subrange is *the* generic type that can be used to convert an iterator and a sentinel into a single object representing it as a range. In fact, the range is even a view, which internally just stores the iterator and the sentinel. That means subranges have reference semantics and are cheap to copy them around.

Now, we can pass the range to the new algorithms taking ranges as arguments:

```
std::ranges::for_each(rawStringRange,
                      ...);
```

Note that `begin()` and `end()` of a `subrange` may yield different types, because they just yield what was passed to define the range.

Finally, note that the range-based `for` loop accepts passing ranges where the type of the begin and the end iterator differ (this feature was already introduced with C++17, but with ranges you can really benefit from it):

```
for (char c : std::ranges::subrange{rawString, NullTerm{}}) {
  std::cout << ' ' << c;
}
```

We can make this approach even more generic by defining a class template where you can specify the value that ends a range. Consider the following example:

lib/sentinel3.cpp

```
#include <iostream>
#include <algorithm>

template<auto End>
struct EndValue {
  bool operator== (auto pos) const {
```

```
    return *pos == End;   // end is where iterator points to End
  }
};

int main()
{
  std::vector coll = {42, 8, 0, 15, 7, -1};   // -1 marks the end

  // define a range the begin of coll and the element 7 as end:
  std::ranges::subrange range{coll.begin(), EndValue<7>{}};

  // sort the elements of this range:
  std::ranges::sort(range);

  // print the elements of the range:
  std::ranges::for_each(range,
                        [] (auto val) {
                          std::cout << ' ' << val;
                        });
  std::cout << '\n';

  // print all elements of coll up to -1:
  std::ranges::for_each(coll.begin(), EndValue<-1>{},
                        [] (auto val) {
                          std::cout << ' ' << val;
                        });
  std::cout << '\n';
}
```

Here we define `EndValue<>` as the end iterator, checking for the end passed as a template parameter. `EndValue<7>{}` creates an end iterator where `7` ends the range and `EndValue<-1>{}` creates an end iterator where `-1` ends the range.

The output of the program is as follows:

```
0 8 15 42
0 8 15 42 7
```

You could define a value of any supported non-type template parameter type.

As another example of sentinels, look at `std::unreachable_sentinel`. This is a value C++20 defines to represent the "end" of an endless ranges. It can help to optimize code so that it never compares against the end (because that comparison is useless when it always yields `false`).

Ranges of Begin and a Count

The ranges library provides multiple way to deal with ranges defined as begin and a count.

The most convenient way to create a range by a begin iterator and a count is to use the range adaptor `std::views::counted()`. It creates a cheap view to the first *n* elements of a begin iterator/pointer.

For example:

```
std::vector<int> coll{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto pos5 = std::ranges::find(coll, 5);
if (std::ranges::distance(pos5, coll.end()) >= 3) {
  for (int val : std::views::counted(pos5, 3)) {
    std::cout << val << ' ';
  }
}
```

Here, `std::views::counted(pos5, 3)` creates a view that represents the 3 elements starting with to where `pos5` refers. Note that `counted()` does ***not*** check whether there are elements (passing a count that is too high results in undefined behavior). It is up to the programmer to ensure that the code is valid. Therefore, we check with `std::ranges::distance()` whether there are enough elements (not that this check can be expensive if you have no collection with random-access iterators).

If you know that there is an element with value `5` that has at least two elements behind, you can also write:

```
// if we know there is a 5 and at least 2 elements behind:
for (int val : std::views::counted(std::ranges::find(coll, 5), 3)) {
  std::cout << val << ' ';
}
```

The count may be `0`, which means that the range is empty.

For detail, see the description of `std::views::counted()`.

Note that you should only use `counted()` when you really have an iterator and a count. If you already have a range and want to deal with the first *n* elements only, use `std::views::take()`.

5.1.6 Projections

`sort()` and many other algorithms for ranges usually have an additional optional template parameter, a *projection*:

```
template<std::ranges::random_access_range R,
         typename Comp = std::ranges::less,
         typename Proj = std::identity>
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
... sort(R&& r, Comp comp = {}, Proj proj = {});
```

The optional additional parameter allows you to specify a transformation (*projection*) for each element before the algorithm processes it further.

For example `sort()` allows you to specify a projection for the elements to sort separately from the way the resulting values are compared:

```
std::ranges::sort(coll,
                  std::ranges::less{},    // still compare with <
                  [] (auto val) {         // but use the absolute value
                    return std::abs(val);
                  });
```

This might be more readable or easier to program than:

```
std::ranges::sort(coll,
                  [] (auto val1, auto val2) {
                    return std::abs(val1) < std::abs(val2);
                  });
```

See *lib/rangesproj.cpp* for a complete example.

The default projection is `std::identity()`, which yields a passed argument as is (it is defined as a function object in `<functional>`).

User-defined projections simply have to take one parameter and return a value for the transformed parameter.

As you can see, the requirement that the elements have to be sortable takes the projection into account:

```
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
```

5.1.7 Utilities to Implement Code for Ranges

To make it easy to program against all the different kinds of ranges, the ranges library provides

- **Generic functions** that, for example, yield iterators or the size of a range.
- **Type functions** that, for example, yield the type of an iterator or the type of the elements

Consider we want to implement an algorithm that yields the maximum value of a range:

lib/maxvalue1.hpp

```
#include <ranges>

template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(const Range& rg)
{
  if (std::ranges::empty(rg)) {
    return std::ranges::range_value_t<Range>{};
  }
  auto pos = std::ranges::begin(rg);
  auto max = *pos;
  while (++pos != std::ranges::end(rg)) {
    if (*pos > max) {
      max = *pos;
    }
  }
  return max;
```

```
}
```

Here we use a couple of standard utilities to deal with the range `rg`

- The concept `std::ranges::input_range` to require that the passed parameter is a range we can read from.
- The type function `std::ranges::range_value_t` that yields the type of the elements in the range.
- The helper function `std::ranges::empty()` that yields whether the range is empty.
- The helper function `std::ranges::begin()` that yields an iterator to the first element (if any).
- The helper function `std::ranges::end()` that yields a *sentinel* (end iterator) of the range.

With the help of these type utilities, the algorithm even works for all ranges (including arrays), because the utilities are also defined for them.

For example, `std::ranges::empty()` tries to call a member function `empty()`, a member function `size()`, a free-standing function `size()`, or to check whether a begin and end iterator are the same.

Section *Range Utilities* lists the utilities for ranges in detail.

Note that the generic `maxValue()` function better declares the passed range `rg` as a ***universal reference*** (also called *forwarding reference*), because you cannot iterate over some lightweight ranges (views) when they are `const`:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
```

This is discussed later in detail.

5.1.8 Limitations and Drawbacks of Ranges

At least in C++20, ranges also have some major limitations and drawbacks that should be mentioned in a general introduction of them:

- There is not support of ranges for numeric algorithms yet. To pass ranges to them, you have to pass `begin()` and `end()` explicitly:

```
std::ranges::accumulate(cont, 0L);                    // ERROR: not provided
std::accumulate(cont.begin(), cont.end(), 0L);        // OK
```

- There is not support of ranges for parallel algorithms yet.

```
std::ranges::sort(std::execution::par, cont);                  // ERROR: not provided
std::sort(std::execution::par, cont.begin(), cont.end());      // OK
```

- Several traditional API's for begin and end require that they have the same type (containers, algorithms in namespace `std`). You might have to harmonize their types with `std::views::common()` or `std::common_iterator`.
- For some views, you cannot iterate over elements when the views are `const`. Generic code might therefore have to use universal/forwarding references.
- `cbegin()` and `cend()` functions, which were designed to ensure that you cannot (accidentally) modify the element you iterate over are broken for views that refer to non-`const` objects.
- When views refer to containers their propagation of constness is broken.
- Ranges lead to a significant namespace chaos. Consider just the following declaration of `sort()`:

```
template<std::ranges::input_range Rg,
         typename T,
         typename Proj = std::identity>
requires std::indirect_binary_predicate<std::ranges::equal_to,
                                        std::projected<std::ranges::iterator_t<Rg>, Proj>,
                                        const T*>
constexpr std::ranges::borrowed_iterator_t<Rg>
  find(Rg&& rg, const T& value, Proj proj = {});
```

It is really not easy to know, where which namespace has to be used.

In addition, we have some symbols in both namespace `std` and namespace `std::ranges` with slightly different behavior. Here, `equal_to` is such an example. You could also use `std::equal_to`, but in general utilities in `std::ranges` provide better support and are more robust for corner cases.

5.2 Using Views

As introduced, views are lightweight ranges can be used as building blocks to deal with the (modified) values of all or some elements of other ranges and views.

C++20 provides several view types and ***range adaptors*** to create them. For almost each view type there is a corresponding *range adaptor*, which is a function object that allows us to call a function to create the view. The adaptors are defined in namespace `std::views` (which is equivalent to namespace `std::ranges::views`) and usually have the name of the view without the `_view` suffix.

Range and view adaptors provide the easiest way to create views. They can be use to convert a range into a view or yield a view to a range/view where the elements are modified in various ways:

- filter out elements
- modify the values of elements
- change the order of elements
- combine or create sub-ranges

Table *Initial views* lists the standard views and adaptors of C++20 that yield an initial view. They can especially serve as the starting building block in a pipeline of views (either taking a range such as a container as input or generating values themselves).

`std::string_view` is the only view type that did already exist before C++20. All other views were introduced with C++20 and usually end with `_view` (the only view type not ending with `_view` is `std::span`).

Table *Views to Operate on Ranges* lists the standard view and adaptors of C++20 that process ranges and other views. They can serve as building block anywhere in a pipeline of views.

In general, the *adaptors* allow programmers to create views easily and with optimal performance. For this, the type of the view an adaptor creates might depend on its input parameters.

The adaptors `all()`, `counted()`, and `counted()` are described in a special section. All other adaptors are described together with their corresponding view types.

Adaptor	Type	Effect
`all(...)`	varies	Converts a range into a view
`counted(...)`	varies	Creates a view from a begin iterator and a count
	`subrange`	Creates a view from a begin iterator and a sentinel
	`ref_view`	Creates a view that refers to a range (low level)
`iota(`*fst*`)`	`iota_view`	Creates a view with an endless incrementing sequence of values starting with *fst*
`iota(`*fst*`, `*lst*`)`	`iota_view`	Creates a view with an incrementing sequence of values from *fst* up to *lst* (not included)
`single(`*val*`)`	`single_view`	Creates a view with *val* as the only element
`empty<`*type*`>`	`empty_view`	Creates an empty of element type *type*
	`basic_istream_view`	Creates a view that reads elements from an input stream
	`basic_string_view`	Creates a read-only view for a sequence of characters (since C++17)
	`span`	Creates a view to a sequence of elements in contiguous memory

Table 5.1. Initial views

Adaptor	Type	Effect
`take(`*num*`)`	varies	Takes the first *num* elements of a range
`take_while(`*pred*`)`	`take_while_view`	Takes all leading elements that match a predicate
`drop(`*num*`)`	varies	Drops the first *num* elements of a range
`drop_while(`*pred*`)`	`drop_while_view`	Drops all leading elements that match a predicate
`filter(`*pred*`)`	`filter_view`	Filters to use all elements that match a predicate
`transform(`*pred*`)`	`transform_view`	Transforms all elements of a range
`elements<`*idx*`>`	`elements_view`	Use the *idx*th member of each tuple-like element
`keys`	`elements_view`	Use the first member of each tuple-like element
`values`	`elements_view`	Use the second member of each tuple-like element
`reverse`	varies	Reverse the order of elements
`join`	`join_view`	Joins the elements of multiple ranges
`split(`*pred*`)`	`split_view`	Split a range into sub-ranges
`lazy_split(`*pred*`)`	`lazy_split_view`	Split an input/`const` range into sub-ranges
`common`	varies	Harmonizes types of begin iterator and sentinel

Table 5.2. Views to Operate on Ranges

All of the views provide move (and optional copy) operations and a destructor with constant complexity (the time these operations take does not depend on the number of elements).[1] The concept std::ranges::view checks the corresponding requirements.

[1] The original C++20 standard did also require views to have a default constructor. However, that requirement was removed later with `http://wg21.link/P2325R3`

5.2.1 Views from Ranges

Containers and strings are not views because they are not lightweight enough: they own and copy their elements.

To use containers and string as views you should use the `all()` adaptor or a specific view that already performs some modification with the range.

As written, you have different options for this:

- You can pass the container as parameter to an adaptor:
  ```
  auto first4 = std::views::take(coll, 4);
  ```
- You can pipe the container into an adaptor:
  ```
  auto first4 = coll | std::views::take(4);
  ```
- You can pass the container as parameter to the view type:
  ```
  std::ranges::take_view first4{coll, 4};
  ```

All these creations of a view have an important restriction: they do not compile for temporary objects and other rvalues passed as arguments:

```
std::vector<int> getValues();  // forward declaration

std::ranges::take_view take5{getValues(), 5};  // compile-time ERROR
```

This means that you cannot compile code like this:

```
// process first 5 elements of returned collection:
for (const auto& elem : std::ranges::take_view(getValues(), 5)) {  // ERROR
  ...
}
```

or:

```
// process first 5 elements of returned collection:
for (const auto& elem : getValues() | std::views::take(5)) {       // ERROR
  ...
}
```

Instead, you have to initialize a reference with the return value, which by rule extends the lifetime of the return value and is an lvalue (because it has a name):

```
// process first 5 elements of returned collection:
const auto& r = getValues();  // reference extends lifetime of return value
for (const auto& elem : std::views::take(r, 5)) {                  // OK
  ...
}
```

If you want to preserve non-constness (which is necessary to iterating over some views). you have to use a universal/forwarding reference:

```
// process first 5 elements of returned collection:
auto&& r = getValues();       // reference extends lifetime of return value
for (const auto& elem : std::views::take(r, 5)) {                  // OK
```

```
  ...
}
```

However, as long as the range-based `for` loop has the lifetime problem that it internally destroys temporary object when iterating over a reference to it (this problem is described in http://wg21.link/p2276), making code like this not compile is a good thing.

5.2.2 Pipelines for Temporary Ranges

Again, note that views cannot operate on temporary objects (prvalues) that are ranges but not views. This also applies to pipelines:

```
std::vector<int> getValues();  // forward declaration

auto allBut5 = getValues() | std::views::drop(5);  // ERROR
```

This means that you cannot compile code like this:[2]

```
// process first 5 elements of returned collection:
for (const auto& elem : getValues() | std::views::take(5)) {  // ERROR
  ...
}
```

Instead, you have to initialize a reference with the return value, which by rule extends the lifetime of the return value and is an lvalue (because it has a name). You can

```
// process first 5 elements of returned collection:
const auto& r = getValues();  // reference lifetime of return value
for (const auto& elem : r | std::views::take(5)) {            // OK
  ...
}
```

Again, there are two way to do that:

- Initialize a `const` lvalue reference:

```
// process first 5 elements of returned collection:
const auto& r = getValues();       // extend lifetime of return value
for (const auto& elem : r | std::views::take(5)) {            // OK for most views
  ...
}
```

- Initialize a *universal reference* (also called *forwarding reference*):

```
// process first 5 elements of returned collection:
auto&& r = getValues();            // extend lifetime of return value
for (const auto& elem : r | std::views::take(5)) {            // OK for all views
  ...
}
```

[2] As long as the range-based `for` loop still has the lifetime problem described in http://wg21.link/p2276, this is a good thing.

Using universal/forwarding references is the more general approach, because you cannot iterate over the elements of some views, when they are `const`. Even when `getValues()` yields a view that does not support to iterate over elements when it is `const`, the code works.

Often, you anyway need a name for the return value. For example, checking the return value of `find()` accordingly looks as follows:

```
auto&& data = getData();                        // extend lifetime to use it twice
auto pos = std::ranges::find(data, 42);         // yields valid iterator
if (pos != data.end()) {                        // OK
  std::cout << *pos;                            // OK
}
```

5.2.3 Lazy Evaluation

Consider the following program:

lib/filttrans.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };

  // define a view:
  auto vColl = coll
               | vws::filter([] (int i) {
                               std::cout << " filter " << i << '\n';
                               return i % 3 == 0;
                             })
               | vws::transform([] (int i) {
                                  std::cout << "  trans " << i << '\n';
                                  return -i;
                                });

  // and use it:
  std::cout << "*** coll | filter | transform:\n";
  for (int i : vColl) {
    std::cout << "   => " << i << '\n';
  }
}
```

The program has the following output:

```
*** coll | filter | transform:
 filter 8
 filter 15
  trans 15
   => -15
 filter 7
 filter 0
  trans 0
   => 0
 filter 9
  trans 9
   => -9
```

We call `filter()` for each element, and for the elements that were not filtered out we call `transform()`. That is probably what everybody would expect.

However, there is one thing that is remarkable. The whole processing does ***not*** happen when we define the view. It happens after the print statement when we *use* the view. That is, we use *lazy evaluation*. A view is just the description of a processing. The processing is performed element by element when we ask for the next resulting value.

That has a big benefit: We do not start with the processing of elements we never need. For example, consider we use the view to find the first resulting value that is `0`:

```
std::ranges::find(vColl, 0);
```

The output would be then only:

```
filter 8
filter 15
 trans 15
filter 7
filter 0
 trans 0
```

Pipelines of views use a ***pull model***. Whenever a consumer asks for the next value of a view, it gets processed using all adaptors involved. This is one reason why pipelines also work with initial ranges and views yielding an infinite number of elements: we do not compute an unlimited number of values not knowing how many are used; we compute as many values as requested.

5.2.4 Performance Issues with Filters

The pull model used for pipelines also has its drawbacks. To demonstrate that, let us change the order of the two adaptors involved:

lib/transfilt.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
```

```
namespace vws = std::views;

int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };

  // define a view:
  auto vColl = coll
               | vws::transform([] (int i) {
                                  std::cout << " trans: " << i << '\n';
                                  return -i;
                                })
               | vws::filter([] (int i) {
                               std::cout << "  filt: " << i << '\n';
                               return i % 3 == 0;
                             });

  // and use it:
  std::cout << "*** coll | transform | filter:\n";
  for (int i : vColl) {
    std::cout << "   => " << i << '\n';
  }
}
```

Now, the output of the program is as follows:

```
*** coll | transform | filter:
 trans: 8
  filt: -8
 trans: 15
  filt: -15
 trans: 15
   => -15
 trans: 7
  filt: -7
 trans: 0
  filt: 0
 trans: 0
   => 0
 trans: 9
  filt: -9
 trans: 9
   => -9
```

We obviously have additional calls of the adaptors. First, we call `transform()` for each element; not only for those not being filtered out. That is reasonable. We decide which element to filter out *after* its value was transformed. So now we have to call `transform()` for each element.

However, there is an additional problem: For some elements, we call the transformation twice. In fact, for those that are not filtered out.

The reason lies in the nature of a pipeline using the pull model with view objects that can only yield iterators. When we ask view `vColl` for the next element

1. `vColl` asks `filter()` for the next element.
2. `filter()` asks `transform()` for the next element.
3. `transform()` asks `coll` for the next element.
4. `transform()` processes the argument and passes it to `filter()` to process it which can only signal back whether this is an element to use.
5. `vColl` then yields the next element as an iterator. So, when we then process the next element of `vColl`, we need the value again and have to apply the transformation to get it.

That is, with each filter, we have to perform all previous transformations once to *check* the value. If `true`, we have to perform the previous transformations once again to *use* the value. In fact, each filter adds one more call of all previous translations on `true` (and skips all following translations on `false`).

Given the following pipeline of transformations `t2`, `t2`, `t3` and filters `f1`, `f2`:

```
t1 | t2 | f1 | t3 | f2
```

we have the following behavior:

- For elements where `f1` yields `false`, we call:
  `t1 t2 f1`
- For elements where `f1` yields `true` but `f2` yields `false`, we call:
  `t1 t2 f1 t1 t2 t3 f2`
- For elements where `f1` and `f2` yield `true`, we call:
  `t1 t2 f1 t1 t2 t3 f2 t1 t2 t3`

See *`lib/viewscalls.cpp`* for a complete example.

Multiple calls are only a problem for the `filter` view (type `std::ranges::filter_view` and adaptor `std::views::filter`). All other filters like `take` and `drop` stop the whole processing when finding the first element that should not be processed, so that they do not have to process it again.

So, for `filter` views take the following into account:

- Prefer to have it at the beginning of a pipeline.
- Be careful with expensive transformations ahead of filters.

For more details about filters, see the section about the filter view.

5.2.5 Views and Pipelines with Write Access

Views have to be read-only, stateless, and equality preserving. They should never modify a passed argument or call a non-`const` operation for it. That way a view (and all its copies) always always do the same for the same input.

For predicates passed to views such as the `filter_view` or the `transform_view` this means they should either take it by value or by `const` reference. If you modify the argument passed as non-`const` reference, you have undefined behavior

```
vws::transform([] (auto& i) {  // better declare it as const&
                 ++i;          // ERROR: undefined behavior
               })
```

Note that compilers might not check that (because they cannot always do it). Therefore, it is up to you to make it right.

However, it ***is*** supported that you can use views to restrict the collection of elements you want to modify. For example:

```
// assign 0 to all but the first 5 elements of coll:
for (auto& elem : coll | vws::drop(5)) {
  elem = 0;
}
```

5.2.6 Write Access with Filter Views

However, when using filter views there is an important restriction on write access: You have to ensure that the value still fulfills the predicate after the modification.[3] That means the following code is well formed:

```
// modify all even elements:
for (auto& elem : coll | vws::filter(isEven)) {
  elem += 2;    // OK: even values are still even afterwards
}
```

However, the following code results in undefined behavior:

```
// modify all even elements:
for (auto& elem : coll | vws::filter(isEven)) {
  elem += 1;    // ERROR: undefined behavior: filter predicate is broken
}
```

To understand why this is the case, consider the following program:

lib/viewswrite.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

void print(const auto& coll)
{
  std::cout << "coll: ";
  for (int i : coll) {
```

[3] Thanks to Tim Song for pointing this out.

```
    std::cout << i << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<int> coll{1, 4, 7, 10, 13, 16, 19, 22, 25};

  // view for all even elements of coll:
  auto isEven = [] (auto&& i) { return i % 2 == 0; };
  auto evens = coll | vws::filter(isEven);

  print(coll);

  // modify evens:
  for (int& i : evens) {
    std::cout << " increment " << i << '\n';
    //i += 2;  // OK
    i += 1;      // ERROR: undefined behavior because filter predicate is broken
  }
  print(coll);

  // modify evens:
  for (int& i : evens) {
    std::cout << " increment " << i << '\n';
    //i += 2;  // OK
    i += 1;      // ERROR: undefined behavior because filter predicate is broken
  }
  print(coll);
}
```

The program has the following output:

```
coll: 1 4 7 10 13 16 19 22 25
 increment 4
 increment 10
 increment 16
 increment 22
coll: 1 5 7 11 13 17 19 23 25
 increment 5
coll: 1 6 7 11 13 17 19 23 25
```

What the hell is going on here? The first time, we use the view that ensures to only deal with even elements, it works fine. However, the view caches the position of the first element that matches the predicate so that `begin()` does not have to recalculate it. So, when we access the value there the filter does not apply the

predicate again because it knows already that this is the first matching element. Therefore, when we iterate the second time, the filter gives back the former first element. However, for all other elements we have to perform the check again, which means that the filter finds no more elements because they are all odd now.

It was under some discussion whether one-pass write access with a filter should be well-formed even if it breaks the predicate of the filter. Because the requirement that a modification should not violate the predicate of a filter invalidates some very reasonable examples: Here is one of them:[4]

```
for (auto& m : collOfMonsters | filter(isDead)) {
  m.resurrect();      // a shaman's doing, of course
}
```

This code usually compiles and works. However, formally we have undefined behavior again, because the predicate of the filter ("monsters have to be dead") is broken. Any other "modification" of a (dead) monster would be possible, though (e.g., to "burn" them).

To break the predicate, you have to use an ordinary loop instead:

```
for (auto& m : collOfMonsters) {
  if (m.isDead()) {
    m.resurrect();   // a shaman's doing, of course
  }
}
```

5.3 Borrowed Iterators and Ranges

When passing a range as a single argument to an algorithm, you get lifetime problems. This section describes how the ranges library deals with this problem.

5.3.1 Borrowed Iterators

Many algorithms return iterators to the ranges they operate on. However, when passing ranges as a single argument, we can get a new problem that was not possible when a range did require two arguments (for begin end end): if you pass a temporary range (such as a range returned by a function) and return an iterator to it, the returned iterator might become invalid at the end of the statement when the range gets destroyed. Using the returned iterator (or a copy of it) would result in undefined behavior.

For example, consider passing a temporary range to a `find()` algorithm searching for a value in a range:

```
std::vector<int> getData();    // forward declaration

auto pos = find(getData(), 42);                  // returns iterator to temporary vector
// temporary vector returned by getData() is destroyed here
std::cout << *pos;                               // OOPS: using a dangling iterator
```

[4] Thanks to Patrice Roy for this example.

The return value of `getData()` is destroyed with the end of the statement where it is used. Therefore, `pos` refers to an element of a collection that is no longer there. Using `pos` causes undefined behavior (at best, you get a core dump so that you see the problem).

To deal with this problem, the ranges library has introduced the concept of *borrowed iterators*. The lifetime of a borrowed iterator does not depend on the lifetime of the range object it was created from. This is possible if the range the iterator was created from if not used by the iterator because the range refers to a different object that is still alive (another option is that the iterator does refer to no range at all). If you have a borrowed iterator into a range, the iterator is safe to use and does not dangle even when the range is destroyed.[5]

By using type `std::ranges::borrowed_iterator_t<>` algorithms can declare the returned iterator as *borrowed*. That means that the algorithm always returns an iterator that is safe to use. If it could dangle, a special type is used to signal this and convert possible runtime errors into compile-time errors.

For example, `std::ranges::find()` for a single range is declared as follows:

```
template<std::ranges::input_range Rg,
         typename T,
         typename Proj = identity>
...
constexpr std::ranges::borrowed_iterator_t<Rg>
  find(Rg&& r, const T& value, Proj proj = {});
```

By specifying the return type as `std::ranges::borrowed_iterator_t<>` of `Rg`, the standard enables a compile-time check: if the range *R* passed to the algorithm is a temporary object (a prvalue), the return type becomes a *dangling iterator*. In that case the return value is an object of type `std::ranges::dangling`. Any use of such an object (except copying and assignment) result is a compile-time error.

Therefore, the following code results in a compile-time error:

```
std::vector<int> getData();     // forward declaration

auto pos = std::ranges::find(getData(), 42);  // returns iterator to temporary vector
// temporary vector returned by getData() was destroyed
std::cout << *pos;                            // compile-time ERROR
```

To be able to call `find()` for a temporary, you have to pass it as an lvalue. That is, it has to have a name. That way the algorithm ensures that the collection still exists after calling it. And that way you can also check whether a value was found (what is usually appropriate to do anyway).

The best way to returned collection a name is to use references. That they they are not copied. Note that by rule references to temporary objects always extend their lifetime:

```
std::vector<int> getData();                  // forward declaration

reference data = getData();                  // give return value a name to use it as an lvalue
// lifetime of returned temporary vector ends now with destruction of data
...
```

The are two kinds of references you can use here:

[5] For this reason, such iterators were called *safe iterators* in draft versions of the ranges library.

- You can declare a `const` lvalue reference:

```
std::vector<int> getData();                 // forward declaration

const auto& data = getData();               // give return value a name to use it as an lvalue
auto pos = std::ranges::find(coll, 42);     // yields no dangling iterator
if (pos != coll.end()) {
  std::cout << *pos;                        // OK
}
```

  This reference makes the return value `const`, which might not be what you want (note that you cannot iterate over some views when they are `const`; although due to the reference semantics of views you have to be careful when returning them).

- In more generic code you better use a *universal reference* (also called *forwarding reference*) so that you keep the nature non-constness of the return value:

```
... getData();                              // forward declaration

auto&& data = getData();                    // give return value a name to use it as an lvalue
auto pos = std::ranges::find(coll, 42);     // yields no dangling iterator
if (pos != coll.end()) {
  std::cout << *pos;                        // OK
}
```

This feature has the consequence that you cannot pass a temporary object to an algorithm even if the resulting code would be valid:

```
process(std::ranges::find(getData(), 42));  // compile-time ERROR
```

Although the iterator would be valid during the function call (the temporary vector would be destroyed after the call), `find()` returns a `std::ranges::dangling` object.

Again, the best way to deal with this issue is to declare a reference for the return value of `getData()`:

- Using a `const` lvalue reference:

```
const auto& data = getData();               // give return value a name to use it as an lvalue
process(std::ranges::find(data, 42));       // passes a valid iterator to process()
```

- Using a *universal/forwarding reference*:

```
auto&& data = getData();                    // give return value a name to use it as an lvalue
process(std::ranges::find(data, 42));       // passes a valid iterator to process()
```

Remember that often you anyway need a name for the return value to check whether the return value refers to an element and is not the `end()` of a range:

```
auto&& data = getData();                    // give return value a name to use it as an lvalue
auto pos = std::ranges::find(data, 42);     // yields valid iterator
if (pos != data.end()) {                    // OK
  std::cout << *pos;                        // OK
}
```

5.3.2 Borrowed Ranges

Range types can claim that they are *borrowed ranges*. That means that they always return iterators that cannot dangle. When temporary borrowed ranges are used as parameters, algorithms can return iterators to them because the iterators can never dangle.

In general, containers and views are not borrowed ranges. However, we have two kinds of views that *are* borrowed ranges:

- Views where iterators store all information to iterate locally:
  - `std::ranges::iota_view`, generates an incrementing sequence of values. Here the iterator locally stores the current value and does not refer to any other object.
  - `std::ranges::empty_view`, for which any iterator always is at the end so that it cannot iterate over element values at all.
- Views that refer to other ranges so that the iterators directly refer to the underlying ranges:
  - `std::ranges::subrange`
  - `std::ranges::ref_view`
  - `std::span`
  - `std::string_view`

Note that borrowed iterators can still dangle, when they refer to an underlying range (the latter category above) and the underlying range is no longer there.

As a result, we can catch ***some*** but not all possible runtime errors at compile time, which I can demonstrate with the various ways to try to find an element with the value `8` in various ranges (yes, usually we should check, whether an end iterator was returned):

- All lvalues (objects with names) are borrowed ranges so that the returned iterator cannot be `dangling` as long as the iterator exist in the same or a sub-scope of the range.

```
std::vector coll{0, 8, 15};

auto pos0 = std::ranges::find(coll, 8);                    // borrowed range
std::cout << *pos0;                              // OK (runtime error if no 8 found)

auto pos1 = std::ranges::find(std::vector{8}, 8);          // yields dangling
std::cout << *pos1;                              // compile-time ERROR
```

- For temporary views it depends. For example:

```
auto pos2 = std::ranges::find(std::views::single(8), 8);   // yields dangling
std::cout << *pos2;                              // compile-time ERROR

auto pos3 = std::ranges::find(std::views::iota(8), 8);     // borrowed range
std::cout << *pos3;                              // OK (runtime error if no 8 found)

auto pos4 = std::ranges::find(std::views::empty<int>, 8); // borrowed range
std::cout << *pos4;                              // runtime ERROR as no 8 found
```

  For example, single view iterators refer to the value of their element in the view; therefore, single views are no borrowed ranges.

On the other hand, iota view iterators hold copies of the element they refer to so that iota views are declared as borrowed ranges.

- For views that refer to another range (as a whole or to a sub-sequence of it) the situation is more complicated. If they can, they try to detect similar problems. For example, the adaptor `std::views::take()` also checks for prvalues:

```
auto pos5 = std::ranges::find(std::views::take(std::vector{0, 8, 15}, 2), 8);
                                                    // compile-time ERROR
```

Here, calling `take()` is already a compile-time error.

However, if you use `counted()`, which only takes an iterator, it is the task of the programmer to ensure that the iterator is valid:

```
auto pos6 = std::ranges::find(std::views::counted(std::vector{0, 8, 15}.begin(),
                                                  2), 8);
std::cout << *pos6;                          // runtime ERROR even if 8 found
```

The views, which are here created with `counted()`, are by definition borrowed ranges, because they pass their internal references to their iterators. In other words: an iterator of a counted view does not need the view it belongs to. However, it might still happen that the iterator refers to a range that no longer exists (because its view referred to an object that no longer exists). The last row of the example with `pos6` demonstrates this situation. We still get a fatal runtime error even if the value, `find()` is looking for, could be found in the temporary range.

You can check whether a range is a borrowed range with the concept `std::ranges::borrowed_range`. If you implement a container or view, you can signal that it is a borrowed range by specializing the variable template `std::ranges::enable_borrowed_range<>`

5.4 Ranges and `const`

When using views (and the ranges library in general), there are a few thing that are surprising or even broken regarding constness:

- Views remove the propagation of constness to elements.
- Functions like `cbegin()` and `cend()`, which have the goal to ensure that elements are `const` while iterating over them, are either not provided or even broken.
- It might not be possible to iterate over a view when it is `const`.

These drawbacks have to do with the nature of views and some design decisions that were made.

There are fixes under way that at least fix the broken constness of `cbegin()` and `cend()`. Unfortunately, the C++ standards committee decided not to apply them to C++20. These will hopefully might come (and change behavior) with C++23.

5.4.1 Views Remove the Propagation of `const`

Containers have *deep constness*. Because they have value semantics and own their elements, they propagate any constness to their elements. When a containers is `const`, its elements are `const`. As a consequence, the following code fails to compile:

```
template<typename T>
```

```
void constfunc(const T& range)
{
  range.front() += 1;                    // modify an element of a const range
}

std::array<int, 10> coll{};              // array of 10 ints with value 0
...
constfunc(coll);                         // compile-time ERROR
```

This compile-time error is helpful, because it helps to detect code that otherwise would be broken at runtime. For example, if you accidentally use the assignment operator instead of a comparison, the code does not compile:

```
template<typename T>
void constfunc(const T& range)
{
  if (range[0] = 0) {                    // OOPS: should be ==
    ...
  }
}

std::array<int, 10> coll{};              // array of 10 ints with value 0
...
constfunc(coll);                         // compile-time ERROR (thanks goodness)
```

Knowing that elements cannot be modified also helps for optimizations (such as avoiding the need to make backups and check for changes) or to be sure that element access cannot race when multiple threads are used.

Views have reference semantics. They only refer to elements stored somewhere else and usually do not own any elements or values. As a consequence, views do ***not*** propagate constness to their elements. They have *shallow constness*:

```
std::array<int, 10> coll{};              // array of 10 ints with value 0
...
std::ranges::take_view tv{coll, 5};      // view to first 5 elements of coll
constfunc(tv);                           // OK, modifies the first element of coll
```

That applies to almost all views independent from how they are created:

```
constfunc(coll | std::views::take(5));   // OK, modifies the first element of coll
constfunc(std::ranges::subrange(coll));  // OK, modifies the first element of coll
constfunc(std::views::all(coll));        // OK, modifies the first element of coll
constfunc(std::span(coll));              // OK, modifies the first element of coll
```

Note that this is not a problem, if the view is applied to a `const` container:

```
const std::array<int, 10> coll{};        // array of 10 ints with value 0
...
std::ranges::take_view tv{coll, 5};      // view to first 5 elements of coll
constfunc(tv);                           // compile-time ERROR
```

Therefore, you can avoid the problem by making a range `const` ***before*** the first view is used:

```
std::array<int, 10> coll{};                  // array of 10 ints with value 0
...
std::ranges::take_view tv{std::as_const(coll), 5};          // view to first 5 elements of const coll
constfunc(tv);                                              // compile-time ERROR
constfunc(std::as_const(coll) | std::views::take(5));       // compile-time ERROR
```

The function `std::as_const()` is provided since C++17 ins header `<utility>` Calling `std::as_const()` *after* the view was created has no effect (except that for a few views you are no longer able to iterate over the elements).

There was the idea to disable iteration for `const` views at all. Probably, this would have been the right solution (especially because for some views, iterations are disabled anyway when they are `const`).[6]

5.4.2 Bringing Back Deep Constness to Views

As we now are in a situation, that a `const` in generic code for ranges might not be propagated to the elements, the question is how you can ensure that code does not modify the elements of a range.

The only easy way is to require constness on element access:

- Either by using `const` in the range-based `for` loop:

```
for (const auto& elem : range) {
  ...
}
```

- Or by passing elements converted to `const`:

```
for (auto pos = range.begin(); pos != range.end(); ++pos) {
  elemfunc(std::as_const(*pos));
}
```

To convert the range as a whole to a range with `const` elements, we usually provide `const_iterator` and `cbegin()`/`cend()`.

```
for (decltype(range)::const_iterator pos = range.begin(); pos != range.end(); ++pos) {
  elemfunc(std::as_const(*pos));
}

callTraditionalAlgo(range.cbegin(), range.cend());
```

In fact, we could very simply convert the range as a whole:

```
std::ranges::subrange<decltype(range)::const_iterator> crange{range};
```

However, both `const_iterator` and `cbegin()`/`cend()` members are usually **not provided for views**. Yes, they are not although here we would really really need them.

Some of you might know that there are free-standing helper functions in the C++ standard library: `std::cbegin()` and `std::cend()`. Even worse, **these functions are broken by views**. Their only purpose

[6] Disabling iterations for `const` views was already discussed in 1916, as you can see at http://github.com/ericniebler/range-v3/issues/385.

is that elements are const when you iterate over them. However, they were not defined to deal with containers that have shallow constness (internally they call the const member functions begin() and do ***not*** add constness to the value type). With views these functions are simply broken (they have been broken before but it did not matter for standard types because they all had deep constness and nobody complained):

```
for (auto pos = std::cbegin(range); pos != std::cend(range); ++pos) {
  elemfunc(*pos);   // does not provide constness for the values
}
```

These helper function anyway do not work in all situations well (e.g., ADL might be broken). Even worse, also the new corresponding helper functions by the ranges library are broken the same way:

```
for (auto pos = std::ranges::cbegin(range); pos != std::ranges::cend(range); ++pos) {
  elemfunc(*pos);   // does not provide constness for the values
}
```

Thus:
**BE CAREFUL when using cbegin(), cend(), cdata() in generic code,
because these functions are CURRENTLY BROKEN**.

I do not know what is worse that we got this problem or that the C++ standards committee and its subgroup for the ranges library were not willing to fix this problem for C++20. We could simply provide const_iterator support by views in std::ranges::view_interface<>, which serves as base class for all views. The funny thing is that only one view provides const_iterator support so far: std::string_view. Unfortunately, that is more or less the only view where we do not need it, because in a string view the characters are always const.

However, there is hope: http://wg21.link/p2278 provides an approach to fix this broken constness for C++23 (hopefully also for std::cbegin() and std::cend()).

5.4.3 Generic Code Should Take Ranges with Non-const &&

If you implement generic code for ranges, where you can pass both containers and views, there is a caveat to deal with: Views might change their state while iterating over the elements. And for this reason, they cannot provide begin() and end() as const member function.

In fact, you cannot iterator over the elements of the following standard views if they are declared with const:[7]

- std::ranges::**filter_view**
- std::ranges::**drop_while_view**
- std::ranges::**split_view**
- std::ranges::**drop_view**, if it refers to a range that has no random access or no size() member
- std::ranges::**reverse_view**, if it refers to a range that has different types for the begin and end iterator (sentinel)
- std::ranges::**join_view**, if it refers to a range that generates elements

[7] Thanks to Hannes Hauswedell for pointing out missing parts of this list.

To also support these views in generic code, you should declare the range parameters as *universal references* (also called *forwarding references*). These are references that can refer to all expressions while keeping the fact that the referred object is not `const`.[8]

For these views, there are some ranges for iterating over the elements might change the state. In that case, the generic `maxValue()` function introduced before could not be used:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(const Range& rg)
{
  ...   // ERROR when iterating over the elements of a filter view
}

// max value of a filtered range:
auto odd = [] (auto val) {
             return val % 2 != 0;
           };
std::cout << maxValue(arr | std::views::filter(odd)) << '\n';  // ERROR
```

For this reason, the generic `maxValue()` function better is implemented as follows:

lib/maxvalue2.hpp

```
#include <ranges>

template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
{
  if (std::ranges::empty(rg)) {
    return std::ranges::range_value_t<Range>{};
  }
  auto pos = std::ranges::begin(rg);
  auto max = *pos;
  while (++pos != std::ranges::end(rg)) {
    if (*pos > max) {
      max = *pos;
    }
  }
  return max;
}
```

Here we declare parameter `rg` as a universal/forwarding reference:

```
template<std::ranges::input_range Range>
```

[8] In the C++ standard universal references are called ***forwarding references***, which is unfortunate, because they do not really forward until `std::forward<>()` is called for them, and perfect forwarding is not always the reason to use these reference (as this example demonstrates).

```
std::ranges::range_value_t<Range> maxValue(Range&& rg)
```

That way we ensure that the passed views are not becoming `const` so that we can also pass a filter view now:

lib/maxvalue2.cpp

```
#include "maxvalue2.hpp"
#include <iostream>
#include <algorithm>

int main()
{
  int arr[] = {0, 8, 15, 42, 7};

  // max value of a filtered range:
  auto odd = [] (auto val) {                    // predicate for odd values
               return val % 2 != 0;
             };
  std::cout << maxValue(arr | std::views::filter(odd)) << '\n';  // OK
}
```

The output of the program is:

```
15
```

Open

5.5 Afternotes

The ranges library has a long history.

******* This chapter/section is at work *******

The ranges library was primarily adopted by merging the *Ranges TS* into the C++ standard as proposed by Eric Niebler, Casey Carter, and Christopher Di Bella in http://wg21.link/p0896r4.

This page is intentionally left blank

Chapter 6

Components for Ranges and View

After introducing ranges and views, this chapter gives a first overview of all types and utilities a programmer should know when using ranges.

The general chapter about concepts presents all range concepts.

The chapter about view types presents all the details of the standard view types.

6.1 Major Range Adaptors

As introduced with views and the way they are used, C++ provides several ***range adaptors*** to easily create views with the best performance.

Several of these adaptors apply to specific view types. However, some of them might create different things depending on the characteristics of the passed ranges. For example, adaptors might yield the passed ranges if they already have the characteristics of the result.

There are a few major adaptors that make it easy to create views or convert ranges to views with specific characteristics (independent from the content). These adaptors are:

- `std::views::all()`, which is the primary adaptor to yield a view for a passed range
- `std::views::counted()`, is the primary adaptor to convert a begin and a count into a view
- `std::views::common()`, which yields a view with harmonized types for the begin iterator and the sentinel (end iterator) to be able to pass the range/view to traditional range parameters

These adaptors are described in this section.

All these major adaptors always yield a borrowed range. That is, the lifetime of iterators from these adaptors only depends on the lifetime of the argument passed to them. In other words: Using these adaptors does not introduce additional lifetime issues.

6.1.1 Range Adaptor `all()`

The range adaptor `std::views::all()` is *the* adaptor to convert any range that is not a view yet into a view. That way you guarantee that you have a cheap handle to deal with the elements of a range.

`all(rg)` yields[1]

- A copy of *rg* if it is already a view
- Otherwise, a `std::ranges::ref_view` of *rg*

For example:

```
std::vector<int> vec;
std::ranges::iota_view iv{1, 10};
...
auto v1 = std::views::all(vec);  // std::ranges::ref_view<decltype(vec)>
auto v2 = std::views::all(iv);   // decltype(iv)
auto v3 = std::views::all(std::views::all(vec));
                                 // std::ranges::ref_view<decltype(vec)>
```

The arguments you can pass to `all()` must either already be a view or be a range passed as lvalue:

```
std::vector vec{1, 2, 3};           // a container
std::ranges::iota_view aView{1};    // a view

std::views::all(vec);               // OK
std::views::all(getVector());       // ERROR
std::views::all(std::move(vec));    // ERROR
std::views::all(aView);             // OK
std::views::all(getIotaView());     // OK
```

C++20 also defines type `std::views::all_t<>` as type `all()` yields. This type reflects the fact that `all()` does only work for views or lvalue ranges. It does not work for rvalue ranges that are not views yet:

```
std::views::all_t<std::vector<int>>;              // ERROR
std::views::all_t<std::vector<int>&>;             // OK: ref_view<vector<int>>>
std::views::all_t<std::vector<int>&&>;            // ERROR
std::views::all_t<std::ranges::iota_view<int>>;   // OK: iota_view<int>>
```

Therefore, if you need the type of what `all()` yields in generic code for an expression or name *arg*, instead of writing:

```
decltype(std::views::all(arg))
```

you can write:

```
std::views::all_t<decltype(arg)&>;
```

The `&` is important here (unless you already know that *arg* is a view). If you pass to `decltype` the name of a range object that is not a view, you can also convert it into an expression using additional parentheses. However, using just the name will not compile:

```
std::views::all_t<decltype((name))>;  // OK
std::views::all_t<decltype(name)>;    // ERROR if name is the name of an object
```

[1] C++20 originally stated that `all()` might yield a subrange, but that is an option that can rarely happen in practice and was removed later with `http://wg21.link/p2415`.

This is something to consider, when using `all_t<>` to declare views.

The concept `viewable_range` can be used to check whether a type can be used for `all_t<>` (and therefore corresponding objects be passed to `all()`):[2]

```
std::ranges::viewable_range<std::vector<int>>              // false
std::ranges::viewable_range<std::vector<int>&>             // true
std::ranges::viewable_range<std::vector<int>&&>            // false
std::ranges::viewable_range<std::ranges::iota_view<int>>   // true
```

6.1.2 Range Adaptor `counted()`

The range adaptor `std::views::counted()` provides the most flexible way to create a view from a begin iterator and a count. Depending on what you pass, it creates a cheap view to the first *n* elements of a begin iterator/pointer.

It is up to the programmer to ensure that the count is valid (passing a count that is too high results in undefined behavior). The count may be 0, which means that the range is empty.

`counted(beg, sz)` yields different types depending on the characteristics of the range it is called for:

- It yields a `std::span` if the passed begin iterator is a `contiguous_iterator` (refers to elements that are stored in contiguous memory). This applies to raw pointers and iterators of `std::vector<>`, `std::array<>`, and raw arrays.
- Otherwise, it yields a `std::ranges::subrange` if the passed begin iterator is a `random_access_iterator` (supports to jump back and forth between elements). This applies to iterators of `std::deque<>`.
- Otherwise, it yields a `std::ranges::subrange` of a `std::counted_iterator` with a dummy sentinel of type `std::default_sentinel_t`. This means that the iterator in the subrange counts while iterating. This applies to iterators of lists, associative containers, and unordered containers (hash tables).

For example:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vec5 = std::ranges::find(vec, 5);
auto v1 = std::views::counted(vec5, 3);  // view to element 5 and 2 more elements in vec
  // v1 is std::span<int>

std::deque<int> deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto deq5 = std::ranges::find(deq, 5);
auto v2 = std::views::counted(deq5, 3);  // view to element 5 and 2 more elements in deq
  // v2 is std::ranges::subrange<std::deque<int>::iterator

std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto lst5 = std::ranges::find(lst, 5);
auto v2 = std::views::counted(lst5, 3);  // view to element 5 and 2 more elements in lst
  // v3 is std::ranges::subrange<std::counted_iterator<std::list<int>::iterator>,
  //                             std::default_sentinel_t>
```

[2] For lvalues of move-only view types there is an issue that `all()` is ill-formed although `viewable_range` is satisfied.

Note that this code is at risk because it creates undefined behavior if there is no 5 in the collection with two elements behind. So, you should check this if you are not sure that this is the case.

Note also that if you want to have the first 3 elements of a range, using the take view is a safer way to get the result:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto v1 = std::views::take(vec, 3);  // view to first 3 elements in vec
```

The take view ***does*** check whether there are enough elements and yields less elements if not.

6.1.3 Range Adaptor `common()`

The range adaptor `std::views::common()` yields a view with harmonized types for the begin iterator and the sentinel (end iterator) for the passed range. It acts like the range adaptor `std::views::all()` with the additional behavior to create a `std::ranges::common_view` from the passed argument, if its iterators have different types.

For example, assume we want to call a traditional algorithm `algo()` that requires the same type for both the begin and the end iterator. Then we can call it for iterators of may-be different types as follows:

```
template<typename TBeg, typename TEnd>
void callAlgo(TBeg beg, TEnd end)
{
  auto v = std::views::common(std::ranges::subrange(beg,end));
  algo(v.begin(), v.end());  // assume algo() requires iterators with the same type
}
```

`common(`*rg*`)` yields

- A copy of *rg* if it is already a view with the same begin and end iterator types
- Otherwise, a `std::ranges::ref_view` of *rg* if it is a range object with the same begin and end iterator types
- Otherwise, a `std::ranges::common_view` of *rg*

For example:

```
std::list<int> lst;
std::ranges::iota_view iv{1, 10};
...
auto v1 = std::views::common(lst);   // std::ranges::ref_view<decltype(lst)>
auto v2 = std::views::common(iv);    // decltype(iv)
auto v3 = std::views::common(std::views::all(vec));
                                     // std::ranges::ref_view<decltype(lst)>

std::list<int> lst {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vt = std::views::take(lst, 5);  // begin() and end() have different types
auto v4 = std::views::common(vt);    // std::ranges::common_view<decltype(vt)>
```

Note that both the constructors of a `std::ranges::common_view` and the helper type `std::common:iterator` require that passed iterators have different types. So, you should use this adaptor if you do not know whether iterator types differ.

6.2 New Iterators

For the (better) support of ranges and views, C++20 introduces a couple of new iterator types:

- `std::counted_iterator` for an iterator that itself has a *count* to specify the end of a range
- `std::common_iterator` for a common iterator type that can be used for two iterators having different types
- `std::default_sentinel_t` for an end iterator that forces an iterator to check for its end
- `std::unreachable_sentinel_t` for an end iterator that can never be reached so that the range is endless

6.2.1 `std::counted_iterator`

A *counted iterator* is an iterator that has a counter for the maximum number of elements to iterate over.

There are two ways to use such an iterator:

- Iterating while checking how many elements are left:

```
for (std::counted_iterator pos{coll.begin(), 5}; pos.count() > 0; ++pos) {
  std::cout << *pos << '\n';
}
std::cout << '\n';
```

- Iterating while comparing with a default sentinel:

```
for (std::counted_iterator pos{coll.begin(), 5};
     pos != std::default_sentinel; ++pos) {
  std::cout << *pos << '\n';
}
```

This feature is used when the view adaptor `std::ranges::counted()` yields a subrange for iterators that are not random-access iterators.

Table *Operations of Class* `std::counted_iterator<>` lists the API of a counted iterator.

Operation	Effect
countedItor *pos*`{}`	Creates a counted iterator that refers to no element (*count* is `0`)
countedItor *pos*`{`*pos2*`,` *num*`}`	Creates a counted iterator of *num* elements starting with *pos2*
countedItor *pos*`{`*pos2*`}`	Creates a counted iterator as a (type converted) copy of a counted iterator *pos2*
pos`.count()`	Yields how many elements are left (`0` means we are at the end)
pos`.base()`	Yields (a copy of) the underlying iterator
...	All standard iterator operations of the underlying iterator type
pos `==` `std::default_sentinel`	Yields whether the iterator is at the end
pos `!=` `std::default_sentinel`	Yields whether the iterator is not at the end

Table 6.1. Operations of Class `std::counted_iterator<>`

It is up to the programmer to ensure that

- The initial *count* is not higher than the iterator initially passed
- The counted iterator does not increment more than *count* times
- The counted iterator does not access elements beyond the *count*-th element (as usual, the position behind the last element is valid).

6.2.2 `std::common_iterator`

Type `std::common_iterator<>` is used to harmonize the type of two iterators. It wraps the two iterators so that from the outside both iterators have the same type. Internally a value of one of the two types is stored (for this the iterator typically uses a `std::variant<>`).

For example, if a traditional algorithm taking a begin and an end iterator does not support that these iterators have different types, you can use this type to enable the call:

```
algo(beg, end);      // if this is an error due to different types

algo(std::common_iterator<decltype(beg), decltype(end)>{beg},    // OK
     std::common_iterator<decltype(beg), decltype(end)>{end});
```

Note that it is a compile-time error if the types passed to `common_iterator<>` are the same. Thus, in generic code you might have to call:

```
template<typename TBeg, typename TEnd>
void callAlgo(TBeg beg, TEnd end)
{
  if constexpr(std::same_as<TBeg, TEnd>) {
    algo(beg, end);
  }
  else {
    algo(std::common_iterator<decltype(beg), decltype(end)>{beg},
         std::common_iterator<decltype(beg), decltype(end)>{end});
  }
}
```

A more convenient way to have the same effect is to use the common() adaptor:

```
template<typename TBeg, typename TEnd>
void callAlgo(TBeg beg, TEnd end)
{
  auto v = std::views::common(std::ranges::subrange(beg,end));
  algo(v.begin(), v.end());
}
```

******* This chapter/section is at work *******

6.2.3 `std::default_sentinel`

A *default sentinel* is an iterator that provides no operations at all. C++20 provides a corresponding object `std::default_sentinel`, which has type `std::default_sentinel_t` in `<iterator>`. The type has no members:

```
namespace std {
  class default_sentinel_t {
  };
  inline constexpr default_sentinel_t default_sentinel{};
}
```

The type and the value are provided to serve as a dummy sentinel (end iterator), which is used when an iterator knows when a range ends. For those iterators, a comparison with `std::default_sentinel` triggers the internal check of the iterator whether it is at the end or how far it is from it.

For example, counted iterators define an operator `==` for themselves with a default sentinel to perform checking whether they are at the end:

```
namespace std {
  template<std::input_or_output_iterator I>
  class counted_iterator {
    ...
    friend constexpr bool operator==(const counted_iterator& p,
                                     std::default_sentinel_t) {
      ...   // returns whether p is at the end
    }
  };
}
```

This allows code like this:

```
// iterate over the first 5 elements:
for (std::counted_iterator pos{coll.begin(), 5};
     pos != std::default_sentinel;
     ++pos) {
  std::cout << *pos << '\n';
}
```

The standard defines the following operations with default sentinels:

- For `std::counted_iterator`:
  - Comparisons with operator `==` and `!=`
  - Computing the distance with operator `-`
- `std::views::counted()` may create a subrange of a counted iterator and a default sentinel
- For `std::istream_iterator`:
  - Default sentinels can be used as initial value, which has the same effect as the default constructor
  - Comparisons with operator `==` and `!=`
- For `std::istreambuf_iterator`:
  - Default sentinels can be used as initial value, which has the same effect as the default constructor

  - Comparisons with operator == and !=
- For std::ranges::basic_istream_view<>:
  - Istream views yield std::default_sentinel as end()
  - Istream view iterators can compare with operator == and !=
- For std::ranges::take_view<>:
  - Take views may yield std::default_sentinel as end()
- For std::ranges::split_view<>:
  - Split views may yield std::default_sentinel as end()
  - Split view iterators can compare with operator == and !=

6.2.4 std::unreachable_sentinel

The value std::unreachable_sentinel of type std::unreachable_sentinel_t was introduced in C++20 to specify a sentinel (end iterator of a range) that is not reachable. It effectively says "*Do not compare with me at all*." The effect is that it can be used to specify unlimited ranges.

When it is used it can optimize generated code, because a compiler can detect that comparing it with another iterator never yields true, so that it might skip any check against the end.

For example, ***if we know*** that a value 42 exists in a collection, we can search for it as follows:

```
auto pos42  = std::ranges::find(coll.begin(), std::unreachable_sentinel,
                                42);
```

Normally, the algorithm would compare against both, 42 and coll.end() (or whatever is passed as end/sentinel). Because unreachable_sentinel is used, for which any comparison with an iterator always yields false, compilers can optimize the code to compare only against 42. Of course, programmers have to ensure that a 42 exists.

The iota view provides an example of using unreachable_sentinel for endless ranges.

6.3 Helper Functions in std::ranges

Table *Generic utility functions for ranges* lists all utility functions for ranges.

Table *Generic utility functions for ranges* lists all generic type definitions for iterators of ranges.

Almost all of these utility functions were also available before C++20 directly in namespace std. The only exceptions are

- ssize(), which in C++20 was also introduced as std::ssize()
- cdata(), which in C++20 does not exist in namespace std at all

So, why did we not fix the functions in namespace std and which functions should be used? The answer to the second part of the question is simple: **prefer the functions/utilities in namespace std::ranges over those in namespace std**.

The reason is not only that the functions/utilities in namespace std::ranges use concepts, which helps to find problems and bugs at compile time. The reason is that the functions in namespace std sometimes have flaws, which the new implementation in std::ranges fixes:

- One problem is *argument dependent lookup*.

Function	Meaning
`rng::`**`empty(`***rg***`)`**	Yields whether the range is empty
`rng::`**`size(`***rg***`)`**	Yields the size of the range
`rng::`**`ssize(`***rg***`)`**	Yields the size of the range as value of a signed type
`rng::`**`begin(`***rg***`)`**	Yields an iterator to the first element of the range
`rng::`**`end(`***rg***`)`**	Yields a sentinel (an iterator to the end) of the range
`rng::`**`cbegin(`***rg***`)`**	Yields a constant iterator to the first element of the range
`rng::`**`cend(`***rg***`)`**	Yields a constant sentinel (a constant iterator to the end) of the range
`rng::`**`rbegin(`***rg***`)`**	Yields a reverse iterator to the first element of the range
`rng::`**`rend(`***rg***`)`**	Yields a reverse sentinel (an iterator to the end) of the range
`rng::`**`crbegin(`***rg***`)`**	Yields a reverse constant iterator to the first element of the range
`rng::`**`crend(`***rg***`)`**	Yields a reverse constant sentinel (a constant iterator to the end) of the range
`rng::`**`data(`***rg***`)`**	Yields the raw data of the range
`rng::`**`cdata(`***rg***`)`**	Yields the raw data of the range with `const` elements

Table 6.2. Generic utility functions for ranges

Function	Meaning
`rng::`**`distance(`***from***`, `***to***`)`**	Yields the distance (number of elements) between *from* and *to*
`rng::`**`distance(`***rg***`)`**	Yields number of elements in *rg* (size even for ranges that have no `size()`)
`rng::`**`next(`***pos***`)`**	Yields the position of the next element behind *pos*
`rng::`**`next(`***pos***`, `***n***`)`**	Yields the position of the *n*-th next element behind *pos*
`rng::`**`next(`***pos***`, `***to***`)`**	Yields the position *to* behind *pos*
`rng::`**`next(`***pos***`, `***n***`, `***maxpos***`)`**	Yields the position of the *n*-th element after *pos* but not behind *maxpos*
`rng::`**`prev(`***pos***`)`**	Yields the position of the element before *pos*
`rng::`**`prev(`***pos***`, `***n***`)`**	Yields the position of the *n*-th element before *pos*
`rng::`**`prev(`***pos***`, `***n***`, `***minpos***`)`**	Yields the position of the *n*-th element before *pos* but not before *minpos*
`rng::`**`advance(`***pos***`, `***n***`)`**	Advances *pos* forward/backward *n* elements
`rng::`**`advance(`***pos***`, `***to***`)`**	Advances *pos* forward to *to*
`rng::`**`advance(`***pos***`, `***n***`, `***maxpos***`)`**	Advances *pos* forward/backward *n* element but not further than *maxpos*

Table 6.3. Iterator utility functions for ranges

- One problem is `const` correctness.

For argument dependent lookup the problem is that utilities like `std::begin()` do not work if you explicitly qualify the call with namespace `std`.[3] Assume you want to write code that always finds the right `begin()` function for an object `obj` of an arbitrary type.

- To also support types that have no `begin()` member you cannot always call the member function

  ```
  obj.begin();        // only works if a member function begin() is provided
  ```

- However, calling a free-standing `begin()` has to find `std::begin()`, which only works without qualification if `obj` has a type known in `std`:

  ```
  begin(obj);         // does not find std::begin() if obj is not in std
  ```

- So we might explicitly qualify to use `std.begin()`, but then the implementation of `std::begin()` does not find an overloaded free-standing `begin()`:

  ```
  class MyType {
    ...
  };
  int* begin(MyType);
  ...
  MyType obj;
  std::begin(obj);   // std::begin() does not find ::begin(MyType)
  ```

- The known workaround to use `std::begin()` in generic code is to put an additional `using` declaration in front of the call and ***not*** qualify the call:

  ```
  using std::begin;
  begin(obj);         // OK, works in all these cases
  ```

The new `std::ranges::begin()` does not have this problem

```
std::ranges::begin(obj);     // OK, works in all these cases
```

The trick is that `begin` is not a function that uses `ADL`, but a function object that implements all possible lookups. See *`lib/begin.cpp`* for a complete example.

And that applies to all utilities that are defined in `std::ranges`. Therefore, code using `std::ranges` utilities in general supports more types and more complex use cases.

6.4 Helper Types in `std::ranges`

Table *Generic type definitions for ranges* lists all generic type definitions for ranges. They are defined as alias templates.

******* This chapter/section is at work *******

`std::ranges::`**`iterator_t`**`<`*Rg*`>`

- yields the type `std::ranges::begin(`*r*`)` returns for an object *r* of type *Rg*.

[3] Thanks to Tim Song and Barry Revzin for pointing this out.

Type Function	Meaning
`rng::`**`iterator_t`**`<`*Rg*`>`	Type of an iterator iterating over *Rg* (what `begin()` yields)
`rng::`**`sentinel_t`**`<`*Rg*`>`	Type of an end iterator for *Rg* (what `end()` yields)
`rng::`**`range_difference_t`**`<`*Rg*`>`	Type of the difference of two iterators
`rng::`**`range_size_t`**`<`*Rg*`>`	Type of what the `size()` function returns
`rng::`**`range_value_t`**`<`*Rg*`>`	Type of the value the iterator refers to
`rng::`**`range_reference_t`**`<`*Rg*`>`	Type of a reference to the element type
`rng::`**`range_rvalue_reference_t`**`<`*Rg*`>`	Type of an rvalue reference to the element type
`rng::`**`borrowed_iterator_t`**`<`*Rg*`>`	`rng::iterator_t<`*Rg*`>` for a borrowed range, otherwise `std::ranges::dangling`
`rng::`**`borrowed_subrange_t`**`<`*Rg*`>`	The subrange type of the iterator type for a borrowed range, otherwise `std::ranges::dangling`

Table 6.4. Generic type definitions for ranges

`std::ranges::`**`sentinel_t`**`<`*Rg*`>`

- yields the type `std::ranges::end(`*r*`)` returns for an object *r* of type *Rg*.

6.5 Open

******* This chapter/section is at work *******

6.6 Afternotes

This page is intentionally left blank

Chapter 7

View Types in Detail

This chapter discusses the details of each and every view type the C++20 standard library introduced.

We start with a first overview and introducing the base classes C++20 provides for views. Then, we discuss all view types of C++20 to create views (views that refer to external elements or views that generate values themselves). Finally, we document all modifying views types that especially may be used for components in pipelines of views: views that filter out elements, views that modify elements, views that mutate (change the order of) elements, and views that deal with multiple ranges.

7.1 Overview of all Views

C++20 provides a significant number of different views. Most, but not all of them can be created with a range adaptor, which is usually simpler to use and creates a view that has the best performance.

7.1.1 Overview of Features That Create Views

Table *Features to Create Views and their Adaptors* lists the features to create views and the corresponding adaptors. Some views refer to external elements (and therefore wrap ranges), while others create a sequence of values themselves.

In general, you should prefer using the adaptors. For example, you should always prefer to use the adaptor `std::views::all()` over the type `std::ranges::ref_view<>`. However, for some view types such as `basic_istream_view` that is not possible.

Note the different namespaces used:

- `span` and `string_view` are in namespace `std`.
- All adaptors are in namespace `std::views`.
- All other view types are in namespace `std::ranges`.

Type	Adaptor	Effect
rng::subrange		Creates a view defined as begin iterator and sentinel (end iterator)
rng::ref_view		Creates a view from a range (a new view that refers to all elements of a range)
	all(...)	Converts a range into a view (uses a view as it is or creates a ref_view)
std::span		Creates a view from a range with sequentially stored elements (arrays and vectors)
	counted(...)	Converts a begin and a count into a view (converts to a span or to a subrange)
std::basic_string_view		Creates a read-only view to a character sequence (since C++17)
rng::basic_istream_view<>		Creates a view reading elements from a stream
rng::iota_view	iota(...)	Creates a view by repeatedly incrementing an initial value
rng::single_view	single(...)	Creates a view with exactly one element
rng::empty_view	empty<...>	Creates a view with no elements
rng::common_view	common(...)	Converts to a view where begin and end have the same type (to pass it to legacy code)

Table 7.1. Features to Create Views and their Adaptors

7.1.2 Overview of Modifying Views

Table *Modifying Views for Pipelines* lists the standard views that modify elements of a given range in various way (filter out elements, modify the values of elements, change the order of elements, and combine or create sub-ranges). They can especially be used in pipelines of views.

7.2 Base Classes for Views

All standard views are derived from class std::ranges::**view_interface<***viewType***>**, which is derived from class std::ranges::**view_base**:

- The root class std::ranges::view_base is empty and only ensures that a derived class models the concept std::ranges::view (for this, it initializes std::ranges::enable_view<> for the type of the view with true).
- The class template std::ranges::**view_interface<>** introduces several basic member functions based on the definition of begin() and end() of a derived view type, which is passed to this based class as template parameter.

Table *Operations of* std::ranges::view_interface<> lists its API.

Whenever you define your own view type, you should derive it from view_interface<> with your own type passed as argument. For example:

```
template<typename T>
```

Type	Adaptor	Effect
take_view	take(*num*)	Takes the first *num* elements of a range
take_while_view	take_while(*pred*)	Takes all leading elements that match a predicate
drop_view	drop(*num*)	Drops the first *num* elements of a range
drop_while_view	drop_while(*pred*)	Drops all leading elements that match a predicate
filter_view	filter(*pred*)	Filters to use all elements that match a predicate
transform_view	transform(*pred*)	Transforms all elements of a range
elements_view	elements<*idx*>	Use the *idx*th member of each tuple-like element
keys_view	keys	Use the first member of each tuple-like element
values_view	values	Use the second member of each tuple-like element
reverse_view	reverse	Reverse the order of elements
join_view	join	Joins the elements of multiple ranges
split_view	split(*pred*)	Split a range into sub-ranges
lazy_split_view	lazy_split(*pred*)	Split an input/const range into sub-ranges

Table 7.2. Modifying Views for Pipelines

Operation	Effect	Provided if
r.empty()	Yields whether *r* is empty (begin() == end())	At least forward iterators
if (*r*)	true if *r* is not empty	At least forward iterators
r.size()	Yields the number of elements	Can compute the difference between begin and end
r.front()	Yields the first element	At least forward iterators
r.back()	Yields the last element	At least bidirectional iterators and end() yields the same type as begin()
r[*idx*]	Yields the *n*-th element	At least random-access iterators
r.data()	Yields a raw pointer to the memory of the elements	Elements are in contiguous memory

Table 7.3. Operations of std::ranges::view_interface<>

```
class MyView : public std::ranges::view_interface<MyView<T>> {
 public:
  ... begin() ...;
  ... end() ...;
  ...
};
```

Based on the return types of begin() and end() your type then automatically provides the member functions listed in table *Operations of* std::ranges::view_interface<> if the preconditions for their availability fits.

7.3 Creating Views to External Elements

This section discusses all features of C++20 that create views that refer to external elements (passed as a range or as begin and end or begin and a count).

7.3.1 `std::ranges::subrange`

Class template `std::ranges::subrange<>` defines a view to the elements of a range that is either passed as single argument or defined as pair of begin iterators and iterator or sentinel. The view itself internally represents the elements by storing begin and end/sentinel.

The major use case of the subrange view is to convert a pair of begin iterator and sentinel (end iterator) into *one* object. For example:

```
void foo(auto beg, auto end)
{
  std::ranges::subrange rg{beg, end};
  ...
}
```

Two range adaptors can also create a subrange:

- `std::views::all(`*`rg`*`)`, which creates a subrange if *`rg`* is not already a view. That can be used to be able to pass containers as lightweight objects.
- `std::views::counted(`*`beg`*`, `*`sz`*`)`, which creates a subrange of *`sz`* elements starting with the iterator *`beg`* if *`beg`* does not refer to consecutive memory (for consecutive memory, `counted()` creates a span).

Table *Operations of Class* `std::ranges::subrange<>` lists the API of a subrange.

The constructors taking a *`szHint`* allows programmers to convert an unsized range into a a sized subrange. If you pass *`szHint`* and the range does not have a known size (e.g., it is a forward list), the hint is used as size. It is undefined behavior if you specify a wrong size.

Iterators do not refer to this view. Instead they refer to the underlying range. Therefore, a subrange is a borrowed range. However, note that iterators can still dangle, when the underlying range is no longer there.

Tuple-Like Interface of Subranges

`std::ranges::subrange` also has a tuple-like interface to support structured bindings (introduced to C++ with C++17). Thus, you can easily initialize a begin iterator and a sentinel (end iterator) doing something like the following:

```
auto [beg, end] = std::ranges::subrange(coll);
```

For this, class `std::ranges::subrange` provides

- a specialization of `std::tuple_size<>`
- specializations of `std::tuple_element<>`
- `get<>()` functions for the indexes `0` and `1`

Operation	Effect
subrange *r*{}	Creates an empty subrange
subrange *r*{*rg*}	Creates a subrange with the elements of range *rg*
subrange *r*{*rg*, *szHint*}	Creates a subrange with the elements of range *rg* specifying that the range has *szHint* elements
subrange *r*{*beg*, *end*}	Creates a subrange with the elements of range [*beg*, *end*)
subrange *r*{*beg*, *end*, *szHint*}	Creates a subrange with the elements of range [*beg*, *end*) specifying that the range has *szHint* elements
r.begin()	Yields the begin iterator
r.end()	Yields the sentinel (end iterator)
r.empty()	Yields whether *r* is empty
if (*r*)	true if *r* is not empty
r.size()	Yields the number of elements (available if sized)
r.front()	Yields the first element (available if forwarding)
r.back()	Yields the last element (available if bidirectional)
r[*idx*]	Yields the *n*-th element (available if random-access)
r.data()	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r.next()	Yields a subrange starting with the second element
r.next(*n*)	Yields a subrange starting with the *n*-th element
r.prev()	Yields a subrange starting with the element before the first element
r.prev(*n*)	Yields a subrange starting with the *n*-th element before the first element
r.advance(*dist*)	Modifies *r* so that it starts *num* elements later (starts earlier with negative *num*)
auto [*beg*, *end*] = *r*	Initialize *beg* and *end* with begin and end/sentinel of *r*

Table 7.4. Operations of Class `std::ranges::subrange<>`

Using Subranges to Harmonize Array Types

The type of subranges only depends on the type of the iterators (and whether or not `size()` is provided). This can be used to harmonize the type of arrays without losing any information.

Consider the following example:

```
int arr1[5] = {...};
int arr2[10] = {...};
std::array<int, 15> arr3 = {...};
std::array<int, 20> arr4 = {...};
```

Passing `arr1` til `arr4` to a subrange yields the same type. This can be useful to avoid multiple instantiations and code generations of generic code for raw arrays and `std::array`'s of same the element type but different size. If function templates are called and the calls are not optimized away, the function template is instantiated only once, when subranges are passed:

```
template<std::ranges::input_range T>
void foo(const T& coll)
{
  ...
}

// 4 instantiation for 4 different types:
foo(arr1);
foo(arr2);
foo(arr3);
foo(arr4);

// 1 instantiation (all have the same type):
foo(std::ranges::subrange{arr1});
foo(std::ranges::subrange{arr2});
foo(std::ranges::subrange{arr3});
foo(std::ranges::subrange{arr4});
```

You can get the same effect when only using `begin()` and `end()` of these types.

7.3.2 `std::ranges::ref_view`

Class template `std::ranges::ref_view<>` defines a view that simply refers to a range. That way passing the view by value has an effect like passing the range by reference.[1]

The major use case of the `ref_view` is to convert a container into a lightweight object that is cheap to copy.

A range adaptors can also create a `ref_view`:

- `std::views::all(rg)`, which creates a `ref_view` if *`rg`* is not already a view.

Table *Operations of Class* `std::ranges::ref_view<>` lists the API of a `ref_view`.

Note that you can only create a view to an lvalue (a range that has a name):

```
std::vector coll{0, 8, 15};
...
std::ranges::ref_view v1{coll};                    // OK, refers to coll
std::ranges::ref_view v2{std::move(coll)};         // ERROR
std::ranges::ref_view v3{std::vector{0, 8, 15}};   // ERROR
```

Iterators do not refer to this view. Instead they refer to the underlying range. Therefore, the ref view is a borrowed range. However, note that iterators can still dangle, when the underlying range is no longer there.

[1] The effect is similar to type `std::reference_wrapper<>` which makes references to first-class objects created with `std::ref()` and `std::cref()`.

Operation	Effect
refview `r{}`	Creates a `ref_view` that refers to no range
refview `r{rg}`	Creates a `ref_view` that refers to range *rg*
`r.begin()`	Yields the begin iterator
`r.end()`	Yields the sentinel (end iterator)
`r.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (r)`	`true` if *r* is not empty (available if `empty()` is defined)
`r.size()`	Yields the number of elements (available if it refers to a sized range)
`r.front()`	Yields the first element (available if forwarding)
`r.back()`	Yields the last element (available if bidirectional)
`r[idx]`	Yields the *n*-th element (available if random-access)
`r.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
`r.base()`	Yields a reference to the range *r* refers to

Table 7.5. Operations of Class `std::ranges::ref_view<>`

7.3.3 `std::ranges::common_view`

Class template `std::ranges::`**`common_view<>`** is a view that harmonizes the begin and end iterator type of a range to be able to pass them to code where the same type is required (such as to constructors of containers or traditional algorithms).

For the class template there is also a range adaptor `std::views::common()` provided, which takes a range as argument and converts it to a `common_view` if `begin()` and `end()` do not yield the same type. If the argument already is a `std::ranges::common_range`, `std::views::common()` just yields the passed argument.

The major use case of the common view is to pass ranges or views that have different types for the begin end end to generic code that expects them to have the same type. A typical example is to pass begin and end of a range to a constructor of a container or to a traditional algorithm. For example:

```
std::list<int> lst {1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v1 = std::views::take(lst, 5);       // begin() and end() have different types
std::vector coll{v1.begin(), v1.end()};   // ERROR: containers require the same type

auto v2 = std::views::common(std::views::take(lst, 5));  // same types now
std::vector<int> coll{v2.begin(), v2.end()};             // OK
```

Table *Operations of Class* `std::ranges::common_view<>` lists the API of a common view.

Note that the constructor of `common_view` requires that the iterators of an initial range have different types. If you use the `common()` adaptor the types might be the same:

```
auto v1 = std::ranges::common_view{std::ranges::subrange(beg,beg)};  // ERROR
auto v2 = std::views::common(std::ranges::subrange(beg,beg));        // OK
```

Internally, `common_view` uses `std::common_iterator`.

Operation	Effect
commonview `r{}`	Creates a `common_view` that refers to no range
commonview `r{rg}`	Creates a `common_view` that refers to range `rg`
`r.begin()`	Yields the begin iterator
`r.end()`	Yields the sentinel (end iterator)
`r.empty()`	Yields whether `r` is empty (available if the range supports it)
`if (r)`	`true` if `r` is not empty (available if `empty()` is defined)
`r.size()`	Yields the number of elements (available if it refers to a sized range)
`r.front()`	Yields the first element (available if forwarding)
`r.back()`	Yields the last element (available if bidirectional)
`r[idx]`	Yields the *n*-th element (available if random-access)
`r.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
`r.base()`	Yields a reference to the range `r` refers to

Table 7.6. Operations of Class `std::ranges::common_view<>`

7.4 Generating Views

This section discusses all features of C++20 to create views that generate values themselves (so that they do not refer to elements or values outside the view).

7.4.1 `std::ranges::iota_view`

Class template `std::ranges::iota_view<>` is a view that generates a sequence of values. These values may be integral such as

- 1, 2, 3 ...
- 'a', 'b', 'c' ...

or they might use operator `++` to generate a sequence of pointers or iterators. The sequence might be limited or unlimited (endless).

The major use case of the iota view is to iterate over a sequence of values. For example:

```
std::ranges::iota_view v1{1, 100};          // range with values: 1, 2, ... 99
for (auto val : v1) {
  std::cout << val << '\n';                 // print these values
}
```

For the class template there is also a range adaptor `std::views::iota()` provided, which takes one or two arguments, so that you can also write:

```
for (auto val : std::views::iota(1, 100)) {  // iterate over values 1, 2, ... 99
  std::cout << val << '\n';                  // print these values
}
```

Because the iterators hold the current value, the iota view is a borrowed range (the lifetime of its iterators does not depend on the view). If the iota view is limited and the end value has the same type as the begin value, the view is a common range.

Table *Operations of Class* `std::ranges::iota_view<>` lists the API of an iota view.

When declaring an iota view as follows:

```
std::ranges::iota_view v1{1, 100};      // range with values: 1, 2, ... 99
```

the type of `v1` is deduced as `std::ranges::iota_view<int, int>`, which creates an object that provides the basic API with `begin()` and `end()` yielding an iterator to these values. The iterators internally store the current value and increment it when the iterator is incremented:

```
std::ranges::iota_view v1{1, 100};      // range with values: 1, 2, ... 99
auto pos = v1.begin();                  // initialize iterator with value 1
std::cout << *pos;                      // print current value (here value 1)
++pos;                                  // increment to next value (i.e., increment value to 2)
std::cout << *pos;                      // print current value (here value 2)
```

Thus, the type of the values has to support operator `++` For this reason, a value type such as `bool` or `std::string` is not possible.

Note that the type of this iterator is up to the implementation of `iota_view` so that you have to use `auto` when using it here. Alternatively, you could also declare `pos` with `std::ranges::iterator_t<v1>`.

Operation	Effect
iota_view<*type*> *v*	Creates an unlimited sequence starting with the default value of *type*
iota_view v{*begVal*}	Creates an unlimited sequence starting with *begVal*
iota_view v{*begVal*, *endVal*}	Creates a sequence starting with *begVal* up to the value before *endVal*
iota_view v{*beg*, *end*}	Helper constructor to enable sub views
v.begin()	Yields the begin iterator (referring to the starting value)
v.end()	Yields the sentinel (end iterator), which is std::unreachable_sentinel if unlimited
v.empty()	Yields whether *v* is empty
if (*v*)	true if *v* is not empty
v.size()	Yields the number of elements (available if limited and computable)
v.front()	Yields the first element
v.back()	Yields the last element (available if limited)
v[*idx*]	Yields the *n*-th element

Table 7.7. Operations of Class std::ranges::iota_view<>

As for ranges of iterators, the range of value is an half-open range and the end value is not included. To include the end value, you might have to increment the end :

```
std::ranges::iota_view letters{'a', 'z'+1};  // values: 'a', 'b', ... 'z'
for (auto c : letters) {
  std::cout << c << ' ';                     // print these values/letters
}
```

Note that the output of this loop depends on the character set. Only if the lower-case letters have consecutive values (as it is the case with ASCII or ISO-Latin-1 or UTF-8) the view iterates nothing else but lower-case characters. By using char8_t you can ensure that this is portably the case for UTF-8 characters:

```
std::ranges::iota_view letters{u8'a', u8'z'+1};  // UTF-8 values from a to z
```

If no end value is passed, the view is unlimited and generates an endless sequence of values:

```
std::ranges::iota_view v2{10L};          // unlimited range with values: 10L, 11L, 12L, ...
std::ranges::iota_view<int> v3;          // unlimited range with values: 0, 1, 2, ...
```

The type of v3 is std::ranges::iota_view<int, std::unreachable_sentinel_t, so that end() yields std::unreachable_sentinel.

An unlimited iota view is endless. When the iterator represents the highest value and iterates to the next value it calls operator ++, which usually performs an overflow so that the next value is the lowest value of the value type. If the view has an end value that never matches, it behaves the same.

When initialized with iterators, the iota view iterates over iterators referring to the underlying values. In this case, the availability of the member functions size() and operator[] depends on the support for these operators in the passed range. You can use this to deal with iterators to all elements of a range.

```
std::list<int> coll{2, 4, 6, 8, 10, 12, 14};
```

```
...
// pass iterator to each element to foo():
for (const auto& pos : std::views::iota(coll.begin(), coll.end())) {
  foo(pos);
}
```

You can also use this to initialize a container that refers to all current elements of `coll`:

```
std::ranges::iota_view itors{coll.begin(), coll.end()};
std::vector<std::ranges::iterator_t<decltype(coll)>> refColl{itors.begin(),
                                                             itors.end()};
```

Note that if you skip to specify the element type of `refColl` you have to use parentheses. Otherwise, you would initialize the vector with two iterator elements:

```
std::vector refColl(collItors.begin(), collItors.end());
```

The last constructor of `iota_view<>` is provided to support generic code that can create a sub-view of an iota view, such as the drop view does:[2]

```
// generic function to drop the first element from a view:
auto dropFirst = [] (auto v) {
  return decltype(v){++v.begin(), v.end()};
};

std::ranges::iota_view v1{1, 9};
auto v2 = dropFirst(v1);   // iota view from 2 til 9
```

7.4.2 `std::ranges::single_view`

Class template `std::ranges::`**`single_view<>`** is a view with one element. Unless the value type is `const`, you can even modify the value.

The overall effect is that an single view behaves roughly like cheap vector with one element (not allocating any heap memory). In fact is models the following concepts:

- std::ranges::contiguous_range (which implies std::ranges::random_access_range)
- std::ranges::sized_range

The major use case of the single view is to call generic code with a cheap view that has no elements and for which the type system knows that it never has any elements:

```
std::ranges::single_view<int> v1;              // single view
for (auto val : v1) {
  ...                                          // called once
}
```

This can be used for test cases or for cases where code has to provide a view, which in a specific context never can have any elements.

For the class template also a range adaptor `std::views::single(`*`val`*`)` is provided:

[2] Thanks to Hannes Hauswedell for pointing this out.

```
for (auto val : std::views::single(42)) {     // iterate over the single int value 42
  ...                                         // called once
}
```

Table *Operations of Class* `std::ranges::single_view<>` lists the API of an single view.

Operation	Effect
single_view`<`*type*`>` *v*	Creates a view with one value initialized elements of type *type*
single_view *v*`{`*val*`}`	Creates a view with one element of value *val*
v`.begin()`	Yields a raw pointer to the element
v`.end()`	Yields a raw pointer to the element
v`.empty()`	Yields `false`
`if (`*v*`)`	Always `true`
v`.size()`	Yields `1`
v`.front()`	Yields a reference to the current value
v`.back()`	Yields a reference to the current value
v`[`*idx*`]`	Yields the current value for *idx* `0`
r`.data()`	Yields a raw pointer to the element

Table 7.8. Operations of Class `std::ranges::single_view<>`

Note that for a non-`const` single view you can modify the value of the "element:"

```
std::ranges::single_view v2{42};
std::cout << v2.front() << '\n';          // prints 42
++v2.front();                             // OK, modifies value of the view
std::cout << v2.front() << '\n';          // prints 43
```

You can prevent this by declaring the element type as `const`:

```
std::ranges::single_view<const int> v3{42};
++v3.front();                             // ERROR
```

The constructor taking a value either copies or moves it into the view. That means it cannot refer to the initial value (declaring the element type to be a reference does not compile).

Because `begin()` and `end()` yield the same value, the view is always a common range. Because iterators refer to the value stored in the view to be able to modify it, the single view is *not* a borrowed range (the lifetime of its iterators *does* depend on the view).

7.4.3 `std::ranges::empty_view`

Class template `std::ranges::`**`empty_view<>`** is a view with no elements. However, you have to specify the element type.

The overall effect is that an empty view behaves roughly like an empty vector. In fact is models the following concepts:

- std::ranges::contiguous_range (which implies std::ranges::random_access_range)
- std::ranges::sized_range

The major use case of the empty view is to call generic code with a cheap view that has no elements and for which the type system knows that it never has any elements:

```
std::ranges::empty_view<int> v1;              // empty view
for (auto val : v1) {
  ...                                         // never called
}
```

This can be used for test cases or for cases where code has to provide a view, which in a specific context never can have any elements.

For the class template also a range adaptor `std::views::empty<`*type*`>` is provided, which is a variable template, so that you declare it with a template parameter for the type but no call parameters:

```
for (auto val : std::views::empty<int>) {     // iterate over no int values
  ...                                         // never called
}
```

Table *Operations of Class* `std::ranges::empty_view<>` lists the API of an empty view.

Operation	Effect
empty_view`<`*type*`>` *v*	Creates a view with no elements of type *type*
v`.begin()`	Yields a raw pointer to the element type initialized with the `nullptr`
v`.end()`	Yields a raw pointer to the element type initialized with the `nullptr`
v`.empty()`	Yields `true`
`if (`*v*`)`	Always `false`
v`.size()`	Yields `0`
v`.front()`	always undefined behavior (fatal runtime error)
v`.back()`	always undefined behavior (fatal runtime error)
v`[`*idx*`]`	always undefined behavior (fatal runtime error)
r`.data()`	Yields a raw pointer to the element type initialized with the `nullptr`

Table 7.9. Operations of Class `std::ranges::empty_view<>`

Both `begin()` and `end()` simply always yield the `nullptr`, so that the range is also a common range and a borrowed range (the lifetime of its iterators does not depend on any view).

You might wonder why an empty view provides `front()`, `back()`, `operator[]` that ***always*** have undefined behavior, so that calling them is ***always*** a fatal runtime error. However, remember that generic code always have to check (or know) that there are elements before calling `front()`, `back()`, or operator `[]`, so that this generic code would never call these member functions. Such code will compile even for empty views.

For example, you can pass an empty view to code like this and it will compile:

```
void foo(std::ranges::random_access_range auto&& rg)
{
  std::cout << "sortFirstLast(): \n";
  std::ranges::sort(rg);
  if (!std::ranges::empty(rg)) {
    std::cout << "  first: " << rg.front() << '\n';
```

```
    std::cout << "  last:  " << rg.back() << '\n';
  }
}

foo(std::ranges::empty_view<int>{});  // OK
```

7.4.4 IStream View

Class template `std::ranges::`**`basic_istream_view<>`** is a view that reads elements from an input stream (such as the standard input, from a file, or from a string stream).

For example:

```
std::istringstream mystream{"0 1 2 3 4"};
for (const auto& elem : std::ranges::istream_view<int>(mystream)) {  // OK
  std::cout << elem << ' ';
}
std::cout << '\n';
```

Unfortunately, the current specification of `istream_view` break almost all conventions we have for streams and ranges:

- You can fully specify all template parameters:
  ```
  std::ranges::basic_istream_view<int, char, std::char_traits<char>> v1{mystream};
  ```
- The last template parameter has a working default value, so that you can use:
  ```
  std::ranges::basic_istream_view<int, char> v2{mystream};  // OK
  ```
- However, currently you cannot skip more:
  ```
  std::ranges::basic_istream_view<int> v3{mystream};        // ERROR
  ```
- Also the usual convention, that we have a special type without the `basic_` prefix for `char`'s is not satisfied:
  ```
  std::ranges::istream_view<int> v4{mystream};              // ERROR
  ```
- Even worse, a helper function with this name exists in namespace `std::ranges`:
  ```
  auto v5 = std::ranges::istream_view<int>(mystream);  // OK (uses helper function)
  ```
- And no corresponding range adaptor is provided:
  ```
  auto v6 = std::views::istream<int>(mystream);     // ERROR: not supported
  ```

***** **This chapter/section is at work** *****

7.4.5 String View

Class template `std::`**`basic_string_view<>`** and its specialization `std::`**`string_view<>`** is the only view that was already available in the C++ standard before C++20. It was introduced with C++17. For that reason, it does not follow all conventions of ranges:

- It is defined in namespace `std` (instead of `std::ranges`)

- It only provides read-only access to the elements/characters.

For example:

```
for (char c : std::string_view{"hello"}) {
  std::cout << c << ' ';
}
std::cout << '\n';
```

This loop has the following output:

```
h e l l o
```

In addition to the usual interface of views the type also provide the full API of all read-only operations of strings.

Iterators do not refer to this view. Instead they refer to the underlying character sequence. Therefore, a string view is a borrowed range. However, note that iterators can still dangle, when the underlying character sequence is no longer there.

For more details look into my book *C++17 - The Complete Guide*.

7.5 Filtering Views

This section discusses all views that filter out elements of a given range or view.

7.5.1 Take View

Class template `std::ranges::take_view<>` defines a view that refers to the first *n* first elements of a passed range. If the passed range has not enough elements, the view refers to all elements.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::take_view{rg, 5}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 2 3 4 5
```

The view can also be created with a range adaptor:

`std::views::take(rg, n)`

For example:

```
for (const auto& elem : std::views::take(rg, 5)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::take(5)) {
  std::cout << elem << ' ';
}
```

******* This chapter/section is at work *******

Bug in VC++?:

```
// ERROR with VC++:
std::list<int> rg{1, 2, 3, 4, 5, 6};
std::ranges::take_view v{rg, 5};
for (const auto& elem : v) {  // ERROR HERE with VC++
  std::cout << elem << '\n';
}
```

7.5.2 Take-While View

Class template `std::ranges::take_while_view<>` defines a view that refers to all leading elements of a passed range that match a certain predicate.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::take_while_view{rg, [](auto x) {
                                                           return x % 3 != 0;
                                                         }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 2
```

The view can also be created with a range adaptor:

`std::views::take_while(rg, pred)`

For example:

```
for (const auto& elem : std::views::take_while(rg, [](auto x) {
                                                     return x % 3 != 0;
                                                   })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::take_while([](auto x) {
                                                      return x % 3 != 0;
                                                    })) {
  std::cout << elem << ' ';
}
```

***** **This chapter/section is at work** *****

7.5.3 Drop View

Class template `std::ranges::drop_view<>` defines a view that refers to all but the first *n* first elements of a passed range. It yields the opposite elements as the take view.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::drop_view{rg, 5}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
6 7 8 9 10 11 12 13
```

The view can also be created with a range adaptor:

`std::views::drop(rg, n)`

For example:

```
for (const auto& elem : std::views::drop(rg, 5)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::drop(5)) {
  std::cout << elem << ' ';
}
```

******* This chapter/section is at work *******

Drop View and `const`

Note that you can ***not always*** iterate over a `const` drop view. In fact, the referenced range has to be a random-access range and support size().

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};
std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::drop(3));   // OK
printElems(lst | std::views::drop(3));   // ERROR
```

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& view) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(lst | std::views::drop(3));   // OK
```

7.5.4 Drop-While View

Class template `std::ranges::drop_while_view<>` defines a view that skips all leading elements of a passed range that match a certain predicate. It yields the opposite elements as the take-while view.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
```

```
for (const auto& elem : std::ranges::drop_while_view{rg, [](auto x) {
                                                           return x % 3 != 0;
                                                         }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
3 4 5 6 7 8 9 10 11 12 13
```

The view can also be created with a range adaptor:

`std::views::drop_while(rg, pred)`

For example:

```
for (const auto& elem : std::views::drop_while(rg, [](auto x) {
                                                      return x % 3 != 0;
                                                    })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::drop_while([](auto x) {
                                                       return x % 3 != 0;
                                                     })) {
  std::cout << elem << ' ';
}
```

***** **This chapter/section is at work** *****

Drop-While View and `const`

Note that you ***cannot*** iterate over a `const` drop view.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while(...));    // ERROR
```

The problem is that `begin()` is only provided for a non-`const` drop-while view because using the iterator modifies the view.

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& view) {
```

```
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while(...));   // OK
```

7.5.5 Filter View

Class template `std::ranges::`**`filter_view<>`** defines a view that can iterate over the elements of an underlying that match a certain predicate. That is, it filters out all elements that do not match the predicate.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::filter_view{rg, [](auto x) {
                                                       return x % 3 != 0;
                                                     }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 2 4 5 7 8 10 11 13
```

The view can also be created with a range adaptor:

`std::views::`**`filter(`***`rg`***`, `***`pred`***`)`**

For example:

```
for (const auto& elem : std::views::filter(rg, [](auto x) {
                                                 return x % 3 != 0;
                                               })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::filter([](auto x) {
                                                  return x % 3 != 0;
                                                })) {
  std::cout << elem << ' ';
}
```

The filter view is very special and you should know how, where to use it, how to use it, and the side effects it use has. In fact, it has a significant impact on the performance of pipelines and restricts write access to elements in a sometimes surprising way.

So you should use it with care:

- Prefer to have it at the beginning of a pipeline.
- Be careful with expensive transformations ahead of filters.

- Do not use it for write access to elements when the write access breaks the predicate.

Filter View and `const`

Note that you ***cannot*** iterate over a `const` filter view.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::filter(...));    // ERROR
```

The problem is that `begin()` is only provided for a non-`const` filter view because using the iterator modifies the view.

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& view) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::filter(...));    // OK
```

Filter Views in Pipelines

When using a filter view in a pipeline there are a couple of issues to consider:

- Prefer to have it at the beginning of a pipeline.
- Be careful with expensive transformations ahead of filters.

The reason is that views and adaptors in front of the filter view might have to evaluate elements multiple times, once to decide whether they pass the filter and once to finally use their value.

So, a pipeline such as the following:

```
rg | std::views::filter(pred) | std::views::transform(func)
```

has a better performance than

```
rg | std::views::transform(func) | std::views::filter(pred)
```

***** **This chapter/section is at work** *****

7.6 Transforming Views

This section discusses all views that yield modified values of the elements they iterate over.

7.6.1 Transform View

Class template `std::ranges::`**`transform_view<>`** defines a view that yields all elements of an underlying range after applying a passed transformation.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
// print all elements squared:
for (const auto& elem : std::ranges::transform_view{rg, [](auto x) {
                                                          return x * x;
                                                        }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 4 9 16 25 36 49 64 81 100 121 144 169
```

The view can also be created with a range adaptor:

`std::views::`**`transform(`***rg*`, `*pred*`)`**

For example:

```
for (const auto& elem : std::views::transform(rg, [](auto x) {
                                                    return x * x;
                                                  })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::transform([](auto x) {
                                                     x * x;
                                                   })) {
  std::cout << elem << ' ';
}
```

******* This chapter/section is at work *******

7.6.2 Elements View

Class template `std::ranges::`**`elements_view<>`** defines a view that selects the *idx*-th member/attribute/element of all elements of a passed range. The view calls `get<`*idx*`>(`*elem*`)` for each element and can especially be used

- to get the *idx*-th members of all `std::tuple` elements
- to get the *idx*-th alternative of all `std::variant` elements
- to get the `first` or `second` member of `std::pair` elements (although, for elements of associative and unordered containers it is more convenient to use the `keys_view` and the `values_view`)

Note that for this view class template argument deduction does not work because you explicitly have to specify the index as argument. For example:

```
std::vector<std::tuple<std::string, std::string, int>> rg{
    {"Bach", "Johann Sebastian", 1685}, {"Mozart", "Wolfgang Amadeus", 1756},
    {"Beethoven", "Ludwig van", 1770}, {"Chopin", "Frederic", 1810},
};

for (const auto& elem
      : std::ranges::elements_view<decltype(std::views::all(rg)), 2>{rg}) {
    std::cout << elem << ' ';
}
```

The loop prints:

```
1685 1756 1770 1810
```

Note that you have to specify the type of the underlying range with the range adaptor `all()`:

```
std::ranges::elements_view<decltype(std::views::all(rg)), 2>{rg}      // OK
```

You can also specify the type directly using `std::views::all_t<>`:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)&>, 2>{rg}   // OK
std::ranges::elements_view<std::views::all_t<decltype((rg))>, 2>{rg}  // OK
```

However, details matter here. If the range is no view yet, the parameter to `all_t<>` must be an lvalue reference. Therefore, you need a `&` after the type of `rg` or the double parentheses around `rg` (by rule, `decltype` yields an lvalue reference if an expression is passed that is an lvalue). Single parentheses do not work:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)>, 2>{rg}    // ERROR
```

The view can also be created with a range adaptor:

```
std::views::elements<idx>(rg)
```

The adaptor makes the use of the view a lot easier, because you do not have to specify the type of the underlying range. For example:

```
for (const auto& elem : std::views::elements<2>(rg)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::elements<2>) {
  std::cout << elem << ' ';
}
```

Using the range adaptor `elements<idx>(rg)` is always equivalent to:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)&>, idx>{rg}
```

***** This chapter/section is at work *****

7.6.3 Keys View

Class template `std::ranges::keys_view<>` defines a view that selects the first member/attribute/element from the elements of a passed range. It is nothing but a shortcut for using the std::ranges::elements_view with the index `0`. That is, it calls `get<0>(elem)` for each element. It can especially be used

- to get the `first` member of `std::pair` elements, which is especially useful to select the key of the elements of a `map`, `unordered_map`, `multimap`, and `unordered_multimap`.
- to get the first member of `std::tuple` elements
- to get the first alternative of `std::variant` elements

However, for the last two applications, it is probably more readable to use the `elements_view` directly with the index `0`.

Note that for this view class template argument deduction does not the way it is currently specified.[3] For this reason you have to specify the template parameters explicitly. For example:

```
std::map<std::string, int> rg{
    {"Bach", 1685}, {"Mozart", 1756}, {"Beethoven", 1770},
    {"Tchaikovsky", 1840}, {"Chopin", 1810}, {"Vivaldi", 1678},
};

for (const auto& elem : std::ranges::keys_view<decltype(std::views::all(rg))>{rg}) {
    std::cout << elem << ' ';
}
```

The loop prints:

```
Bach Beethoven Chopin Mozart Tchaikovsky Vivaldi
```

The view can also be created with a range adaptor:

```
std::views::keys(rg)
```

The adaptor makes the use of the view a lot easier, because you do not have to specify the type of the underlying range. For example:

```
for (const auto& elem : std::views::keys(rg)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::keys) {
  std::cout << elem << ' ';
}
```

***** This chapter/section is at work *****

[3] See http://wg21.link/lwg3563.

7.6.4 Values View

Class template `std::ranges::`**`values_view<>`** defines a view that selects the first member/attribute/element from the elements of a passed range. It is nothing but a shortcut for using the std::ranges::elements_view with the index `0`. That is, it calls `get<0>(`*`elem`*`)` for each element. It can especially be used

- to get the `first` member of `std::pair` elements, which is especially useful to select the value of the elements of a `map`, `unordered_map`, `multimap`, and `unordered_multimap`.
- to get the first member of `std::tuple` elements
- to get the first alternative of `std::variant` elements

However, for the last two applications, it is probably more readable to use the `elements_view` directly with the index `1`.

Note that for this view class template argument deduction does not the way it is currently specified.[4] For this reason you have to specify the template parameters explicitly. For example:

```
std::map<std::string, int> rg{
    {"Bach", 1685}, {"Mozart", 1756}, {"Beethoven", 1770},
    {"Tchaikovsky", 1840}, {"Chopin", 1810}, {"Vivaldi", 1678},
};

for (const auto& elem : std::ranges::values_view<decltype(std::views::all(rg))>{rg}) {
    std::cout << elem << ' ';
}
```

The loop prints (note that the elements are sorted according to their names):

```
1685 1770 1810 1756 1840 1678
```

The view can also be created with a range adaptor:

`std::views::`**`values(`***`rg`***`)`**

The adaptor makes the use of the view a lot easier, because you do not have to specify the type of the underlying range. For example:

```
for (const auto& elem : std::views::values(rg)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::values) {
  std::cout << elem << ' ';
}
```

******* This chapter/section is at work *******

[4] See http://wg21.link/lwg3563.

7.7 Mutating Views

This section discusses all views that change the order of elements (so far it is only one view).

7.7.1 `std::ranges::reverse_view`

Class template `std::ranges::`**`reverse_view<>`** defines a view that iterates over the elements of the underlying range in the opposite order.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::reverse_view{rg}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
13 12 11 10 9 8 7 6 5 4 3 2 1
```

The view can also be created with a range adaptor:

`std::views::`**`reverse(`***`rg`***`)`**

For example:

```
for (const auto& elem : std::views::reverse(rg)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::reverse) {
  std::cout << elem << ' ';
}
```

******* This chapter/section is at work *******

Reverse View and `const`

Note that you can ***not always*** iterate over a `const` reverse view. In fact, the referenced range has to have the same type for begin and end iterator.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};
```

```
// leading odd elements of vec:
auto vecFirstOdd = std::views::take_while(vec, [](auto x) {
                                                 return x % 2 != 0;
                                               });

printElems(vec | std::views::reverse);          // OK
printElems(vecFirstOdd);                        // OK
printElems(vecFirstOdd | std::views::reverse);  // ERROR
```

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& view) {
  ...
}

std::vector vec{1, 2, 3, 4, 5};

// leading odd elements of vec:
auto vecFirstOdd = std::views::take_while(vec, [](auto x) {
                                                 return x % 2 != 0;
                                               });

printElems(vecFirstOdd | std::views::reverse);  // OK
```

7.8 Views for Multiple Ranges

This section discusses all views that deals with multiple ranges.

7.8.1 Split and Lazy-Split View

Class templates `std::ranges::`**`split_view<>`** and `std::ranges::`**`lazy_split_view<>`** define a view that refers to multiple sub-views of a range separated by a passed separator.[5]

The difference between `split_view<>` and `lazy_split_view<>` is as follows:

- `split_view<>` cannot iterate over a `const` filter view; `lazy_split_view<>` can.
- `split_view<>` can only deal with ranges that have at least forward iterators (concept `forward_range` has to be satisfied).
- `split_view<>` yields `std::ranges::subrange`'s of the iterator type of the original range, which have the category of the source range. `lazy_split_views<>` can only yield the sub-ranges as forward ranges so that even `size()` is not supported.
- `split_view<>` has the better performance.

That is, usually you should use `split_view<>` unless you cannot do that due to using an input range only or the range is `const`.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& sub : std::ranges::split_view{rg, 5}) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

The loops print:

```
1 2 3 4
6 7 8 9 10 11 12 13
```

That is, wherever we find an element with value 5 in `rg`, we end the previous view and start a new view.

The view can also be created with range adaptors:

```
std::views::split(rg, sep)
std::views::lazy_split(rg, sep)
```

For example:

```
for (const auto& sub : std::views::split(rg, 5)) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
```

[5] `std::ranges::lazy_split_view<>` was not part of the original C++20 standard, but added to C++20 afterwards with http://wg21.link/p2210r2.

```
  }
  std::cout << '\n';
}
```

or:

```
for (const auto& sub : rg | std::views::split(5)) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

The created views might be empty. Thus, for each leading and trailing separator and whenever two separators are behind each other, an empty view gets created. For example:

```
std::list<int> rg{5, 5, 1, 2, 3, 4, 5, 6, 5, 5, 4, 3, 2, 1, 5, 5};
for (const auto& sub : std::ranges::split_view{rg, 5}) {
  std::cout << "subview: ";
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Here, the output is as follows:[6]

```
subview:
subview:
subview: 1 2 3 4
subview: 6
subview:
subview: 4 3 2 1
subview:
subview:
```

Instead of a separator, you can also pass a sequence of elements that serves as separator. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3 };
...
// split separated by a sequence of 4 and 5:
for (const auto& sub : std::views::split(rg, std::views::iota(4, 6))) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

[6] The original C++20 standard specified that the last separator is ignored so that we would get only one empty subview at the end. This was fixed with http://wg21.link/p2210r2.

The output of this code is

```
1 2 3 4
1 2 3 4
1 2 3
```

The passed collection of elements must be a valid view and a `forward_range`. So, when specifying the sub-sequence in a container you have to convert it into a view. For example:

```
// split with specified pattern of 4 and 5:
std::array pattern{4, 5};
for (const auto& sub : std::views::split(rg, std::views::all(pattern))) {
  ...
}
```

By using string views, you can use this to split string. For example:

```
std::string str{"No problem can withstand the assault of sustained thinking"};
for (auto sub : std::views::split(str, "th"sv)) {  // split by "th"
  std::cout << std::string_view{sub} << '\n';
}
```

Each sub-string `sub` is of type `std::ranges::subrange<decltype(str.begin())>`. Code like this will not work with a lazy-split view.

******* This chapter/section is at work *******

Split View and `const`

Note that you ***cannot*** iterate over a `const` split view.

For example:

```
std::vector<int> coll{5, 1, 5, 1, 2, 5, 5, 1, 2, 3, 5, 5, 5};
...
const std::ranges::split_view sv{coll, 5};
for (const auto& sub : sv) {                // ERROR for const split view
  std::cout << sub.size() << ' ';
}
```

This especially is a problem if you pass the view to a generic function taking the parameter as a `const` reference:

```
void printElems(const auto& view) {
  ...
}

printElems(std::views::split(rg, 5));       // ERROR
```

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& view) {
  ...
```

```
}
```

Alternatively, you can use a `lazy_split_view`. However, then the sub-views can only be used as forward ranges so that you cannot do things like calling `size()`, iterating backwards, or sorting the elements:

```
std::vector<int> coll{5, 1, 5, 1, 2, 5, 5, 1, 2, 3, 5, 5, 5};
...
const std::ranges::lazy_split_view sv{coll, 5};
for (const auto& sub : sv) {                // OK for const lazy-split view
  std::cout << sub.size() << ' ';           // ERROR
  std::sort(sub);                           // ERROR
  for (const auto& elem : sub) {            // OK
    std::cout << elem << ' ';
  }
}
```

7.8.2 Join View

Class template `std::ranges::`**`join_view<>`** defines a view iterates over all element of a view of multiple ranges.

For example:

```
std::vector<int> rg1{1, 2, 3, 4};
std::vector<int> rg2{0, 8, 15};
std::vector<int> rg3{5, 4, 3, 2, 1, 0};
std::array coll{rg1, rg2, rg3};
...
for (const auto& elem : std::ranges::join_view{coll}) {
  std::cout << elem << ' ';
}
```

The loops print:

```
1 2 3 4 0 8 15 5 4 3 2 1 0
```

The view can also be created with a range adaptor:

`std::views::`**`join(`***`rg`***`)`**

For example:

```
for (const auto& elem : std::views::join(coll)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : coll | std::views::join) {
  std::cout << elem << ' ';
}
```

Together with type `subrange` you can use it to join elements of multiple arrays. For example:

```
int arr1[]{1, 2, 3, 4, 5};
```

```
int arr2[] = {0, 8, 15};
int arr3[10]{1, 2, 3, 4, 5};
...
std::array<std::ranges::subrange<int*>, 3> coll{arr1, arr2, arr3};
for (const auto& elem : std::ranges::join_view{coll}) {
  std::cout << elem << ' ';
}
```

Alternatively, you can declare `coll` as follows:

```
std::array coll{std::ranges::subrange{arr1},
                std::ranges::subrange{arr2},
                std::ranges::subrange{arr3}};
```

******* This chapter/section is at work *******

Join View and `const`

Note that you can ***not always*** iterate over a `const` join view. In fact, the referenced range must not be a generating range.

******* This chapter/section is at work *******

7.9 Open

P1035 Input range adaptors
- P1252 Ranges Design Cleanup
- P1391 Range constructor for std::string_view
- P1394 Range constructor for std::span
- P1664 Reconstructible Ranges

7.10 Afternotes

This page is intentionally left blank

Chapter 8

Spans

With C++20, a couple of types were introduced to refer to sequences of elements without allocating memory for them. The `std::span<>` class template is one of them. By requiring that the elements are stored in contiguous memory, the class provides the best performance for subsequences.

Objects of type `std::span<>` refer to external sequences of elements All elements have to be stored one after another in contiguous memory. You can have read-only or write access. And you can choose between a specified fixed number of elements or leave the number of elements open until it is clear to which elements the span refers to.

A span object is in fact just as raw pointer to a sequence of elements combined with a size for the number of elements. That allows us to deal with the elements like with an array. However, special rules apply due to its reference semantics (especially regarding `const` correctness).

Using a span is cheap and fast (you should always pass it by value). However, it is also potentially dangerous because, in the same way as for raw pointers, it is up to the programmer to ensure that the referred element sequence is still valid when using a span. In addition, the fact that a span supports write access can introduce situations where `const` correctness is broken (or at least does not work the way you might expect).

8.1 Using Spans

Let us look at some first examples of using spans. However, first we have to discuss how the number of elements in a span can be specified.

8.1.1 Fixed and Dynamic Extent

When you declare a span, you can choose between a specified fixed number of elements or leave the number of elements open until it is clear to which elements the span refers to.

A span with a specified fix number of elements is called a span with *fixed extent*. It is declared with the element type and the size (like a `std::array<>`).

```
std::span<int, 5>    // type of a span with with fixed extent of 5 elements
```

For such a span the member function `size()` always yields the size specified as part of the type. The default constructor is not callable here (unless the extent is `0`).

A span where the number of elements might change is called a span with *dynamic extent*. The number of elements depends on the sequence of elements the span refers to and might change by using the assignment operator. However, there is no other way to change the extent (number of elements).

For both cases, let us come with a first example.

8.1.2 Example Using a Span with Fixed Extent

Here is a first example, using a span with fixed extent:

lib/spanfix.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>
#include <cassert>

template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

  // define view to first 3 elements:
  assert(vec.size() >= 3);
  std::span<const std::string, 3> sp3{vec.begin(), 3};
  std::cout << "first 3: ";
  printSpan(sp3);

  // sort referenced elements:
  std::ranges::sort(vec);
  std::cout << "first 3 after sort(): ";
  printSpan(sp3);
```

```
  // insert new element
  // - have to re-assign the span if vector did allocate new memory
  auto oldCapa = vec.capacity();
  vec.push_back("Cairo");
  if (oldCapa != vec.capacity()) {
    sp3 = std::span<std::string, 3>{vec.begin(), 3};
  }
  std::cout << "first 3: ";
  printSpan(sp3);

  // assign view to last 3 elements:
  sp3 = std::span<std::string, 3>{vec.end() - 3, vec.end()};
  std::cout << "last 3: ";
  printSpan(sp3);

  // pass vector as span:
  std::cout << "vec: ";
  printSpan(std::span{vec});  // OK (explicit conversion to std::span necessary)
}
```

The output of the program is as follows:

```
first 3: "New York" "Tokyo" "Rio"
first 3 after sort(): "Berlin" "New York" "Rio"
first 3: "Berlin" "New York" "Rio"
last 3: "Sydney" "Tokyo" "Cairo"
vec: "Berlin" "New York" "Rio" "Sydney" "Tokyo" "Cairo"
```

Let us go through the example step by step.

Declaring a Span

In `main()`, we first initialize a span of 3 constant strings with the first 3 elements of a vector:

```
std::vector<std::string> vec{"New York", "Rio", "Tokyo", "Berlin", "Sydney"};

assert(vec.size() >= 3);
std::span<const std::string, 3> sp3{vec.begin(), 3};
```

For initialization we pass the begin of the sequence and the number of elements. In this case we refer to the first 3 elements of `vec`.

Just in this declaration there are a lot of things to note:

- It is up to the programmer that the number of elements matches with the extent of the span and that the elements are valid. If the vector has not enough elements or the count passed as second argument does not match the extent, the behavior is undefined.

```
std::span<const std::string, 3> sp3{vec.begin(), 4};  // undefined behavior
```

- ***Never*** initialize a span with a (returned) temporary value, because the span would then refer to an object that is destroyed after the assignment:

```
std::span<int, 3> sp3a = std::array{1, 2, 3};   // fatal runtime ERROR
std::span<int, 3> sp3b = returnArray();         // fatal runtime ERROR
```

- By specifying that the elements are `const std::string`, we cannot modify them via the span. Note that declaring the span as `const` does ***not*** provide read-only access to the references elements:

```
std::span<const std::string, 3> sp3{vec.begin(), 3};   // elements cannot be modified
const std::span<std::string, 3> sp3{vec.begin(), 3};   // elements can be modified
```

- Using a different element type for the span than for the referenced elements looks like you can use any type for a span that converts to the underlying element type. However, that is not true. You can only add qualifiers such as `const`:

```
std::vector<int> vec{...};
std::span<long, 3> sp3{vec.begin(), 3};  // ERROR
```

- You could also use `data()` instead of `begin()` here. The important thing is that the passed sequence is valid.

  However, using `begin()` provides better options for compilers that try to detect undefined behavior by passing an invalid number of elements.

Passing and Printing a Span

Next, we print the span, passing it to a generic print function:

```
printSpan(sp3);
```

The print function can deal with any span (as long as an output operator for the elements is defined):

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}
```

As you can see, the span is passed by value. That is the recommended way because internally a span is just a pointer and a size.

Inside the function, we use a range-based `for` loop to iterate over the elements of the span. This is possible because a span provides iterator support with `begin()` and `end()`.

However, beware: whether we pass the span by value or by `const` reference, inside the function we can still modify the elements as long as they are not declared to be `const`.

Dealing with Reference Semantics

Next, we sort the elements the span refers to (we use the new std::ranges::sort here that takes the container as a whole):

```
std::ranges::sort(vec);
```

Because the span has reference semantics, this sorting also affects the elements of the span: they are also sorted.

If we do not have a span of `const` elements, we could also call `sort()` for the span. In that case we would only sort the first 3 elements in the vector:

```
std::vector<std::string> vec{"New York", "Rio", ...};  // elements not const

std::span<const std::string, 3> span3{vec.begin(), 3};

std::ranges::sort(span3);  // sort first 3 elements only
```

Next, we insert a new element into the vector holding the elements the span refers to. Due to the reference semantics of a span, this is something we have to be very careful about, because if the vector allocates new memory it invalidates all iterators and pointers to its elements. Therefore, a reallocation also invalidates a span referring to the elements of the vector. The span refers to elements that are no longer there.

For this reason, we double check the capacity (maximum number of elements for which memory is allocated) before and after the insertion. If it changed, we reinitialize the span to refer to the first three elements in the new memory:

```
auto oldCapa = vec.capacity();
vec.push_back("Cairo");
if (oldCapa != vec.capacity()) {
  span3 = std::span<std::string, 3>{vec.begin(), 3};
}
```

We can only perform this reinitialization, because the span itself is not `const`.

Assigning a Different Subsequence

In general, the assignment operator of a span allows us to assign another sequence of elements. The requirement is that the assigned value has the same number of elements of the same type stored in contiguous memory.

The example uses this to later refer to the last three elements in the vector:

```
span3 = std::span<std::string, 3>{vec.end() - 3, vec.end()};
```

Here you can also see that we can specify the reference sequence with two iterators defining the begin and the end of the sequence as a half-open range (begin included, end excluded). In this case the begin is an iterator 3 elements before the end.

However, the new value can also come from a totally different source. For example:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
...
std::span<const std::string, 3> sp3{vec.begin(), 3};
```

```
...
std::array<std::string, 3> arr{"Tick", "Trick", "Track"};
sp3 = arr;  // OK
```

Note two things:

- This code would not compile:

```
std::span<const std::string, 3> sp3{vec.begin(), 3};
...
std::array arr{"Tick", "Trick", "Track"};  // deduces std::array<const char*, 3>
sp3 = arr;  // ERROR: different element type
```

- ***Never*** assign a (returned) temporary value, because the span would then refer to an object that is destroyed after the assignment:

```
sp3 = std::array<std::string, 3>{...};  // fatal runtime ERROR
sp3 = returnArray();                    // fatal runtime ERROR
```

Using a Container as a Span

Finally, we try to pass the whole vector `vec` to the generic function printing a span:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
std::span<const std::string, 5> sp5{vec};  // OK
```

Spans can take containers with elements in contiguous memory as a whole, provided the container yields access to the elements with the member function `data()`.

However, due to limits of template type deduction, you cannot pass such a container to a function expecting a span. For this reason you have to explicitly specify that you want to convert the vector to a span:

```
printSpan(vec);              // ERROR: template type deduction doesn't work here
printSpan(std::span{vec});   // OK
```

In the specified explicit conversion we do not specify the elements type and extent of the span. We use class template argument deduction here, which is supported by a span. However, because the vector does not have a fixed size, we create a span with a dynamic extent here, which we then pass to the print function:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
std::span sp{vec};  // deduces std::span<std::string>
printSpan(sp);
```

This works, because spans provides constructors that support implicit type conversions from other spans (provided the conversion is not between spans of different fixed extent).

8.1.3 Example Using a Span with a Dynamic Extent

As a first example of a span with a dynamic extent, let us convert the previous example just using a span with a dynamic extent:

lib/spandyn.cpp

```
#include <iostream>
```

```
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>
#include <cassert>

template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

  // define view to first 3 elements:
  assert(vec.size() >= 3);
  std::span<const std::string> sp{vec.begin(), 3};
  std::cout << "first 3: ";
  printSpan(sp);

  // sort referenced elements:
  std::ranges::sort(vec);
  std::cout << "first 3 after sort(): ";
  printSpan(sp);

  // insert new element
  // - have to re-assign the span if vector did allocate new memory
  auto oldCapa = vec.capacity();
  vec.push_back("Cairo");
  if (oldCapa != vec.capacity()) {
    sp = std::span{vec.begin(), 3};
  }
  std::cout << "first 3: ";
  printSpan(sp);

  // assign view to last 3 elements:
  sp = std::span{vec.end() - 3, vec.end()};
  std::cout << "last 3: ";
  printSpan(sp);
```

```
  // pass vector as span:
  std::cout << "vec: ";
  printSpan(std::span{vec});   // OK (explicit conversion to std::span necessary)

  // assign sequence with different number of elements:
  std::cout << "sp: ";
  sp = vec;
  printSpan(sp);
}
```

The output of the program is as follows:

```
first 3: "New York" "Tokyo" "Rio"
first 3 after sort(): "Berlin" "New York" "Rio"
first 3: "Berlin" "New York" "Rio"
last 3: "Sydney" "Tokyo" "Cairo"
vec: "Berlin" "New York" "Rio" "Sydney" "Tokyo" "Cairo"
sp: "Berlin" "New York" "Rio" "Sydney" "Tokyo" "Cairo"
```

Again, let us go through the example step by step.

Declaring a Span

In `main()`, we first initialize a span. However, this time we do not specify a fixed extent:

```
std::vector<std::string> vec{"New York", "Rio", "Tokyo", "Berlin", "Sydney"};

assert(vec.size() >= 3);
std::span<const std::string> sp3{vec.begin(), 3};
```

We could even use CTAD here, so that the span deduces the type of the elements:

```
std::span sp3{vec.begin(), 3};   // deduces std::span<std::string>
```

The effect is that the elements have to be strings, but we can assign sequences with a different number of elements.

As for spans with a fixed extent:

- It is up to the programmer to ensure that the passed sequence of elements is valid.
- ***Never*** initialize a span with a (returned) temporary value.
- By specifying that the elements are `const std::string`, we cannot modify them via the span.

```
std::span<const std::string> sp3{vec.begin(), 3};   // elements cannot be modified
const std::span<std::string> sp3{vec.begin(), 3};   // elements can be modified
```

- Using a different element type for the span than for the referenced elements is only possible by adding qualifiers (such as `const`).
- You could also use `data()` instead of `begin()` here.

Passing and Printing a Span

Next, we print the span, passing it to a generic print function:

```
printSpan(sp3);
```

You might be surprised that the function template `printSpan<>()` still can be called although is has a template parameter for the size of the span:

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  ...
}
```

It works because a `std::span<T>` is a shortcut for a span having the pseudo size `std::dynamic_extent`:

```
std::span<int> sp;  // equivalent to std::span<int, std::dynamic_extent
```

In fact, the class template `std::span<>` is declared as follows:

```
namespace std {
  template<typename ElementType, size_t Extent = dynamic_extent>
  class span {
    ...
  };
}
```

This way, your can provide function templates like `printSpan<>()` that work for both spans with fixed and spans with dynamic extents. When calling `printSpan<>()` with a span with a dynamic extent, `std::dynamic_extent` is passed as size:

```
std::span<int> sp;  // equivalent to std::span<int, std::dynamic_extent

printSpan(sp);      // calls printSpan<int, std::dynamic_extent>(sp)
```

Again, note that the span is passed by value, and again, beware: inside the function we can still modify the elements as long as they are not declared to be `const`.

Assigning a Container to a Span

The rest of the program is as in the version with a span of fixed extent.

However, we have added one thing at the end: we assign the vector as a whole to the span and print it out:

```
std::span<const std::string> sp{vec.begin(), 3};
printSpan(sp);    // prints first 3 elements of the vector

sp = vec;
printSpan(sp);    // prints all elements of the vector
```

Here, you can see that an assignment to the span can change the number of elements.

Using Spans with fixed vs. dynamic Extent

Both fixed and dynamic extents have their benefits.

Specifying a fixed size allows us to detect violations better (at runtime or even at compile time). For example, you cannot assign a `std::array<>` with the wrong number of elements to a span with a fixed extent:

```
std::vector vec{1, 2, 3};
std::array arr{1, 2, 3, 4, 5, 6};

std::span<int, 3> sp3{vec};
std::span sp{vec};

sp3 = arr;   // compile-time ERROR
sp = arr;    // OK
```

Using spans with dynamic extents provides more flexibility:

```
std::span<int> sp;          // OK
...
std::vector vec{1, 2, 3, 4, 5};
sp = vec;                   // OK (span has 5 elements)
sp = {vec.begin(), 3};      // OK (span has 3 elements)
```

8.2 Spans Considered Harmful

Spans refer to a sequence of elements. As that they have the usual problems types with reference semantics have. It is up to the programmer to ensure that the sequence a span refers to is valid.

Errors happen pretty fast. For example, if a function `getData()` returns a collection of `int`s by value (e.g., as vector, `std::array`, or raw array), the following statements create fatal runtime errors:

```
std::span<int, 3> first3{getData()};     // ERROR: reference to temporary object

std::span sp{getData().begin(), 3};      // ERROR: reference to temporary object

sp = getData();                          // ERROR: reference to temporary object
```

This can look a little bit more subtle such as using the range-based `for` loop:

```
// for the first 3 returned elements:
for (const auto& elem : std::span{getData().data(), 3}) // fatal runtime ERROR
```

See http://wg21.link/p2012 why this is an issue (as long as the fix proposed in that paper is not accepted and implemented).

Compilers will probably be able to detect these problems with the "lifetime extension" which is currently being implemented for major compilers. However, that can only detect simple lifetime dependencies like the one between a span and the object it is initialized with.

In addition, you have to ensure that the referenced sequence of elements remains valid. This can be a problem if other parts of the program end the lifetime of a referred sequence while the span still is in use.

If we refer into objects (such as into a vector), this can even happen while the vector still is alive. Consider:

```
std::vector<std::string> vec{...};

std::span first3{vec.begin(), 3};   // view to first 3 elements
...
while (...) {
  vec.push_back(...);               // might reallocate memory
  std::cout << first3[0];           // fatal runtime ERROR (invalidated the referenced sequence)
}
```

As a workaround, you have to reinitialize the span with the original vector.

Using spans is in general as dangerous as using raw pointers. Use them with care.

8.3 Design Aspects of Spans

Designing a type that refers to a sequence of elements is not easy. A lot of aspects and trade-offs have to be thought out and decided about:

- Performance versus Safety
- `const` correctness
- Possible implicit and explicit type conversions
- Requirements for the supported types
- Supported API's

Let me first say something very clear:

- A span is not a container. It might have some attributes like a container (e.g., the ability to iterate over elements with `begin()` and `end()`), but there are already several issues:
  - Should elements also be `const` if the span is `const`?
  - What does an assignment do: assigning a new sequence or assigning new values to the referenced elements?
  - Should we provide `swap()` and what does it do?
- A span is not a pointer (with a size). It does not make sense to provide an `operator*`

Type `std::span` is a very specific reference to a sequence of elements. And it is important to understand these specific attributes to use it right.

Iterators of spans do not refer to the span itself. Instead they refer to the underlying range. Therefore, a span is a borrowed range. However, note that iterators can still dangle, when the underlying range is no longer there.

Note that we have other ways to deal with references to (sub-) sequences, such as subranges, which are also introduced with C++20.

There is a very helpful blog post by Barry Revzin about this I strongly recommend to read:
http://brevzin.github.io/c++/2018/12/03/span-best-span/

8.3.1 Performance of Spans

Spans are designed with best performance as a goal. For this, they internally use just a raw pointer to the sequence of elements. However, raw pointers expect elements being stored sequentially in one chunk of memory (otherwise ++ could not be used by the raw pointers to iterate over the elements). For this reason, spans require to have the elements in contiguous memory.

With this requirement, spans can provide the best performance of all types of views. Spans do not need any allocation and do not come with any indirection. The only overhead on using a span is the overhead of constructing it.[1]

Spans check whether a referred sequence has its elements in contiguous memory via concepts at compile time. When a sequence is initialized or a new sequence is assigned, iterators have to satisfy the concept std::contiguous_iterator and containers/ranges have to satisfy the concepts `std::ranges::contiguous_range` and `std::ranges::sized_range`.

Because spans internally are just a pointer and a size, it is very cheap to copy them. For this reason, you should always prefer to pass spans by value instead of passing them by `const` reference.

Type Erasure

The fact that spans perform element access with raw pointers to the memory means that a span type erases the information in which container these elements are. A span to the elements of a vector has the same type as a span to the elements of an array (provided they have the same extent):

```
std::array arr{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5};

std::span<int> vecSpanDyn{vec};
std::span<int> arrSpanDyn{arr};
std::same_as<decltype(arrSpanDyn), decltype(vecSpanDyn)>    // true
```

However, note that class template argument deduction for spans deduces a fixed extent from arrays and a dynamic extent from vectors. That means:

```
std::array arr{1, 2, 3, 4, 5};    // deduces std::span<int, 5>
std::vector vec{1, 2, 3, 4, 5};   // deduces std::span<int>
std::span<int, 5> vecSpan5{vec};

std::same_as<decltype(arrSpan), decltype(vecSpan)>          // false
std::same_as<decltype(arrSpan), decltype(vecSpan5)>         // true
```

Spans vs. Subranges

The requirement for contiguous storage of the elements is the major difference to subranges, which are also introduced with C++20. Subranges internally still use iterators and can therefore refer to all types of containers and ranges. However, that may lead to significant more overhead.

[1] Thanks to Barry Revzin for pointing that out.

In addition, spans do not require iterator support of the type they refer to. You can pass any type that provides a `data()` member for access to a sequence of elements.

8.3.2 `const` Correctness of Spans

Spans are views that have reference semantics. In that sense, they behave like pointers: when a span is `const` it does not automatically mean that the elements the span refers to are `const`.

This means that you have write access to the elements of a `const` span (provided the elements are not `const`):

```
std::array a1{1, 2, 3, 4, 5, 6,7 ,8, 9, 10};
std::array a2{0, 8, 15};

const std::span<int> sp1{a1};   // span/view is const
std::span<const int> sp2{a1};   // elements are const

sp1[0] = 42;    // OK
sp2[0] = 42;    // ERROR

sp1 = a2;       // ERROR
sp2 = a2;       // OK
```

Note that as long as the elements of a `std::span<>` are not declared to be `const`, a couple of operations provide write access to the elements even for `const` spans where you might not expect it (following the rules for ordinary containers):

- `operator[]`, `first()`, `last()`
- `data()`
- `begin()`, `end()`, `rbegin()`, `rend()`
- `std::cbegin()`, `std::cend()`, `std::crbegin()`, `std::crend()`
- `std::ranges::cbegin()`, `std::ranges::cend()`, `std::ranges::crbegin()`, `std::ranges::crend()`

Yes, all the `c*` functions that were designed to ensure that the elements are `const` **are broken** by `std::span`.

For example:

```
template<typename T>
void modifyElemsOfConstCollection (const T& coll)
{
  coll[0] = {};   // OK for spans, ERROR for regular containers

  auto ptr = coll.data();
  *ptr = {};      // OK for spans, ERROR for regular containers

  for (auto pos = std::cbegin(coll); pos != std::cend(coll); ++pos) {
    *pos = {};    // OK for spans, ERROR for regular containers
  }
}
```

```
std::array a1{1, 2, 3, 4, 5, 6, 7 ,8, 9, 10};

modifyElemsOfConstCollection(a1);              // ERROR: elements are const
modifyElemsOfConstCollection(std::span{a1});   // OOPS: OK, modifies elements of the array
```

The problem here is not that `std::span` is broken; the problem is that functions like `std::cbegin()` and `std::ranges::cbegin()` are currently broken for collections with reference semantics (such as views).

To ensure that a function only takes sequences where you cannot modify the elements that way, you can require that `begin()` for a `const` container returns an iterator to `const` elements:

```
template<typename T>
void ensureReadOnlyElemAccess (const T& coll)
requires std::is_const_v<std::remove_reference_t<decltype(*coll.begin())>>
{
  ...
}
```

At least the fact that even `std::cbegin()` provides write access is something under discussion after standardizing C++20. The whole point of providing `cbegin()` and `cend()` is to to ensure that elements cannot be modified when iterating over them. Originally, spans did provide members for type `const_iterator`, `cbegin()`, and `cend()` to ensure that you could not modify elements. However, it turned out that `std::cbegin()` still iterate over mutable elements (and `std::ranges::cbegin()` has the same problem). But, instead of fixing `std::cbegin()` (and `std::ranges::cbegin()`), the members for `const` iterators in spans were removed (see http://wg21.link/lwg3320), which made the problem even worse because now there is no easy way anymore for a read-only iteration over a span. The correct way to fix the problem is to fix the way `std::cbegin()` is defined. We are working on fixes for C++23 (see http://wg21.link/p2276 and http://wg21.link/p2278).

8.3.3 Using Spans as Parameters in Generic Code

As written, you can implement a generic function just for all spans with the following declaration:

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp);
```

This even works for spans with dynamic extents, because they just use the special value `std::dynamic_extent` as size.

So, in the implementation you can deal with the difference of fixed and dynamic extents as follows:

lib/spanprint.hpp

```
#ifndef SPANPRINT_HPP
#define SPANPRINT_HPP

#include <iostream>
#include <span>
```

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  std::cout << '[' << sp.size() << " elems";
  if constexpr (Sz == std::dynamic_extent) {
    std::cout << " (dynamic)";
  }
  else {
    std::cout << " (fixed)";
  }
  std::cout << ':';
  for (const auto& elem : sp) {
    std::cout << ' ' << elem;
  }
  std::cout << "]\n";
}

#endif // SPANPRINT_HPP
```

You might also consider to declare the element type to be `const`:

```
template<typename T, std::size_t Sz>
void printSpan(std::span<const T, Sz> sp);
```

However, note that then you cannot pass spans with non-`const` element types. Conversions from a non-`const` type to a `const` do not propagate to templates (for good reasons).

Lack of type deduction and conversions also disable passing an ordinary container such as a vector to this function. You need either an explicit type specification or an explicit conversion:

```
printSpan(vec);                          // ERROR: template type deduction doesn't work here
printSpan(std::span{vec});               // OK
printSpan<int, std::dynamic_extent>(vec); // OK (provided it is a vector of ints)
```

For this reason, `std::span<>` should not be used as a vocabulary type for generic functions dealing with sequences of elements stored in contiguous memory.

For performance reasons, you might do something like this:[2]

```
template<typename E>
void processSpan(std::span<typename E>) {
  ...      // span specific implementation
}

template<typename T>
void print(const T& t) {
  if constexpr (std::ranges::contiguous_range<T> t) {
    processSpan<std::ranges::range_value_t<T>>(t);
```

[2] Thanks to Arthur O'Dwyer for pointing that out.

```
    }
    else {
      ...   // generic implementations for all containers/ranges
    }
  }
```

Using Spans as Ranges and Views

A span is a range and can be used in all algorithms and function for ranges. It is even a borrowed range, meaning that you can use a temporary span as a range in algorithms that yield iterators to it:

```
std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};
auto pos1 = std::ranges::find(std::span{coll.data(), 3}, 42);  // no dangling iterator
std::cout << *pos1 << '\n';
```

However, note that this is an error if the span refers to a temporary object. The following code compiles although an iterator to a destroyed temporary object is returned:

```
auto pos2 = std::ranges::find(std::span{getData().data(), 3}, 42);
std::cout << *pos2 << '\n';       // runtime ERROR
```

Spans are also *views* and model the concept `std::ranges::view`.[3]

8.4 Span Operations

This section describes the types and operations of spans in detail.

8.4.1 Span Operations and Member Types Overview

Table *Span Operations* lists all operations that are provided for spans.

At first, note all the operations that are not supported:

- Comparisons (not even ==)
- `swap()`
- `assign()`
- `at()`
- I/O operators
- `cbegin()`, `cend()`, `crbegin()`, `crend()`
- Hashing
- Tuple-like API for structured bindings

That is, spans are neither containers (in the traditional C++ STL sense) nor regular types.

[3] The original C++20 standard did require that views have to have a default constructor, which is not the case for fixed spans. However, that requirement was removed later with `http://wg21.link/P2325R3`

Operation	Effect
constructors	Creates or copies a span
destructor	Destroys a span
`=`	Assigns a new sequence of values
`empty()`	Returns whether the span is empty
`size()`	Returns the number of elements
`size_bytes()`	Returns the size of memory used for all elements
`[]`	Accesses an element
`front()`, `back()`	Accesses the first or last element
`begin()`, `end()`	Provides iterator support (no `const_iterator` support)
`rbegin()`, `rend()`	Provides constant reverse iterator support
`subspan()`	Returns a span with a certain subsequence
`first()`	Returns a sub-span with a fixed extent of the first *n* elements
`last()`	Returns a sub-span with a fixed extent of the last *n* elements
`data()`	Returns a pointer to the elements
`as_bytes()`	Returns the memory of the elements as a span of read-only `std::bytes`
`as_writable_bytes()`	Returns the memory of the elements as a span of writable `std::bytes`

Table 8.1. Span Operations

Regarding static members and member types, spans provide the usual members of containers (except `const_iterator`) plus two special ones: `element_type` and `extent` (see table *Static and Type Members of Spans*).

Member	Effect
`extent`	Number of elements or `std::dynamic_extent` if size varies
`size_type`	Type of `extent` (always `std::size_t`)
`difference_type`	Difference type for pointers to elements (always `std::difference_type`)
`element_type`	Specified type of the elements
`pointer`	Type of a pointer to the elements
`const_pointer`	Type of a pointer for read-only access to the elements
`reference`	Type of a reference to the elements
`const_reference`	Type of a reference for read-only access to the elements
`iterator`	Type of an iterator to the elements
`reverse_iterator`	Type of a reverse iterator to the elements
`value_type`	Type of elements without `const` or `volatile`

Table 8.2. Static and Type Members of Spans

Note that `std::value_type` is ***not*** the specified element type (as the `value_type` for `std::array` and several other types usually is). It is the element type with `const` and `volatile` removed.

Construction

You can create a span with the default constructor only with a dynamic extent or when the extent is 0:

```
std::span<int> sp1;      // OK
std::span<int, 0> sp2;   // OK
std::span<int, 5> sp3;   // compile-time ERROR
```

If that is valid, `size()` is 0 and `data()` is `nullptr`.

You can also create and initialize a span with an iterator and a length, or two iterators that define a valid range provided the iterators refer to a contiguous sequence of elements (concept std::contiguous_iterator).

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> sp4a{vec};                    // OK, refers to all elements
std::span<int> sp4b{vec.data(), vec.size()}; // OK, refers to all elements
std::span<int> sp4c{vec.begin(), vec.end()}; // OK, refers to all elements
std::span<int> sp4d{vec.data(), 5};          // OK, refers to first 5 elements
std::span<int> sp4e{vec.begin()+2, 5};       // OK, refers to elements 3 to 7 (including)
std::list<int> lst{...};
std::span<int> sp4f{lst.begin(), lst.end()}; // compile-time ERROR
```

If the span has a fixed extent, it must match the number of elements in the passed range. Otherwise it is undefined behavior (which might or might not work):

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int, 10> sp5a{vec};                // OK, refers to all elements
std::span<int, 5> sp5b{vec};                 // runtime ERROR (undefined behavior)
std::span<int, 20> sp5c{vec};                // runtime ERROR (undefined behavior)
std::span<int, 5> sp5d{vec, 5};              // compile-time ERROR
std::span<int, 5> sp5e{vec.begin(), 5};      // OK, refers to first 5 elements
std::span<int, 3> sp5f{vec.begin(), 5};      // runtime ERROR (undefined behavior)
std::span<int, 8> sp5g{vec.begin(), 5};      // runtime ERROR (undefined behavior)
std::span<int, 5> sp5h{vec.begin()};         // ERROR
```

You can also create and initialize a span directly with a raw array or a `std::array`. In that case, some runtime errors due to an invalid number of elements become a compile-time error:

```
int raw[10];
std::array arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> sp6a{raw};                    // OK, refers to all elements
std::span<int> sp6b{arr};                    // OK, refers to all elements
std::span<int, 5> sp6c{raw};                 // compile-time ERROR
std::span<int, 5> sp6d{arr};                 // compile-time ERROR
std::span<int, 5> sp6e{arr.data(), 5};       // OK
```

That is: Either you pass a container with sequential elements as a whole or you pass two arguments to specify the initial range of elements. In any case, the number of elements must match a specified fixed extent.

Spans have to have the element type of the elements of the sequence they refer to. Conversions (even implicit standard conversion) are not supported. However, additional qualifiers such as `const` are allowed. This also applies to copy constructors:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<const int> sp7a{vec};        // OK: element type with const
std::span<long> sp7b{vec};             // compile-time ERROR: invalid element type
std::span<int> sp7c{sp7a};             // compile-time ERROR: removing constness
std::span<const long> sp7d{sp7a};      // compile-time ERROR: different element type
```

Class template argument deduction is also supported. When initializing the span with a raw array or `std::array<>`, a span with a fixed extent is deduced. Otherwise it has a dynamic extent:

```
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};   // deduces std::vector<int>
std::array arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};    // deduces std::array<int, 10>

std::span sp8a{vec};                   // OK: deduced as std::span<int>
std::span sp8b{arr};                   // OK: deduced as std::span<int, 10>
std::span sp8c{arr.data(), 5};         // OK: deduced as std::span<int>
```

To allow containers to refer to the elements of user-defined containers, these containers have to signal that they or their iterators require that all elements are in contiguous memory. For that they have to fulfill the concept contiguous_iterator.

The constructors also allow the following type conversions between spans:

- Spans with a fixed extent convert to spans with the same fixed extent and additional qualifiers.
- Spans with a fixed extent convert to spans with dynamic extent.
- Spans with a dynamic extent convert to spans with a fixed extent, provided the current extent fits.

8.5 Afternotes

Spans were motivated as `array_views` by Lukasz Mendakiewicz and Herb Sutter in http://wg21.link/n3851 and first proposed by Neil MacIntosh in http://wg21.link/p0122r0. The finally accepted wording was formulated by Neil MacIntosh and Stephan T. Lavavej in http://wg21.link/p0122r7.

Several modifications were added later on, such as removing all comparison operators as proposed by Tony Van Eerd in http://wg21.link/P1085R2 and removing `const_iterator` support with the resolution of http://wg21.link/lwg3320.

This page is intentionally left blank

Chapter 9

Non-Type Template Parameter (NTTP) Extensions

C++20 supports additional types as *non-type template parameter (NTTP)*. Floating-point values, objects of some data structures and simple classes, and even lambdas can be passed as template arguments now. This chapter describes these types and useful applications of this feature.

9.1 New Types for Non-Type Template Parameters

Since C++20 you can use new types for non-type template parameters:

- Floating-point types (such as `double`)
- Structures and simple classes (such as `std::pair<>`), which indirectly also allows us to use string literals as template parameters
- Lambdas

9.1.1 `double` Values as Non-Type Template Parameters

Consider the following example:

lang/nttpdouble.cpp

```
#include <iostream>
#include <cmath>

template<double Vat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + Vat)));
}
```

```
int main()
{
  std::cout << addTax<0.19>(100) << '\n';
  std::cout << addTax<0.19>(4199) << '\n';
  std::cout << addTax<0.07>(1950) << '\n';
}
```

By declaring `addTax()` as

```
template<double Vat>
int addTax(int value)
```

the function template takes a `double` as a template parameter, which is used as value-added tax to add it to an integral value.

The output of the program is as follows:

```
119
4997
2087
```

Passing a floating-point value is now also allowed when the non-type template parameter is declared with `auto`:

```
template<auto Vat>
int addTax(int value)
{
  ...
}

std::cout << addTax<0>(1950) << '\n';      // Vat is int
std::cout << addTax<0.07>(1950) << '\n';   // Vat is double
```

And in the same way, you can use floating-point values now in class templates (declared as `double` or as `auto`):

```
template<double Vat>
class Tax {
  ...
};
```

9.1.2 Objects as Non-Type Template Parameters

Since C++20, you can use an object/value of a data structure or class as a non-type template parameter provided all members are public and the type is a literal type.

Consider the following example:

lang/nttpstruct.cpp

```
#include <iostream>
#include <cmath>
```

```
#include <cassert>

struct Tax {
  double value;

  constexpr Tax(double v)
   : value{v} {
      assert(v >= 0 && v < 1);
  }

  friend std::ostream& operator<< (std::ostream& strm, const Tax& t) {
    return strm << t.value;
  }
};

template<Tax Vat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + Vat.value)));
}

int main()
{
  constexpr Tax tax{0.19};
  std::cout << "tax: " << tax << '\n';

  std::cout << addTax<tax>(100) << '\n';
  std::cout << addTax<tax>(4199) << '\n';
  std::cout << addTax<Tax{0.07}>(1950) << '\n';
}
```

In this case we declare a literal data structure `Tax` with public members, a `constexpr` constructor, and an addition member function:

```
struct Tax {
  double value;

  constexpr Tax(double v) {
    ...
  }
  friend std::ostream& operator<< (std::ostream& strm, const Tax& t) {
    ...
  }
};
```

This allows us to pass objects of this type as template arguments:

```
constexpr Tax tax{0.19};
std::cout << "tax: " << tax << '\n';
std::cout << addTax<tax>(100) << '\n';   // pass Tax object as a template argument
```

This works if the data structure or class is a *structural type*. This roughly means that it only has public members and that it is a literal type that can be used at compile time (being either an aggregate or having a `constexpr` constructor).

Note that you can initialize the data structure with a character array. That way we can now pretty easily pass string literals as template arguments. For example:

lang/nttpstring.cpp

```
#include <iostream>
#include <string_view>

template<auto Prefix>
class Logger {
  ...
 public:
  void log(std::string_view msg) const {
    std::cout << Prefix << msg << '\n';
  }
};

template<std::size_t N>
struct Str {
  char chars[N];
  const char* value() {
    return chars;
  }
  friend std::ostream& operator<< (std::ostream& strm, const Str& s) {
    return strm << s.chars;
  }
};
template<std::size_t N> Str(const char(&)[N]) -> Str<N>;   // deduction guide

int main()
{
  Logger<Str{"> "}> logger;
  logger.log("hello");
}
```

The program has the following output:

```
> hello
```

9.1.3 Lambdas as Non-Type Template Parameters

Because lambdas are just shortcuts for function objects, they can also be used now as non-type template parameters provided the lambda is usable at compile time.

Consider the following example:

lang/nttplambda.cpp

```
#include <iostream>
#include <cmath>

template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}

int main()
{
  auto getDefaultTax = [] {
    return 0.19;
  };

  std::cout << addTax<getDefaultTax>(100) << '\n';
  std::cout << addTax<getDefaultTax>(4199) << '\n';
  std::cout << addTax<getDefaultTax>(1950) << '\n';
}
```

The function template `addTax()` uses a helper function, which now can also be a lambda:

```
template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}
```

To this function template, we can pass a lambda now:

```
auto getDefaultTax = [] {
  return 0.19;
};

addTax<getDefaultTax>(100)          // passes lambda as template argument
```

We could even define the lambda directly when calling the function template:

```
addTax<[]{ return 0.19; }>(100)   // passes lambda as template argument
```

Note that it is a good idea to constrain the template parameter with the concept std::invocable or std::regular_invocable. That way you can ensure that the passed parameter is a callable you can call with no arguments. If the passed lambda should take arguments, you need something like this:

```
template<std::invocable<std::string> auto GetVat>
int addTax(int value, const std::string& name)
{
  double vat = GetVat(name);   // get VAT according to the passed name
  ...
}
```

Note that you cannot skip `auto` in the declaration of the function template. We use `std::invocable` as type constraint of the declaration of an object, which here is callable (function, function object, or lambda) to call:

```
template<std::invocable auto GetVat>   // GetVat is type-constrained function/lambda to call
```

Without `auto` we would declare a function template that has ordinary ***type*** parameter, which we can use to specify or deduce the ***type*** of a callable taking no arguments:

```
template<std::invocable GetVat>        // GetVat is a constrained type
```

It would also work if we declare the function template with the concrete type of the lambda (which, however, means that we have to define the lambda first):

```
auto getDefaultTax = [] {
  return 0.19;
};

template<decltype(getDefaultTax) GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}
```

Note the following constraints for using lambdas as non-type template parameters:

- The lambda may not capture anything.
- The lambda must be usable at compile time.

Fortunately, since C++17 any lambda is implicitly `constexpr` provided it only uses features that are valid for compile-time computing. Alternatively, you can declare the lambda with `constexpr` or `consteval` to get an error on the lambda itself instead of when using it as a template parameter if invalid compile-time features are used.

9.2 Details of Floating-Point Values as NTTP's

******* This chapter/section is at work *******

9.3 Details of Objects as NTTP's

******* This chapter/section is at work *******

9.4 Afternotes

The request for more non-type template parameters exists since the first C++ standard. We discussed it already in the first edition of *C++ Templates – The Complete Guide* (see http://tmplbook.com).

Allowing arbitrary literal types for non-type template parameters was first proposed by Jens Maurer in http://wg21.link/n3413. Allowing class objects as non-type template parameters was then proposed for C++20 by Jeff Snyder in http://wg21.link/p0732r0. Allowing floating-point values as non-type template parameters was then proposed for C++20 by Jorg Brown in http://wg21.link/p1714r0.

The finally accepted wordings were formulated by Jeff Snyder and Louis Dionne in http://wg21.link/p0732r2 and by Jens Maurer in http://wg21.link/p1907r1.

This page is intentionally left blank

Chapter 10

Compile-Time Computing

This chapter presents several extensions of C++ to support compile-time computing.

Part of this chapter are the two new keywords `constinit` and `consteval`, and the extensions that allow programmers to use heap memory, vectors, and strings at compile time.

10.1 Keyword `constinit`

One new keyword C++20 introduces is **`constinit`**. It can be used to force and ensure that a *mutable* static or global variable gets initialized at compile time. So, roughly speaking the effect is described as:[1]

```
constinit = constexpr - const
```

Yes, a `constinit` variable is ***not*** **`const`** (the keyword would better have been named `compiletimeinit`). The name comes from the fact that it initializations happens when usually compile-time constants are initialized.

You can use `constinit` whenever you declare a static or global variable. For example:

```
// outside any function:
constinit auto i = 42;

int getNextClassId() {
  static constinit int maxId = 0;
  return ++maxId;
}

class MyType {
  static constinit long max = sizeof(int) * 1000;
  ...
};
```

[1] Thanks to Jonathan Müller for pointing this out.

```
constexpr std::array<int, 5> getColl() {
  return {1, 2, 3, 4, 5};
}
constinit auto globalColl = getColl();
```

As written, you can still modify the declared values. The following code using the declarations above for the first time:

```
std::cout << i << " " << coll[0] << '\n';  // prints 42 1
i *= 2;
coll = {};
std::cout << i << " " << coll[0] << '\n';  // prints 84 0
```

has the following output:

```
42 1
84 0
```

The effect of using `constinit` is that the initialization only compiles if the initial value is a constant value known at compile-time. That means that is contrast to a declaration such as:

```
auto x = f();              // f() might be runtime function
```

the same declaration with `constinit`

```
constinit auto x = f();    // f() must be a compile-time function
```

requires a compile-time initialization so that `f()` must be callable at compile time (which means it has to be `constexpr` or `consteval`).

If you initialize an object with `constinit`, the constructor has to be usable at compile time

```
constinit std::pair p{42, "ok"};    // OK
constinit std::list l;              // ERROR: default constructor not constexpr
```

The reason to use `constinit` are as follows:

- You can require initialization of mutable global/static objects at compile-time. That way you can especially avoid that the initialization takes time at runtime. This especially can improve performance when using `thread_local` variables.
- You can ensure that a global/static object is always initialized when being used. In fact, `constinit` can be used to fix the *static initialization order fiasco*, which can occur when an initial value of a static/global object depends on another static/global object.

Note that using `constinit` never changes the functional behavior of a program (unless we have the static initialization order fiasco). It can only lead to the consequence that code does not longer compile.

10.1.1 Using `constinit` in Practice

There are couple of thing to respect when using `constinit`.[2]

First, you cannot initialize a `constinit` value with another `constinit` value:

[2] Thanks to `cppreference.com` for pointing out some of the issues mentioned here.

```
constinit auto x = f();    // f() must be a compile-time function
constinit auto y = x;      // ERROR: x is not constant initializer
```

The reason is that the initial value must be a constant value known at compile time, but `constinit` values are not constant. Only this compiles:

```
constexpr auto x = f();    // f() must be a compile-time function
constinit auto y = x;      // OK
```

When initializing objects, a compile-time constructor is required. However, a compile-time ***destructor*** is not required. For this reason, you can use `constinit` for smart pointers:

```
constinit std::unique_ptr<int> up;  // OK
constinit std::shared_ptr<int> sp;  // OK
```

`constinit` does not imply `inline` (this is different from `constexpr`). For example:

```
class Type {
  constinit static int val1 = 42;          // ERROR
  inline static constinit int val2 = 42;   // OK
  ...
};
```

You can use `constinit` together with `extern`:

```
// header:
extern constinit int max;

// translation unit:
constinit int max = 42;
```

For the same effect, you can also skip `constinit` in the declaration. However, skipping `constinit` in the definition would no longer force compile-time initialization.

You can use `constinit` together with `static` and `thread_local`:

```
static thread_local constinit int numCalls = 0;
```

Any order of `constinit`, `static`, and `thread_local` is fine.

Note that for `thread_local` variables, using `constinit` might create a performance improvement in that internally no guard has to be generated to signal whether the variable is already initialized inside the thread:

```
extern thread_local int x1 = 0;
extern thread_local constinit int x2 = 0;   // better (might avoid an internal guard)
```

Using `constinit` to declare references is possible but makes no sense, because the reference refers to a constant object. You should use `constexpr` instead.

10.1.2 How `constinit` Solves the Static Initialization Order Fiasco

In C++ there is a problem called *static initialization order fiasco*, which `constinit` can solve. The problem is that the order of static and global initializations in different translation units is not defined. For that reason, the following code can be a problem:

- Assume we have a type with a constructor to initialize the objects and introduce an extern global object of this type:

comptime/truth.hpp

```
#ifndef TRUTH_HPP
#define TRUTH_HPP

struct Truth {
  int value;
  Truth() : value{42} {   // ensure all objects are initialized with 42
  }
};

extern Truth theTruth;    // declare global object

#endif // TRUTH_HPP
```

- And then we initialize the object in its own translation unit:

comptime/truth.cpp

```
#include "truth.hpp"

Truth theTruth;           // define global object (should have value 42)
```

- And then n another translation unit we initialize another global/static object with `theTruth`:

comptime/fiasco.cpp

```
#include "truth.hpp"
#include <iostream>

int val = theTruth.value;      // may be initialized before theTruth is initialized

int main()
{
  std::cout << val << '\n';    // OOPS: may be 0 or 42
  ++val;
  std::cout << val << '\n';    // OOPS: may be 1 or 43
}
```

There is a good chance that `val` is initialized with `theTruth` ***before*** it was initialized. And that means, the program might have the following output:[3]

```
0
```

[3] For example, this happens with the gcc compiler if the linker arguments pass `truth.o` before `fiasco.o`.

```
1
```

When using `constinit` to declare `val`, that problem cannot occur. `constinit` ensures that an object is *always* initialized before it is used, because the initialization happens at compile time. If the guarantee cannot be given, the code does not compile. Note that an initialization would also be guaranteed if `val` was declared with `constexpr`; however, then you could not modify the value anymore.

In our example, just using `constinit` would first result in a compile-time error (signaling that initialization cannot be guaranteed at compile time):

```
// truth.hpp:
struct Truth {
  int value;
  Truth() : value{42} {
  }
};
extern Truth theTruth;

// main translation unit:
constinit int val = theTruth.value ;   // ERROR: no constant initializer
```

You have to modify the declaration of `theTruth` and class `Truth` so that it can be used at compile time:

comptime/truthc.hpp

```
#ifndef TRUTH_HPP
#define TRUTH_HPP

struct Truth {
  int value;
  constexpr Truth() : value{42} {   // enable compile-time initialization
  }
};

constexpr Truth theTruth;           // force compile-time initialization

#endif // TRUTH_HPP
```

Now, the program compiles and `val` is guaranteed to be initialized with the initialized value of `theTruth`:

comptime/constinit.cpp

```
#include "truthc.hpp"
#include <iostream>

constinit int val = theTruth.value ;   // initialized after theTruth is initialized

int main()
{
```

```
  std::cout << val << '\n';    // guaranteed to be 42
  ++val;
  std::cout << val << '\n';    // guaranteed to be 43
}
```

Therefore, the output of the program is now guaranteed to be:

```
42
43
```

There are other ways to solve the static initialization order fiasco (using static function to get the value, using `inline`). Nevertheless, you might carefully think about following a programming style that global and static variables are always declared with `constinit` provided the initialization does not need any runtime value/feature. Using it in a function like this does at least not hurt:

```
long nextId()
{
  constinit static long id = 0;
  return ++id;
}
```

10.2 Keyword `consteval`

Since C++11, C++ has the keyword `constexpr` to support evaluating functions at compile time. Provided all aspects of the functions are known at compile time, you can also use the results in compile-time contexts. However, `constexpr` functions also serve as "normal" runtime functions.

C++20 introduces a similar keyword **`consteval`**, which mandates compile-time computing. In contrast to functions marked with `constexpr`, functions marked with `consteval` cannot be called at runtime; instead, they are ***required*** to be called at compile time. If this is not possible, the program is ill-formed. Because these functions are called immediately, when the compiler sees calls of them these functions are also called ***immediate functions***.

10.2.1 A First `consteval` Example

Consider the following example:

lang/consteval.cpp

```
#include <iostream>
#include <array>

constexpr
bool isPrime(int value)
{
  for (int i = 2; i < value/2; ++i) {
    if (value % i == 0) {
```

```
      return false;
    }
  }
  return value > 1;  // 0 and 1 are no prime numbers
}

template<int Num>
consteval
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  int idx = 0;
  for (int val = 1; idx < Num; ++val) {
    if (isPrime(val)) {
      primes[idx++] = val;
    }
  }
  return primes;
}

int main()
{
  // init with prime numbers:
  auto primes = primeNumbers<100>();

  for (auto v : primes) {
    std::cout << v << '\n';
  }
}
```

Here, we use `consteval` to define function `primeNumbers<N>()` to yield an array of the first *N* prime numbers at compile time:

```
template<int Num>
consteval
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  ...
  return primes;
}
```

For this it uses a helper function `isPrime()` that is declared with `constexpr` to be usable at both runtime and compile time (we could also declare it with `consteval`, but then it would not be usable at runtime).

The program then uses `primeNumbers<>()` to initialize an array of 100 prime numbers:

```
auto primes = primeNumbers<100>();
```

Because `primeNumbers<>()` is `consteval`, it is required that this initialization happens at compile time. You would have the same effect if `primeNumbers<>()` is declared with `constexpr` and the function is called in a compile-time context (such as to initialize a `constexpr` or `constinit` array `prime`):

```
template<int Num>
constexpr
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  ...
  return primes;
}
...

constinit static auto primes = primeNumbers<100>();
```

In both cases you can see that the compile time becomes significantly slower when the number of prime numbers to compute for the initializations grows significantly.

However, note that compilers usually have a limit when evaluating constant expressions. The C++ standard only guarantees the evaluation of 1,048,576 expressions within a core constant expressions.

`consteval` Lambdas

You can also declare lambdas to be `consteval` now. That way it is required to evaluate the lambda at compile-time

Consider the following example:

```
int main(int argc, char* argv[])
{
  // compile-time function to compute hash value for string literals:
  auto hashed = [] (const char* str) consteval {
                  std::size_t hash = 5381;  // see http://www.cse.yorku.ca/~oz/
hash.html
                  while (*str != '\0') {
                    hash = hash * 33 ^ *str++;
                  }
                  return hash;
                };

  // OK (requires hashed() in compile-time context):
  enum class drinkHashes : long { beer = hashed("beer"), wine = hashed("wine"),
                                  water = hashed("water"), ... };

  // OK (hashed() guaranteed to be called at compile-time):
  std::array arr{hashed("beer"), hashed("wine"), hashed("water")};

  if (argc > 1) {
```

```
    switch (hashed(argv[1])) {   // ERROR: argv is not known at compile time
      ...
    }
  }
}
```

Here we initialize `hashed` with a lambda that you can only use at compile time. Therefore, the use of the lambda as a function has to happen in a compile-time context with compile-time values. For this reason, only the calls that take a string literal, which is know at compile time, are valid.

If you declare the lambda with `constexpr` (you can also skip `constexpr`, because since C++17 all lambdas are implicitly `constexpr` if possible), the `switch` statement would become valid. However, it is no longer guarantee that the computation of the initial values for `arr` happens at compile time.

See the discussion of `consteval` lambdas in the chapter of all new lambda features for more details and another example.

10.2.2 `constexpr` versus `consteval`

With `constexpr` and `consteval` we have now the following options to influence when a function can be and gets called:

- Neither `constexpr` nor `consteval`:
  These functions can only be used in runtime contexts. The compiler can still perform optimizations that evaluate them at compile time, though.
- `constexpr`:
  These functions can be used in compile-time and runtime contexts. Even in runtime contexts, the compiler can still perform optimizations that evaluate them at compile time. Also in compile-time contexts the compiler is still allowed to evaluate them at runtime.
- `consteval`:
  These functions can be used at compile-time only. However, the result can be used in a runtime context.

For example, consider the following 3 functions declared in a header file (therefore, `squareR()` is declared with `inline`):

comptime/consteval.hpp

```
// square() for run time:
inline int squareR(int x) {
  return x * x;
}

// square() for compile and run time:
constexpr int squareCR(int x) {
  return x * x;
}

// square() for compile time only:
consteval int squareC(int x) {
  return x * x;
```

```
}
```

We can use these functions as follows:

comptime/consteval.cpp

```
#include "consteval.hpp"
#include <iostream>
#include <array>

int main()
{
  int i = 42;

  // using the square functions at runtime with runtime value:
  std::cout << squareR(i) << '\n';       // OK
  std::cout << squareCR(i) << '\n';      // OK
  //std::cout << squareC(i) << '';       // ERROR

  // using the square functions at runtime with compile-time value:
  std::cout << squareR(42) << '\n';      // OK
  std::cout << squareCR(42) << '\n';     // OK
  std::cout << squareC(42) << '\n';      // OK: square computed at compile time

  // using the square functions at compile-time:
  //std::array<int, squareR(42)> arr1;   // ERROR
  std::array<int, squareCR(42)> arr2;    // OK: square computed at compile time
  std::array<int, squareC(42)> arr3;     // OK: square computed at compile time
  //std::array<int, squareC(i)> arr4;    // ERROR
}
```

The difference between `constexpr` and `consteval` is that.

- The `consteval` function is not allowed to process parameters that are not known at compile time:

```
std::cout << squareCR(i) << '\n';      // OK
std::cout << squareC(i) << '\n';       // ERROR
std::array<int, squareC(i)> arr4;      // ERROR
```

- The `consteval` function is required to perform its computing at compile time:

```
std::cout << squareCR(42) << '\n';     // may be computed at compile time or runtime
std::cout << squareC(42) << '\n';      // computed at compile time
```

This means that `consteval` makes sense to be used in two situations:

- You want to enforce compile-time computation.
- You want to disable a function to be used at runtime.

For example, whether a function square() or hashed() is constexpr or consteval makes no difference in compile-time contexts, such as the following:

```
enum class Drink = { water = hashed("water"), wine = hashed("wine") };

switch (value) {
  case square(42):
    ...
}

if constexpr(hashed("wine") > hashed("water")) {
  ...
}
```

However, in runtime contexts, consteval will make a difference because compile-time computing is not required then:

```
std::array drinks = { hashed("water"), hashed("wine") };

std::cout << hashed("water");

if (hashed("wine") > hashed("water")) {
  ...
}
```

10.2.3 Using consteval in Practice

For consteval function a couple of restrictions apply when using them in practice.

Constraints for consteval

consteval functions share most of the other aspects of constexpr functions (note that these constraints were relaxed for constexpr functions with C++20):

- Parameters and return type (if any) have to be literal types.
- The body may only contain variables of literal types that are neither static nor thread_local
- Using goto and labels is not allowed.
- The function may only be a constructor or destructor, the class has no virtual base class.
- They are implicitly inline.
- They cannot be used as coroutines.

Call Chains with consteval Functions

Functions marked with consteval can call other functions marked with constexpr or consteval:

```
constexpr int funcConstExpr(int i) {
  return i;
}
```

```
consteval int funcConstEval(int i) {
  return i;
}

consteval int foo(int i) {
  return funcConstExpr(i) + funcConstEval(i);  // OK
}
```

However, `constexpr` functions cannot call `consteval` functions for variables:

```
consteval int funcConstEval(int i) {
  return i;
}

constexpr int foo(int i) {
  return funcConstEval(i);  // ERROR
}
```

The function `foo()` does not compile. The reason is that `i` is still not a compile-time value because it *could* be called at runtime. `foo()` could only call `funcConstEval()` with a compile-time variable:

```
constexpr int foo(int i) {
  return funcConstEval(42);  // OK
}
```

Note that even `if(std::is_constant_evaluated())` does not help here.

Functions marked with `consteval` can also not call pure runtime function (functions neither marked with `constexpr` nor with `consteval`). However, this is only checked if the call is really performed. For a `consteval` function it is not an error to have statements that call runtime functions if they are not reached (we already had this rule for `constexpr` functions called at compile time).

Consider the following example:

```
void compileTimeError()
{
}

consteval int nextTwoDigitValue(int val)
{
  if (val < 0 || val >= 99) {
    compileTimeError();  // call something not valid to call at compile time
  }
  return ++val;
}
```

This compile-time function has the interesting effect that it can only be used for ***some*** arguments:

```
constexpr int i1 = nextTwoDigitValue(0);   // OK (initializes i1 with 1)
constexpr int i2 = nextTwoDigitValue(77);  // OK (initializes i2 with 78)
constexpr int i3 = nextTwoDigitValue(99);  // compile-time ERROR
```

```
constexpr int i4 = nextTwoDigitValue(-1);   // compile-time ERROR
```

Note that using `static_assert()` would not work here, because it can only be called for values known at compile time and `consteval` does not make `val` a compile-time value inside the function:

```
consteval int nextTwoDigitValue(int val)
{
  static_assert(val >= 0 && val < 99);      // always ERROR: val is no compile-time value
  ...
}
```

By using this trick, you can constrain compile-time functions to certain values. That way, you could signal an invalid format of a string parsed at compile time.

10.2.4 Compile-Time Value versus Compile-Time Context

You might assume that each and every value in a compile-time function is a compile-time value. However, that is not true. Also in compile-time functions, there is a difference between static typing of the code and dynamic computing of values.

Consider the following example:

```
consteval void process()
{
  constexpr std::array a1{0, 8, 15};
  constexpr auto n1 = std::ranges::count(a1, 0);  // OK
  std::array<int, n1> a1b;                        // OK

  std::array a2{0, 8, 15};
  constexpr auto n2 = std::ranges::count(a2, 0);  // ERROR

  std::array a3{0, 8, 15};
  auto n3 = std::ranges::count(a3, 0);            // OK
  std::array<int, n3> a3b;                        // ERROR
}
```

Although we are in a function that can only be used at compile time, `a2` is not a compile-time value. Therefore, any processing with it cannot be used as compile-time value such as initializing the `constexpr` variable `n2`.

For the same reason, only `n1` can be used to declare the size of a `std::array`. Using `n3`, which is not `constexpr`, fails (even if `n3` would be declared as `const`).

The same applies if `process()` is declared as `constexpr` function. Whether it is called in a compile-time context does not matter.

10.3 Relaxed Constraints for `constexpr` Functions

Every C++ version since C++11 C++20 relaxed the constraints for `constexpr` functions. This also means that the `consteval` functions only have these relaxed constraints.

Basically, the constraints for `constexpr` and `consteval` functions are now as follows:

- Parameters and return type (if any) have to be literal types.
- The body may only contain variables of literal types that are neither `static` nor `thread_local`
- Using `goto` and labels is not allowed.
- The function may only be a constructor or destructor, the class has no virtual base class.
- They are implicitly `inline`.
- They cannot be used as coroutines.

******* This chapter/section is at work *******

10.4 `std::is_constant_evaluated()`

C++20 provides a new helper function to switch code between compile-time and runtime computing: `std::is_constant_evaluated()`. It is defined in header `<<type_traits>` (although it is not really a type function).

It allows code to switch from calling a helper function only callable at runtime to code that can be used at compile-time. For example:

comptime/isconsteval.hpp

```
#include <type_traits>
#include <cstring>

constexpr int len(const char* s)
{
  if (std::is_constant_evaluated()) {
    int idx = 0;
    while (s[idx] != '\0') {       // compile-time friendly code
      ++idx;
    }
    return idx;
  }
  else {
    return std::strlen(s);         // function called at runtime
  }
}
```

In function `len()` we compute the length of a raw string or string literal. If called at runtime, we use the standard C function `strlen()`. However, to enable that the function can be used at compile-time, we provide a different implementation if we are in a compile-time context.

Here is how the two branches get called:

```
constexpr int l1 = len("hello");   // uses then branch
int l2 = len("hello");             // uses else branch (no required compile-time context)
```

The first call of `len()` happens in a compile-time context. For that reason, `is_constant_evaluated()` yields `true` so that we use the *then* branch. The second call of `len()` happens in a runtime context, so that `is_constant_evaluated()` yields `false` and `strlen()` is called. The latter happens even if the compiler decides to evaluate the call at compile-time. The important point is that the call of `len()` and therefore the call of `std::is_constant_evaluated()` happens in a runtime context.

Here is a function that does the opposite: converting and integral value to a string both at compile time or at runtime:

comptime/asstring.hpp

```
#include <string>
#include <format>

// convert integral value to std::string
// - can be used at compile time or runtime
constexpr std::string asString(long long value)
{
  if (std::is_constant_evaluated()) {
    // compile-time version:
    if (value == 0) {
      return "0";
    }
    if (value < 0) {
      return "-" + asString(-value);
    }
    std::string s = asString(value / 10) + std::string(1, value % 10 + '0');
    if (s.size() > 1 && s[0] == '0') {  // skip leading 0 if any
      s.erase(0, 1);
    }
    return s;
  }
  else {
    // runtime version:
    return std::format("{}", value);
  }
}
```

At runtime we simply use `std::format()`. At compile-time we manually create a string of the optional negative sign and all digits (we use a recursive approach to bring them in the right order). An example of its use you can find in the section about exporting compile-time string to runtime strings.

10.4.1 `std::is_constant_evaluated()` in Detail

According to the C++20 standard, `std::is_constant_evaluated()` yields `true` when it called in a *manifestly constant-evaluated* expression or conversion. That is roughly the case if we call it:

- in a constant-expression, or
- in a constant context (in `if constexpr`, a `consteval` function, or an constant initialization), or
- for an initializer of a variable usable at compile time

For example:

```
constexpr bool isConstEval() {
  return std::is_constant_evaluated();
}

bool g1 = isConstEval();                        // true
const bool g2 = isConstEval();                  // true
static bool g3 = isConstEval();                 // true
static int g4 = g1 + isConstEval();             // false

int main()
{
  bool l1 = isConstEval();                      // false
  const bool l2 = isConstEval();                // true
  static bool l3 = isConstEval();               // true
  int l4 = g1 + isConstEval();                  // false
  const int l5 = g1 + isConstEval();            // false
  static int l6 = g1 + isConstEval();           // false
  int l7 = isConstEval() + isConstEval();       // false
  const auto l8 = isConstEval() + 42 + isConstEval();  // true
}
```

We would get the same effect if `isConstEval()` is called indirectly via a `constexpr` function.

`std::is_constant_evaluated()` with `constexpr` and consteval

For example, assume we have the following functions defined:

```
bool runtimeFunc() {
  return std::is_constant_evaluated(); // always false
}
constexpr bool constexprFunc() {
  return std::is_constant_evaluated(); // may be false or true
}
consteval bool constevalFunc() {
  return std::is_constant_evaluated(); // always true
}
```

Then we have the following behavior:

```
void foo()
{
  bool b1 = runtimeFunc();           // false
  bool b2 = constexprFunc();         // false
  bool b3 = constevalFunc();         // true

  static bool sb1 = runtimeFunc();    // false
  static bool sb2 = constexprFunc();  // true
  static bool ab3 = constevalFunc();  // true
  const bool cb1 = runtimeFunc();     // ERROR
  const bool cb2 = constexprFunc();   // true
  const bool cb3 = constevalFunc();   // true

  int y = 42;
  static bool sb4 = y + runtimeFunc();    // function yields false
  static bool sb5 = y + constexprFunc();  // function yields false
  static bool ab6 = y + constevalFunc();  // function yields true
  const bool cb4 = y + runtimeFunc();     // function yields false
  const bool cb5 = y + constexprFunc();   // function yields false
  const bool cb6 = y + constevalFunc();   // function yields true
}
```

With `constexpr` or `static constinit` an initialization with `y` we would get the same behavior as for `cb1`, `cb2`, and `cb3`. However, with `y` involved we always get an error because a runtime value is involved.

In general, it makes no sense to use `std::is_constant_evaluated()`

- as condition in a compile-time `if`, because that always yields `true`:

  ```
  if constexpr (std::is_constant_evaluated()) {   // always true
    ...
  }
  ```

  Inside `if constexpr` we have a compile-time context so that the answer to the question whether we are in a compile-time context is always `true` (independent from the context the whole function was called from).

- Inside a pure runtime function, because that usually yields `false`. The only exception is if it is used in a local constant evaluation:

  ```
  void foo() {
    if (std::is_constant_evaluated()) {           // always false
      ...
    }
    const bool b = std::is_constant_evaluated();  // true
  }
  ```

- inside a `consteval` function, because that always yields `true`:

  ```
  consteval void foo() {
    if (std::is_constant_evaluated()) {          // always true
      ...
  ```

```
    }
  }
```

Therefore, using `std::is_constant_evaluated()` usually only makes sense in `constexpr` functions. However, cannot use `std::is_constant_evaluated()` inside a `constexpr` function to call a `consteval` function, because calling a `consteval` function from a `constexpr` function is not allowed in general:[4]

```
consteval int funcConstEval(int i) {
  return i;
}

constexpr int foo(int i) {
  if (std::is_constant_evaluated()) {
    return funcConstEval(i);  // ERROR
  }
  else {
    return funcRuntime(i);
  }
}
```

`std::is_constant_evaluated()` and Operator `?:`

In the C++20 standard is an interesting example to clarify the way `std::is_constant_evaluated()` can be used. Modified it is as follows:

```
int sz = 10;
constexpr bool sz1 = std::is_constant_evaluated() ? 20 : sz;  // true, so 20
constexpr bool sz2 = std::is_constant_evaluated() ? sz : 20;  // false, so ERROR
```

The reason for this behavior is as follows:[5]

- The initialization of `sz1` and `sz2` is either static initialization or dynamic initialization.
- For static initialization, the initializer must be constant. So, the compiler attempts to evaluate the initializer with `std::is_constant_evaluated()` treated as a constant of value `true`.
  - With `sz1`, that succeeds. The result is `1`, and that is a constant. So, `sz1` is constant initialized with `20`.
  - With `sz2`, the result is `sz`, which is not a constant. So, `sz2` is (notionally) dynamically initialized. Therefore, the previous result is discarded and the initializer is evaluated with `std::is_constant_evaluated()` producing `false` instead. So, the expression to initialize `sz2` is also `20`.

    However, `sz2` is not necessarily a constant, because `std::is_constant_evaluated()` is not necessarily a constant-expression during this evaluation. So, the initialization of `sz2` with this `20` does not compile.

By using `const` instead of `constexpr` it becomes even more tricky:

```
int sz = 10;
const bool sz1 = std::is_constant_evaluated() ? 20 : sz;  // true, so 20
```

[4] C++23 will probably enable code like this with `if consteval`.

[5] Thanks to David Vandevoorde for pointing this out.

```
const bool sz2 = std::is_constant_evaluated() ? sz : 20;  // false, so also 20

double arr1[sz1];  // OK
double arr2[sz2];  // may or may not compile
```

Only `sz1` is a compile-time constant an can always be used to initialize and array. For the reason described above, `sz2` is also initialized to `20`. However, because it is not necessarily a constant, the initialization of `arr2` may or may not compile (according to the compiler and optimizations used).

10.5 Using Heap Memory, Vectors, and Strings at Compile Time

Since C++20, compile-time functions can allocate memory provided the memory is also released at compile time. For this reason, you can now use strings or vectors at compile time. However, you cannot use the compile-time created strings or vectors at runtime because memory allocated at compile time has to be released at compile time.

10.5.1 Using Vectors at Compile Time

Here is a first example, using a `std::vector<>` at compile time:

comptime/vector.hpp

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <numeric>

template<std::ranges::input_range T>
constexpr auto modifiedAvg(const T& rg)
{
  using elemType = std::ranges::range_value_t<T>;

  // initialize compile-time vector with passed elements:
  std::vector<elemType> v{std::ranges::begin(rg),
                          std::ranges::end(rg)};
  // perform several modifications:
  v.push_back(elemType{});
  std::ranges::sort(v);
  auto newEnd = std::unique(v.begin(), v.end());
  ...

  // return average of modified vector:
  auto sum = std::accumulate(v.begin(), newEnd,
                             elemType{});
  return sum / static_cast<double>(v.size());
}
```

Here, we define `modifiedAvg()` with `constexpr` so that it could be called at compile time. Inside, we use the `std::vector<>` `v`, initialized with the elements of the passed range. This allows us to use the full API of a vector (especially inserting and removing elements). As an example, we insert an element, sort the elements, and remove consecutive duplicates with `unique()`. All these algorithms are `constexpr` now so that we can use them at compile time.

However, at the end we do ***not*** return the vector. We only return a value computed with the help of a compile-time vector.

We can call this function at compile time:

comptime/vector.cpp

```
#include "vector.hpp"
#include <iostream>
#include <array>

int main()
{
  constexpr std::array orig{0, 8, 15, 132, 4, 77};

  constexpr auto avg = modifiedAvg(orig);
  std::cout << "average: " << avg << '\n';
}
```

Due to the fact that `avg` is declared with `constexpr`, `modifiedAvg()` is evaluated at compile time.

We could also declare `modifiedAvg()` with `consteval`, in which case we could pass the argument by value because we do not copy elements at runtime:

```
template<std::ranges::input_range T>
consteval auto modifiedAvg(T rg)
{
  using elemType = std::ranges::range_value_t<T>;

  // initialize compile-time vector with passed elements:
  std::vector<elemType> v{std::ranges::begin(rg),
                          std::ranges::end(rg)};
  ...
}
```

However, note that we still cannot declare and initialize a vector at compile time that is usable at runtime:

```
int main()
{
  constexpr std::vector orig{0, 8, 15, 132, 4, 77};  // ERROR
  ...
}
```

For the same reason, a compile-time function can only return a vector to the caller when the return value is used at compile time:

comptime/returnvector.cpp

```
#include <vector>

constexpr auto returnVector()
{
  std::vector<int> v{0, 8, 15};
  v.push_back(42);
  ...

  return v;
}

constexpr auto returnVectorSize()
{
  auto coll = returnVector();
  return coll.size();
}

int main()
{
  //constexpr auto coll = returnVector();  // ERROR
  constexpr auto tmp = returnVectorSize();  // OK
  ...
}
```

10.5.2 Returning a Collection at Compile Time

Although you cannot return a compile-time vector so that it can be used at runtime, there is a way to return a collection of elements computed at compile time: you can return a `std::array<>`. The only problem is to know the size of the array because it cannot be initialized by the size of an array:

```
std::vector v;
...
std::array<int, v.size()> arr;  // ERROR
```

Code like this ***never*** compiles because `size()` is a runtime value and the declaration of `arr` needs a compile-time value. It does not matter whether this code is evaluated at compile time.

For that reason, you have to return a fixed-sized array. For example:

comptime/mergevalues.hpp

```
#include <vector>
#include <ranges>
```

```
#include <algorithm>
#include <array>

template<std::ranges::input_range T>
consteval auto mergeValues(T rg, auto... vals)
{
  // create compile-time vector:
  std::vector<std::ranges::range_value_t<T>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));  // and merge passed values

  std::ranges::sort(v);       // sort all elements

  // return extended collection as array:
  constexpr auto sz = std::ranges::ssize(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<T>, sz> arr{};
  std::ranges::copy(v, arr.begin());
  return arr;
}
```

We use a vector in a `consteval` function to merge a variadic number of passed arguments with the elements of the passed range and sort them all. However, we return the resulting collection as `std::array` so that it can be passed to a runtime context. For example, the following program works find with this function:

comptime/mergevalues.cpp

```
#include "mergevalues.hpp"
#include <iostream>
#include <array>

int main()
{
  // compile-time initialization of array:
  constexpr std::array orig{0, 8, 15, 132, 4, 77, 3};

  // initialization of sorted extended array:
  auto merged = mergeValues(orig, 42, 4);

  // print elements:
  for(const auto& i : merged) {
    std::cout << i << ' ';
  }
}
```

It has the following output:

```
0 3 4 4 8 15 42 77 132
```

If we do not know the resulting size of the array at compile-time, we have to declare the returned array with a maximum size and return the resulting size in addition. The function merging the values now might look as follows:

comptime/mergevaluessz.hpp

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <array>

template<std::ranges::input_range T>
consteval auto mergeValuesSz(T rg, auto... vals)
{
  // create compile-time vector:
  std::vector<std::ranges::range_value_t<T>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));  // and merge passed values

  std::ranges::sort(v);       // sort all elements

  // return extended collection as array and its size:
  constexpr auto maxSz = rg.size() + sizeof...(vals);
  std::array<std::ranges::range_value_t<T>, maxSz> arr{};
  auto res = std::ranges::unique_copy(v, arr.begin());
  return std::pair{arr, res.out - arr.begin()};
}
```

In addition, we use std::ranges::unique_copy() to remove consecutive duplicates after sorting and return both the array and the resulting number of elements.

Note that you should declare `arr` with `{}` to ensure that all values in the array are initialized. It is not allowed to let a compile-time function yield uninitialized memory.

We can use the returned value now as follows:

comptime/mergevaluessz.cpp

```
#include "mergevaluessz.hpp"
#include <iostream>
#include <array>
#include <ranges>

int main()
{
  // compile-time initialization of array:
  constexpr std::array orig{0, 8, 15, 132, 4, 77, 3};
```

```
  // initialization of sorted extended array:
  auto tmp = mergeValuesSz(orig, 42, 4);
  auto merged = std::views::counted(tmp.first.begin(), tmp.second);

  // print elements:
  for(const auto& i : merged) {
    std::cout << i << ' ';
  }
}
```

By using the view adaptor `std::views::counted()`, we can easily combine both the returned array and the returned size of elements to use in the array as single range.

The output of the program is now:

```
0 3 4 8 15 42 77 132
```

10.5.3 Using Strings at Compile Time

For compile-time strings, all operations are `constexpr` now. Therefore, you can use `std::string` but also any other string types such as `std::u8string` at compile time now.

However, there is again the restriction that you cannot use a compile-time string at runtime. For example:

```
consteval std::string returnString()
{
  std::string s = "Some string from compile time";
  ...
  return s;
}

void useString()
{
  constexpr auto s = returnString();  // ERROR
  ...
}

constexpr void useStringInConstexpr()
{
  std::string s = returnString();     // ERROR
  ...
}

constexpr void useStringInConsteval()
{
```

```
    std::string s = returnString();      // OK
    ...
}
```

You can also not solve the problem by returning just the compile-time string with `data()` or `c_str()` or as a `std::string_view`. You would return the address of memory allocated at compile time Fortunately, compiler raise a compile-time error if that happens.

However, we can use the same trick as described for vectors above. We can convert the string into an array of fixed size and return both the array and the size of the vector.

Here is a complete example:

comptime/comptimestring.cpp

```
#include <iostream>
#include <string>
#include <array>
#include <cassert>
#include "asstring.hpp"

// function template to export a compile-time string to runtime:
template<int MaxSize>
consteval auto toRuntimeString(std::string s)
{
  // ensure the size of the exported array is large enough:
  assert(s.size() <= MaxSize);

  // create a compile-time array and copy all characters into it:
  std::array<char, MaxSize+1> arr{};  // ensure all elems are initialized
  for (int i = 0; i < s.size(); ++i) {
    arr[i] = s[i];
  }

  // return the compile-time array and the string size:
  return std::pair{arr, s.size()};
}

// function to import an exported compile-time string at runtime:
std::string fromComptimeString(const auto& dataAndSize)
{
  // init string with exported array of chars and size:
  return std::string{dataAndSize.first.data(),
                     dataAndSize.second};
}

// test the functions:
consteval auto comptimeMaxStr()
```

```
{
  std::string s = "max int is " + asString(std::numeric_limits<int>::max())
                  + " (" + asString(std::numeric_limits<int>::digits + 1) + " bits)";

  return toRuntimeString<100>(s);
}

int main()
{
  std::string s = fromComptimeString(comptimeMaxStr());
  std::cout << s << '\n';
}
```

Again, we define two helper functions to export a compile-time string to a runtime string:

- The compile-time function `toRuntimeString()` converts to string to a `std::array<>` and returns the array and the size of the string:

  ```
  template<int MaxSize>
  consteval auto toRuntimeString(std::string s)
  {
    assert(s.size() <= MaxSize);          // ensure array size fits

    // create a compile-time array and copy all characters into it:
    std::array<char, MaxSize+1> arr{};   // ensure all elems are initialized
    for (int i = 0; i < s.size(); ++i) {
      arr[i] = s[i];
    }

    return std::pair{arr, s.size()};      // return array and size
  }
  ```

  By using `assert()` we double check that the size of the array is large enough. The same way we could double ensure at compile time that we do not waste too much memory.

- The runtime function `fromComptimeString()` then takes the returned array and size to initialize a runtime string and returns it:

  ```
  std::string fromComptimeString(const auto& dataAndSize)
  {
    return std::string{dataAndSize.first.data(),
                       dataAndSize.second};
  }
  ```

The test case of the function uses the helper function `asString()`, which can be used at compile time and runtime to convert an integral value to a string.

The program has, for example, the following output:

```
max int is 2147483647 (32 bits)
```

10.6 Other constexpr Extensions

Besides the ability to use heap memory at compile time, compile-time functions (whether declared with constexpr or consteval), can use a couple of additional language features since C++20. As a consequence and in addition, a couple of library features can now also be used at compile time.

10.6.1 constexpr Language Extensions

Since C++20, the following language features are possible to be used in compile.time functions (whether declared with constexpr or consteval):

- You can now use heap memory at compile time.
- Runtime polymorphism is supported:
  - You can now use virtual functions.
  - You can now use dynamic_cast.
  - You can now use typeid.
- You can have try-catch blocks now (but you are still not allowed to throw).
- You can now change the active member of a union.

Note that you are still not allowed to use static in constexpr or consteval functions.

10.6.2 constexpr Library Extensions

constexpr Algorithms and Utilities

Most algorithms in <algorithm>, numeric>, and <utility> are constexpr now. That means that you can sort and accumulate elements now at compile time (see the example using vectors at compile time).

However, note that parallel algorithms (algorithms that have an execution policy parameter) can still only be used at runtime.

constexpr Library Types

Several types of the C++ standard library now have better support for constexpr so that objects can be used (better) at compile time:

- Vectors and strings can be used at compile time now.
- A couple of std::complex<> operations became constexpr.
- A couple of missing constexpr in std::optional<> and std::variant<> were added.
- constexpr was deed to std::invoke(), std::ref(), std::cref(), std::bind(), std::bind_front(), mem_fn(), not_fn().
- In pointer_traits, pointer_to() for raw pointers can now be used at compile time.

10.7 Afternotes

******* This chapter/section is at work *******

The keyword `constinit` was first proposed as an attribute by Eric Fiselier in http://wg21.link/p1143r0. The finally accepted wording was formulated by Eric Fiselier in http://wg21.link/p1143r2.

The keyword `consteval` was first proposed by Richard Smith, Andrew Sutton, Daveed Vandevoorde in http://wg21.link/p1073r0. The finally accepted wording was formulated by Richard Smith, Andrew Sutton, Daveed Vandevoorde in http://wg21.link/p1073r3. A small fix was later added by David Stone in http://wg21.link/p1937r2.

`std::is_constant_evaluated()` was first proposed as `constexpr` operator by Daveed Vandevoorde in http://wg21.link/p0595r0. The finally accepted wording was formulated by Richard Smith, Andrew Sutton, Daveed Vandevoorde in http://wg21.link/p0595r2.

Chapter 11

Lambda Extensions

This chapter presents the supplementary features C++20 introduces for lambdas.

11.1 Generic Lambdas with Template Parameters

C++20 introduces an extension to be able to use template parameters for generic lambdas. You can specify these template parameters between the capture clause and the call parameters (if any):

```
auto foo = []<typename T>(const T& param) {   // OK since C++20
             T tmp{};   // declare object with type of the template parameter
             ...
           };
```

Template parameters for lambdas provide the benefit to have a name for the type or a part of it when declaring generic parameters. For example:

```
[]<typename T>(T* ptr) {   // OK since C++20
  ...     // can use T as type of the value ptr points to
};
```

Or:

```
[]<typename T, int N>(T (&arr)[N]) {
  ...     // can use T as element type and N as size of the passed array
};
```

If you wonder why not to use a function template in these cases, remember that lambdas provide some benefits that functions cannot provide:

- They can be defined inside functions.
- They can capture runtime values to specify their functional behavior at runtime.
- You can pass them as arguments without the need to specify the parameter types.

11.1.1 Using Template Parameters for Generic Lambdas in Practice

Explicit template parameters can be useful to specialize (or partially restrict the type of parameters of) generic lambdas. Consider the following example:

```
[]<typename T>(const std::vector<T>& vec) {  // can only pass vectors
  ...
};
```

This lambda accepts only vectors as arguments. When using `auto` it would be not easy to restrict the argument to be vectors, because C++ does not (yet) support something like `std::vector<auto>`. However, in this case you could also use type constraints to constrain the type of the parameter (such as to require random access or even a specific type).

Explicit template parameters also help to avoid the need of `decltype`. For example, for perfect forwarding a generic parameter pack in a lambda you can write:

```
[]<typename... Types>(Types&&... args) {
  foo(std::forward<Types>(args)...);
};
```

instead of:

```
[] (auto&&... args) {
  foo(std::forward<decltype(args)>(args)...);
};
```

A similar example would be code to provide special behavior of a certain type when visiting a `std::variant<>` (introduced in C++17):

```
std::variant<int, std::string> var;
...
// call generic lambda with type-specific behavior:
std::visit([](const auto& val) {
             if constexpr(std::is_same_v<decltype(val), const std::string&>) {
               ...      // string specific processing
             }
             std::cout << val << '\n';
           },
           var);
```

We have to use `decltype()` to get the type of the parameter and compare that type as a `const&` (or remove `const` and the reference). Since C++20, you can just write the following:

```
std::visit([]<typename T>(const T& val) {  // since C++20
             if constexpr(std::is_same_v<T, std::string>) {
               ...      // string specific processing
             }
             std::cout << "value: " << val << '\n';
           },
           var);
```

You can also declare template parameters for `consteval` lambdas to force that they are executed at compile time.

11.1.2 Explicit Specification of Lambda Template Parameters

Lambdas provide a convenient way to define function objects (functors). For generic lambdas their function call operator (`operator()`) is a template. With the syntax to specify a name for the parameter instead of using `auto`, you have a name for the template parameter in the generated function call operator.

For example, if you define the following lambda:

```
auto primeNumbers = [] <int Num> () {
                      std::array<int, Num> primes{};
                      ...  // compute and assign first Num prime numbers
                      return primes;
                    };
```

the compiler defines a corresponding *closure type*:

```
class NameChosenByCompiler {
 public:
  ...
  template<int Num>
  auto operator() () const {
    std::array<int, Num> primes{};
    ...  // compute and assign first Num prime numbers
    return primes;
  }
};
```

and creates an object of this class (using the default constructor if no values are captured):

```
auto primeNumbers = NameChosenByCompiler{};
```

To explicitly specify the template parameter, you have to pass it to `operator()` when you use the lambda as a function:

```
// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers.operator()<20>();
```

There is no way to avoid specifying `operator()` when specifying its template parameters except using an indirect call.

You can try to make the template parameter deducible by making it a compile-time value; however, the resulting syntax is not much better:[1]

```
auto primeNumbers = [] <int Num> (std::integral_constant<int, Num>) {
                      std::array<int, Num> primes{};
                      ...  // compute and assign first Num prime numbers
                      return primes;
```

[1] Thanks to Arthur O'Dwyer and Jonathan Wakely for pointing this out.

```
                    };

  // initialize array with the first 20 prime numbers:
  auto primes20 = primeNumbers(std::integral_constant<int,20>{});
```

Alternatively, you might think about using a *variable template*, a technique introduced in C++14. With it you can make the variable `primeNumbers` generic instead of making the lambda generic:

```
template<int Num>
auto primeNumbers = [] () {
                      std::array<int, Num> primes{};
                      ... // compute and assign first Num prime numbers
                      return primes;
                    };
...
// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers<20>();
```

However, in that case you cannot define the lambda inside a function scope. Generic lambdas allow you to define generic functionality locally inside a scope.

11.2 Calling the Default Constructor of Lambdas

Lambdas provide a simple way to define function objects. If you define

```
auto cmp = [] (const auto& x, const auto& y) {
             return x > y;
           };
```

this is equivalent to defining a class (the *closure type*) and creating an object of this class:

```
class NameChosenByCompiler {
 public:
  template<typename T1, T2>
  auto operator() (const T1& x, const T2& y) const {
    return x > y;
  }
};

auto cmp = NameChosenByCompiler{};
```

The generated closure type has `operator()` defined so that you can use the lambda object `cmp` as a function:

```
cmp(val1, val);    // yields the result of 42 > obj2
```

However, before C++20 the generated closure type has no callable default constructor and assignment operator. Objects of the generated class could only be initially created by the compiler (copying is possible, though):

```
auto cmp1 = [] (const auto& x, const auto& y) {
```

```
              return x > y;
            };

auto cmp2 = cmp1;      // OK, copy constructor supported
decltype(cmp1) cmp3;   // ERROR until C++20: no default constructor provided
cmp1 = cmp2;           // ERROR until C++20: no assignment operator provided
```

As a consequence, you could not easily pass a lambda as sorting criterion or hash function to a container, where the ***type*** of the helper function was required. Consider we have a class `Customer`:

```
class Customer
{
 public:
  ...
  std::string getName() const;
};
```

To use the name returned by `getName()` as ordering criterion or value for the hash function, you had to pass both the type and the lambda as template and call parameter:

```
// create balanced binary tree with user-defined ordering criterion:
auto lessName = [] (const Customer& c1, const Customer& c2) {
                  return c1.getName() < c2.getName();
                };
std::set<Customer, decltype(lessName)> coll1{lessName};

// create hash table with user-defined hash function:
// (requires also operator == or a equality criterion)
auto hashName = [] (const Customer& c) {
                  return std::hash<std::string>{}(c.getName());
                };
std::unordered_set<Customer, decltype(hashName)> coll2{0, hashName};
```

The containers get the lambda when they are initialized so that they can internally use a copy of the lambda (for the unordered containers, you have to pass a minimum bucket size before). To compile, the type of the containers needs the type of the lambda.

Since C++20, lambdas with no captures have a default constructor and an assignment operator:

```
auto cmp1 = [] (const auto& x, const auto& y) {
              return x > y;
            };

auto cmp2 = cmp1;      // OK, copy constructor supported
decltype(cmp1) cmp3;   // OK since C++20
cmp1 = cmp2;           // OK since C++20
```

For this reason, it is now enough to pass the type of the lambda for the ordering criterion or hash function respectively:

```
// create balanced binary tree with user-defined ordering criterion:
```

```
auto lessName = [] (const Customer& c1, const Customer& c2) {
                  return c1.getName() < c2.getName();
                };
std::set<Customer, decltype(lessName)> coll1;            // OK since C++20

// create hash table with user-defined hash function:
// (requires also operator == or a equality criterion)
auto hashName = [] (const Customer& c) {
                  return std::hash<std::string>{}(c.getName());
                };
std::unordered_set<Customer, decltype(hashName)> coll2;  // OK since C++20
```

This works, because the argument for the ordering criterion or hash function has a default value, which is a default-constructed object of the type of the ordering criterion or hash function. And as lambdas without captures have a default constructor since C++20, the initialization with a default-constructor objects of the type of the lambda compiles now.

You can even define the lambda inside the declaration of the container and use `decltype` to pass its type. For example, you can declare an associative container with an ordering criterion defined in the declaration as follows:

```
// create balanced binary tree with user-defined ordering criterion:
std::set<Customer,
         decltype([] (const Customer& c1, const Customer& c2) {
                    return c1.getName() < c2.getName();
                  })> coll3;                             // OK since C++20
```

The same way you can declare an unordered container with a hash function defined in the declaration as follows:

```
// create hash table with user-defined hash function:
std::unordered_set<Customer,
                   decltype([] (const Customer& c) {
                              return std::hash<std::string>{}(c.getName());
                            })> coll;                    // OK since C++20
```

See *lang/lambdahash.cpp* for a complete example.

11.3 Lambdas as Non-Type Template Parameters

Since C++20, lambdas can be used as non-type template parameters (NTTP's):

```
template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}

auto defaultTax = [] {  // OK
```

```
  return 0.19;
};

std::cout << addTax<defaultTax>(100) << '\n';
```

This feature is a side effect of the new support for using literal types with public members only as non-type template parameter types. See the chapter about non-type template parameter extensions for a detailed discussion and complete example.

11.4 consteval Lambdas

By using the new consteval keyword with lambdas, you can now require that lambdas become *immediate functions* so that "function calls" of them have to be evaluated at compile time. For example:

```
auto hashed = [] (const char* str) consteval {
                ...
              };

auto hashWine = hashed("wine");   // hash() called at compile time
```

Due to the use of consteval in the definition of the lambda, any call has to happen at compile-time with values known at compile-time. Passing a runtime value is an error:

```
const char* s = "beer";
auto hashBeer = hashed(s);            // ERROR

constexpr const char* cs = "water";
auto hashWater = hashed(cs);          // OK
```

Note that hashed itself does not have to be constexpr. It is a runtime object of the lambda for which the "function call" is performed at compile time.

The discussion of consteval for lambdas in the section about the new consteval keyword provides more details of this example.

You can also use the new template syntax for generic lambdas with consteval. This enables programmers to define the initialization of compile-time function inside another function. For example:

```
// local compile-time computation of Num prime numbers:
auto primeNumbers = [] <int Num> () consteval {
                      std::array<int, Num> primes;
                      int idx = 0;
                      for (int val = 1; idx < Num; ++val) {
                        if (isPrime(val)) {
                          primes[idx++] = val;
                        }
                      }
                      return primes;
                    };
```

See *lang/lambdaconsteval.cpp* for a complete program using this lambda.

Note that in this case the template parameter is not deduced. Therefore, the syntax to explicitly specify the template parameter becomes a bit ugly:

```
auto primes = primeNumbers.operator()<100>();
```

Note also that you always have to provide the parameter list before `consteval` (the same applies when specifying `constexpr` there). You cannot skip the parentheses even if there is no parameter declared.

11.5 Changes for Capturing

C++20 introduces several new extensions to the capturing of values and objects in lambdas.

11.5.1 Capturing `this` and `*this`

If a lambda is defined inside a member function, the question is how you can access data of the object for which the member function is called. Before C++20, we had the following rules:

```
class MyType {
  std::string name;
  ...
  void foo() {
    int val = 0;
    ...
    auto l0 = [val] { bar(val, name); };          // ERROR: member name not captured
    auto l1 = [val, name] { bar(val, name); };    // OK, capture val and name by value

    auto l2 = [&] { bar(val, name); };            // OK (val and name by reference)
    auto l3 = [&, this] { bar(val, name); };      // OK (val and name by reference)
    auto l4 = [&, *this] { bar(val, name); };     // OK (val by reference, name by value)

    auto l5 = [=] { bar(val, name); };            // OK (val by value, name by reference)
    auto l6 = [=, this] { bar(val, name); };      // ERROR before C++20
    auto l7 = [=, *this] { bar(val, name); };     // OK (val and name by value)
    ...
  }
};
```

Since C++20, the following rules apply:

```
class MyType {
  std::string name;
  ...
  void foo() {
    int val = 0;
    ...
    auto l0 = [val] { bar(val, name); };          // ERROR: member name not captured
    auto l1 = [val, name] { bar(val, name); };    // OK, capture val and name by value
```

```
      auto l2 = [&] { bar(val, name); };          // deprecated (val and name by ref.)
      auto l3 = [&, this] { bar(val, name); };    // OK (val and name by reference)
      auto l4 = [&, *this] { bar(val, name); };   // OK (val by reference, name by value)

      auto l5 = [=] { bar(val, name); };          // deprecated (val by value, name by ref.)
      auto l6 = [=, this] { bar(val, name); };    // OK (val by value, name by reference)
      auto l7 = [=, *this] { bar(val, name); };   // OK (val and name by value)
      ...
    }
};
```

Thus, since C++20 we have the following changes:

- `[=, this]` is now allowed as a lambda capture (some compilers did allow it before although it was formally invalid).
- The implicit capture of `*this` is deprecated.

11.5.2 Capturing Structured Bindings

Since C++20, it is allowed to capture structured bindings (introduced with C++17):

```
std::map<int, std::string> mymap;
...

for (const auto& [key,val] : mymap) {
   auto l = [key, val] {  // OK since C++20
              ...
            };
   ...
}
```

Some compilers did allow capturing structure bindings before although it was formally invalid.

11.5.3 Capturing Parameter Packs of Variadic Templates

If you have a variadic template, you could capture parameter packs as follows:

```
template<typename... Args>
void foo(Args... args)
{
   auto l1 = [&] {
               bar(args...);  // OK
             };
   auto l2 = [args...] {      // or [=]
               bar(args...);  // OK
             };
   ...
}
```

However, f you wanted to return the created lambda for later use, there was a problem:

- Using `[&]` you would return a lambda that refers to a destroyed parameter pack.
- Using `[args...]` or `[=]` you would copy the passed parameter pack.

Usually you can use *init-captures* to use move semantics when capturing objects:

```
template<typename T>
void foo(T arg)
{
   auto l3 = [arg = std::move(arg)] {  // OK since C++14
               bar(arg);       // OK
             };
   ...
}
```

However, there was no syntax provided to use *init-captures* with parameter packs.

C++20 introduces a corresponding syntax:

```
template<typename... Args>
void foo(Args... args)
{
   auto l4 = [...args = std::move(args)] {  // OK since C++20
               bar(args...);  // OK
             };
   ...
}
```

You can also init-capture parameter packs by reference. For example, we can change to parameter name for the lambda as follows:

```
template<typename... Args>
void foo(Args... args)
{
   auto l4 = [&...fooArgs = args] {          // OK since C++20
               bar(fooArgs...);  // OK
             };
   ...
}
```

Example Capturing Parameter Packs of Variadic Templates

A generic function creating and returning a lambda that captures a variadic number of parameters by value now looks as follows:

```
template<typename Callable, typename... Args>
auto createToCall(Callable op, Args... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
           return op(args...);
```

```
          };
  }
```

Using the new syntax for abbreviated function templates the code looks as follows:

```
auto createToCall(auto op, auto... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
           return op(args...);
         };
}
```

Here is a complete example:

lang/capturepack.cpp

```
#include <iostream>
#include <string_view>

auto createToCall(auto op, auto... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
           return op(args...);
         };
}

void printWithGAndNoG(std::string_view s)
{
  std::cout << s << "g " << s << '\n';
}

int main()
{
  auto printHero = createToCall(printWithGAndNoG, "Zhan");
  ...

  printHero();
}
```

Run it to print a hero!

11.6 Afternotes

The template syntax for generic lambdas was first proposed by Louis Dionne in http://wg21.link/p0428r0. The finally accepted wording was formulated by Louis Dionne in http://wg21.link/p0428r2.

The template syntax for generic lambdas was first proposed by Louis Dionne in http://wg21.link/p0624r0. The finally accepted wording was formulated by Louis Dionne in http://wg21.link/p0624r2.

Allowing lambdas as non-type template parameters was introduced as finally formulated by Jeff Snyder and Louis Dionne in http://wg21.link/p0732r2.

Support for `consteval` lambdas was introduced as finally formulated by Richard Smith, Andrew Sutton, Daveed Vandevoorde in http://wg21.link/p1073r3.

in http://wg21.link/p1073r3. The changes of the rules for capturing `this` and `*this` were first proposed by Thomas Köppe in http://wg21.link/p0409r0 and http://wg21.link/p0806r0. The finally accepted wording was formulated by Thomas Köppe in http://wg21.link/p0409r2 and http://wg21.link/p0806r2.

Capturing structured bindings was introduced as finally formulated by Nicolas Lesser in http://wg21.link/p1091r3.

Init-capturing parameter packs was accepted as finally formulated by Barry Revzin in http://wg21.link/p0780r2 and http://wg21.link/p2095r0.

Chapter 12

Other C++ Language Improvements

This chapter presents several other small features and extensions C++20 introduces for its core language.

12.1 New Character Type `char8_t`

For better UTF-8 support, C++20 introduces the new character type **`char8_t`** and a new corresponding string type `std::u8string`.

Type `char8_t` is a new keyword and serves as ***the*** type to hold UTF-8 characters and character sequences. For example:

```
char8_t c = u8'@';                    // character with UTF-8 encoding for character @
const char8_t* s = u8"K\u00F6ln";     // character sequence with UTF-8 encoding for Köln
```

The introduction of this type is a breaking change:

- `u8` character literals now use `char8_t` instead of `char`
- For UTF-8 strings the new types `std::u8string` and `std::u8string_view` are used now

Consider the following example:

```
auto c = u8'@';                       // character with UTF-8 encoding for @
auto s1 = u8"K\u00F6ln";              // character sequence with UTF-8 encoding for Köln
using namespace std::literals;
auto s2 = u8"K\u00F6ln"s;             // string with UTF-8 encoding for Köln
auto sv = u8"K\u00F6ln"sv;            // string view with UTF-8 encoding for Köln
```

The types of `c` and `s` have changed here:

- **Before C++20** this was equivalent to:

```
char c = u8'@';                       // UTF-8 character type before C++20
const char* s1 = u8"K\u00F6ln";       // UTF-8 character sequence type before C++20
using namespace std::literals;
std::string s2 = u8"K\u00F6ln"s;      // UTF-8 string type before C++20
std::string_view sv = u8"K\u00F6ln"s; // UTF-8 string view type before C++20
```

- **Since C++20** this is equivalent to:

```
char8_t c = u8'@';                          // UTF-8 character type since C++20
const char8_t* s1 = u8"K\u00F6ln";          // UTF-8 character sequence type since C++20
using namespace std::literals;
std::u8string s2 = u8"K\u00F6ln"s;          // UTF-8 string type since C++20
std::u8string_view sv = u8"K\u00F6ln"sv; // UTF-8 string view type since C++20
```

The reason for this change is simple: we can now implement special behavior for UTF-8 characters and strings:

- We can overload for UTF-8 character sequences:

```
void store(const char* s)
{
  storeInFile(convertToUTF8(s));   // store after converting to UTF-8 encoding
}

void store(const char8_t* s)
{
  storeInFile(s);                  // store as is because it has already UTF-8 encoding
}
```

- We can implement special behavior in generic code:

```
void store(const CharT* s)
{
  if constexpr(std::same_as<CharT, char8_t>) {
    storeInFile(s);                  // store as is because it has already UTF-8 encoding
  }
  else {
    storeInFile(convertToUTF8(s));   // store after converting to UTF-8 encoding
  }
}
```

Still you can only store UTF-8 characters of one byte in an object of type `char8_t` (remember that UTF-8 characters have a variable width):

- The ampersand character `@` has the decimal value 64 (hexadecimal `0x40`). Therefore, it is possible to store its value in a `char8_t` and for this reason the `u8` character literal is well defined:

```
char8_t c = u8'@';                          // OK (c has value 64)
```

- The character for the European currency *Euro*, €, consists of three code units: 226 130 172 (hexadecimal: `0xE2 0x82 0xAC`). Therefore, it is ***not*** possible to store its value in a `char8_t`:

```
char8_t cEuro = u8'€';                      // ERROR: invalid character literal
```

  Instead you have to initialize a character sequence or UTF-8 string:

```
const char8_t* cEuro = u8"\u20AC";      // OK
std::u8string  sEuro = u8"\u20AC";      // OK
```

  Here we use the Unicode notation to specify the value of the UTF-8 character, which creates an array of four `const char8_t` (including trailing null character), which is then used to initialize `cEuro` and `sEuro`.

If your compiler accepts the character € in your source code (which means it has to support a source file encoding with a character set such as UTF-8 or ISO-8859-15), you could even use this symbol in literals directly:[1]

```
const char8_t* cEuro = u8"€";          // OK if valid character for the compiler
std::u8string  sEuro = u8"€";          // OK if valid character for the compiler
```

However, because this source code is not portable, you better use Unicode characters (such as \u20AC for the € symbol).

Note that a few things in the C++ standard library changed accordingly:

- Overloads for the new character type char8_t were added
- Functions using or returning UTF-8 strings use the new UTF-8 string types

In addition, note that char8_t is not guaranteed to have 8 bits. It is defined as internally using an unsigned char, which typically has 8 bits, but may have more. As usual, you can use std::numeric_limits<> to check for its number of bits:

```
std::cout << "char8_t has "
          << std::numeric_limits<char8_t>::digits << " bits\n";
```

12.1.1 Changes in the C++ Standard Library for char8_t

The C++ standard library changed the following to support char8_t:

- u8string is now provided (defined as std::basic_string<char8_t>)
- u8string_view is now provided (defined as std::basic_string_view<char8_t>)
- std::numeric_limits<char8_t> is now defined
- std::char_traits<char8_t> is now defined
- Hash functions for char8_t strings and string views are now provided
- std::mbrtoc8() and c8rtomb() are now provided
- codecvt facets to convert between char8_t and char16_t or char32_t are now provided
- For filesystem paths, u8string() now returns std::u8string instead of std::string
- std::atomic<char8_t> is now provided

Note that these changes might break existing code when being compiled with C++20, which is discussed next.

12.1.2 Broken Backward Compatibility

Because C++20 changes the type of UTF-8 literals and signatures returning UTF-8 strings, code using UTF-8 characters might no longer compile.

[1] Thanks to Tom Honermann for pointing this out.

Broken Code Using the Character Type

For example:

```
std::string s0 = u8"text";   // OK in C++17, ERROR since C++20

auto s = u8"K\u00F6ln";      // s is const char* until C++17, but const char8_t* since C++20
const char* s2 = s;          // OK in C++17, ERROR since C++20
std::cout << s << '\n';      // OK in C++17, ERROR since C++20

auto c = u8'c';              // c1 is char in C++17, but char8_t since C++20
char c2 = c;                 // OK (even if char8_t)
char* cp = &c;               // OK in C++17, ERROR since C++20
std::cout << c;              // OK in C++17, ERROR since C++20
```

Broken Code Using the New String Types

Especially code returning UTF-8 strings might now lead to problem.

For example, the following code does no longer compile, because `u8string()` returns a `std::u8string` now instead of a `std::string`:

```
// iterate over directory entries:
for (const auto& entry : fs::directory_iterator(path)) {
  std::string name = entry.path().u8string();     // OK in C++17, ERROR since C++20
  ...
}
```

You have to adjust the code by using a different type, `auto`, or support both types:

```
// iterate over directory entries (C++17 and C++20):
for (const auto& entry : fs::directory_iterator(path)) {
#ifdef __cpp_char8_t
  std::u8string name = entry.path().u8string();   // OK since C++20
#else
  std::string name = entry.path().u8string();     // OK in C++17
#endif
  ...
}
```

Broken Code I/O for UTF-8 Strings

You can also no longer print UTF-8 characters or strings to `std::cout` (or any other standard output stream):

```
std::cout << u8"text";       // OK in C++17, ERROR since C++20
std::cout << u8'X';          // OK in C++17, ERROR since C++20
```

In fact, C++20 deleted the output operator for all extended character types (`wchar_t`, `char8_t`, `char16_t`, `char32_t`) unless the output stream supports the same character type:

```
std::cout << "text";          // OK
std::cout << L"text";         // wchar_t string: OOPS in C++17, ERROR since C++20
std::cout << u8"text";        // UTF-8 string: OK in C++17, ERROR since C++20
std::cout << u"text";         // UTF-16 string: OOPS in C++17, ERROR since C++20
std::cout << U"text";         // UTF-32 string: OOPS in C++17, ERROR since C++20

std::wcout << "text";         // OK
std::wcout << L"text";        // OK
std::wcout << u8"text";       // UTF-8 string: OK in C++17, ERROR since C++20
std::wcout << u"text";        // UTF-16 string: OOPS in C++17, ERROR since C++20
std::wcout << U"text";        // UTF-32 string: OOPS in C++17, ERROR since C++20
```

Note the statements marked with *OOPS*. They all compiled in C++17, but they printed the addresses of the strings instead of their values. So, C++20 disables not only the output of UTF-8 characters but also output that did not work at all.

Dealing with Broken Code

You might wonder now, how to deal with code that formerly worked with UTF-8 characters. The easiest way to deal with it is to use `reinterpret_cast<>`:

```
auto s = u8"text";                              // s is const char8_t* since C++20
std::cout << s;                                 // ERROR since C++20
std::cout << reinterpret_cast<const char*>(s);  // OK
```

For single character using a static_cast is enough:

```
auto c = u8'x';                                 // c is char8_t since C++20
std::cout << c;                                 // ERROR since C++20
std::cout << static_cast<char>(c);              // OK
```

You can bind its use or the use of other workarounds on the feature test macro for the `char8_t` characters feature:[2]

```
auto s = u8"text";                              // s is const char8_t* since C++20
#ifdef __cpp_char8_t
std::cout << reinterpret_cast<const char *>(s); // OK
#else
std::cout << s;                                 // OK in C++17, ERROR since C++20
#endif
```

If you wonder why C++20 does not provide a working output operator for UTF-8 characters, please note that this is a this is a pretty complex issue, for which we simply did not have enough time to solve it yet. You can read more about this here: http://stackoverflow.com/a/58895428

Because using `reinterpret_cast<>` might not scale when UTF-8 characters are heavily used, Tom Honermann wrote a guideline for how to deal with code for UTF-8 characters before and since C++20:

[2] For the final behavior specified with C++20, `__cpp_char8_t` should have (at least) the value 201907.

"*P1423R3* `char8_t` *Backward Compatibility Remediation*." If you deal with UTF-8 characters and string, you should definitely read it. You can download it here: `http://wg21.link/p1423`

12.2 Designated Initializers

C++ provides now a way to specify for initializers of aggregates, which member should be initialized with a value. However, you can use this only to skip parameters, not to revert the order.

For example, assume we have the following aggregate type:

```
struct Value {
  double amount = 0;
  int precision = 2;
  std::string unit = "Dollar";
};
```

The the following way to initialize values of this type are allowed:

```
Value v1{100};                              // OK (not designated initializers)
Value v2{.amount = 100, .unit = "Euro"};    // OK (second member has default value)
Value v3{.precision = 8, .unit = "$"};      // OK (first member has default value)
```

See *lang/designated.cpp* for a complete example.

Note that following constraints:

- You have pass the initial value with = or {}.
- You can skip members, but you have to follow their order.
- You either have to use designated initializers for all or none of the arguments. Mixed initialization is not allowed.
- Nested initialization with designated initializers is possible, but not directly using `.mem.mem`.
- You cannot use designated initializers when initializing aggregates with parentheses.
- Designated initializers can also be used in unions.
- Using designated initialization for arrays is not supported.

The restriction to follow the order, using them for all or none of the arguments, not supporting direct nesting, and not support arrays is a restrictions compared to the programming language C. The restriction to respect the order of the members has the reason that the initialization should reflect the order in which constructors are called (which is the opposite order destructors are called).

For example:

```
Value v4{100, .unit = "Euro"};              // ERROR: all or none designated
Value v5{.unit = "$", .amount = 20};        // ERROR: invalid order
Value v6(.amount = 29.9, .unit = "Euro");   // ERROR: only with {}
```

And as an example of nested initialization with = and {} and unions:

```
union Sub {
  double x = 0;
  int y = 0;
};
```

```
struct Data {
  std::string name;
  Sub val;
};

Data d1{.val{.y=42}};        // OK
Data d2{.val = {.y{42}}};    // OK
```

You cannot directly nest designated initializers:

```
Data d2{.val.y = 42};        // ERROR
```

12.3 Implicit typename for Type Members of Template Parameters

When using type members of template parameters, you usually have to qualify this use with the keyword typename:

```
template<typename T>
typename T::value_type getElem(const T& cont, typename T::iterator pos)
{
  using Itor = typename T::iterator;
  typename T::value_type elem;
  ...
  return elem;
}
```

Before C++20, all qualifications of T's type members value_type and iterator were necessary. Since C++20, you can skip typename in contexts where it is clear that a type is passed. In this case this applies to the specification of the return type and the type used in the alias declaration (where using introduces a new name for a type):

```
template<typename T>
T::value_type getElem(const T& cont, typename T::iterator pos)
{
  using Itor = T::iterator;
  typename T::value_type elem;
  ...
  return elem;
}
```

Note that the parameter pos and the variable elem still need typename. The most important places, where you can skip typename now are:

- When declaring return types (unless local forward declarations)
- When declaring members in class templates
- When declaring parameters of member functions in class templates
- For the types in an alias declarations

Because the rules for implicit `typename` are partially pretty subtle, you might simply still always use `typename` when using the type member of a template parameter.[3]

12.3.1 Rules for Implicit `typename`

Since C++20, you can skip `typename` when using a type member for a template parameters in the following situations:

- In an alias declaration (i.e., when declaring a type name with `using`). Note that a type declaration with `typedef` still needs `typename`.
- When defining or declaring the return type of a function (unless the declaration happens inside a function or block scope)
- When declaring a trailing return type
- When specifying the target type for `static_cast`, `const_cast`, `reinterpret_cast`, or `dynamic_cast`
- When specifying the type for `new`
- Inside a class when
  - declaring a data member
  - declaring the return type of a member function
  - declaring a parameter of a member or friend function or lambda (a default argument might still need it)
- When declaring a default value for a type parameter of a template
- When declaring the type of a non-type template parameter

Note that before C++20 `typename` was not necessary in a few other situations:

- When specifying the base type of an inherited class
- When passing initial values to the base class in a constructor
- When using a type member inside the class declaration

The following example demonstrates most of the cases above (here, `TYPENAME` is used when `typename` is optional since C++20):

```
template<typename T,
         auto ValT = typename T::value_type{}>     // typename required
class MyClass {
  TYPENAME T::value_type val;                      // typename optional
 public:
  using iterator = TYPENAME T::iterator;           // typename optional

  TYPENAME T::iterator begin() const;              // typename optional
  TYPENAME T::iterator end() const;                // typename optional
  void print(TYPENAME T::iterator);                // typename required
  template<typename T2 = TYPENAME T::value_type>   // typename required
    void assign(T2);
};
```

[3] Thanks to Arthur O'Dwyer for pointing this out.

```
template<typename T>
TYPENAME T::value_type                              // typename optional
foo(const T& cont, typename T::value_type arg)      // typename required
{
  typedef typename T::value_type ValT2;             // typename required
  using ValT1 = TYPENAME T::value_type;             // typename optional
  typename T::value_type val;                       // typename required

  typename T::value_type other1(void);              // typename required
  auto other2(void) -> TYPENAME T::value_type;      // typename optional

  auto l1 = [] (TYPENAME T::value_type) {           // typename optional
            };

  auto p = new TYPENAME T::value_type;              // typename optional
  val = static_cast<TYPENAME T::value_type>(0);     // typename optional
  ...
}
```

12.4 Afternotes

The request for a distinct type for UTF-8 character was was first proposed by Tom Honermann in http://wg21.link/p0482r0. The finally accepted wording was formulated by Tom Honermann in http://wg21.link/p0482r6. The corresponding fixes to the output operators were accepted as proposed by Tom Honermann in http://wg21.link/p1423r3.

The request to support designated initializers was first proposed by Tim Shen, Richard Smith, Zhihao Yuan, and Chandler Carruth in http://wg21.link/p0329r0. The finally accepted wording was formulated by Tim Shen and Richard Smith in http://wg21.link/p0329r4.

Making `typename` optional in certain situations was first proposed by by Daveed Vandevoorde in http://wg21.link/p0634r0. The finally accepted wording was formulated by Nina Ranns and Daveed Vandevoorde in http://wg21.link/p0634r3.

This page is intentionally left blank

Chapter 13

Formatted Output

The C++ IOStream library provides only inconvenient and limited ways to support formatted output (specifying a field width, fill characters, etc.). For that reason, formatted output still often uses functions like `sprintf()`.

C++20 introduces a new library for formatted output, which is described in this chapter. It allows a convenient specification of formatting attributes and is extensible.

13.1 Formatted Output by Example

Before we go into details, let us look at some motivating examples.

13.1.1 Using `std::format()`

The key approach is provided by the function `std::format()`, which allows you to specify a string in which placeholders of braces can be used to refer to passed arguments:

```
#include <format>
...

std::string str{"hello"};
...
std::cout << std::format("String '{}' has {} chars\n", str, str.size());
std::cout << std::format("{1} is the size of string '{0}'\n", str, str.size());
```

The output of these two statements are:

```
String 'hello' has 5 chars
5 is the size of string 'hello'
```

The function `std::format()`, defined in `<format>`, takes a format string (a string literal, string, or string view known at compile time), in which `{}` represents the next argument (with the default formatting of its type), and yields a `std::string` with with the formatted inserted arguments. An optional integral value

between the braces specifies the index of the argument so that you can process them in different order or use them multiple times.

Note that you do not have to explicitly specify the type of the argument. That means you can easily use `std::format()` in generic code. For example:

```
void print2(const auto& arg1, const auto& arg2)
{
  std::cout << std::format("args: {} and {}\n", arg1, arg2);
}
```

You could call this function as follows:

```
print2(7.7, true);                          // prints:args: 7.7 and true
```

This even works for user-defined types, which also support formatted output. Formatted output of the chrono library is one example:

```
print2("now", std::chrono::seconds{13});  // prints: args: of now and 13s
```

In addition, you can specify the formatting of the passed argument. For example, you can define a field width:

```
std::format("{:7}", 42)          // yields "     42"
std::format("{:7}", 42.0)        // yields "     42"
std::format("{:7}", 'x')         // yields "x      "
std::format("{:7}", true)        // yields "true   "
```

Note that the different types have different default alignment. Also note that for a `bool` the values `false` and `true` are printed instead of `0` and `1` as for iostream output with operator `<<`.

You can also explicitly specify the alignment (`<` for left, `^` for center, and `>` for right) and specify a fill character:

```
std::format("{:*<7}", 42)        // yields "42*****"
std::format("{:*>7}", 42)        // yields "*****42"
std::format("{:*^7}", 42)        // yields "**42***"
```

Several additional formatting specifications are possible to force a specific notation, a specific precision (or limit strings to a certain size), fill characters, and a positive sign:

```
std::format("{:#7}", 42.0)       // yields "    42."
std::format("{:7.2f}", 42.0)     // yields "  42.00"
std::format("{:7.4}", "corner")  // yields "corn   "
std::format("{:+07d}", '+')      // decimal value of char (e.g., "+000043")
```

13.1.2 Using `std::format_to_n()`

The implementation of `std::format()` has a pretty good performance when compared with other ways of formatting. However, memory has to be allocated for the resulting string. To save time you can force `std::format()` to write to a preallocated array of characters. In that case, you have to specify both the buffer to write to and its size. For example:

```
char buffer[64];
...
```

```
auto ret = std::format_to_n(buffer, std::size(buffer) - 1,
                            "String '{}' has {} chars\n", str, str.size());
*(ret.out) = '\0';
```

or:

```
std::array<char, 64> buffer;
...
auto ret = std::format_to_n(buffer.begin(), buffer.size() - 1,
                            "String '{}' has {} chars\n", str, str.size());
*(ret.out) = '\0';   // write trailing null terminator
```

Note that `std::format_to_n()` does ***not*** write a trailing null terminator. However, the return value holds all information to deal with this. It is a data structure of type `std::format_to_n_result` that has two members:

- `out` for the position of the first character not written
- `size` for the number of characters that would have been written without truncating them to the passed size.

So, either we process `ret.size` or store a null terminator at the end, where `ret.out` points to (that's why we only pass `buffer.size()-1` with room to write):

```
*(ret.out) = '\0';
```

Note also that the output is just cut off if the size does not fit. Thus, we have to write it manually if we want to use the output as a valid C string.

13.1.3 Using `std::format_to()`

By using `std::format_to()` and an output stream buffer iterator, you can even write directly to a stream:

```
std::format_to(std::ostreambuf_iterator<char>{std::cout},
               "String '{}' has {} chars\n", str, str.size());
```

In general, `std::format_to()` takes any output iterator for characters. For example, you can also use a back inserter to append the characters to a string:

```
std::string s;
std::format_to(std::back_inserter(s),
               "String '{}' has {} chars\n", str, str.size());
```

The helper function `std::back_inserter()` creates an object that calls `push_back()` for each character. Note that the implementation of `std::format_to()` can recognize that back insert iterators are passed and write multiple characters for certain containers at once so that we still have good performance.[1]

[1] Thanks to Victor Zverovich for pointing that out.

13.1.4 Using `std::formatted_size()`

To know how many characters will be written (without writing any), you can use `std::formatted_size()`. For example:

```
auto sz = std::formatted_size("String '{}' has {} chars\n", str, str.size());
```

That way, you could reserve or check ahead for enough memory.

13.2 Formatted Output in Detail

This section described the syntax of formatting in detail.

13.2.1 General Format of Format Strings

The general way to specify the formatting of arguments is to pass a *format string* that can have *replacement fields* specified by `{...}` and characters. All other characters are printed as they are. To print the characters `{` and `}`, use `{{` and `}}`.

For example:

```
std::format("With format {{}}: {}", 42);    // yields "With format : 42"
```

The replacement fields may have an index to specify the argument and a format specifier after a colon:

- **`{}`**
  use the next argument with its default formatting
- **`{`***n*`}`**
  use the *n*-th argument (first argument has index `0`) with its default formatting
- **`{:`***fmt*`}`**
  use the next argument formatted according to *fmt*
- **`{`***n*`:`***fmt*`}`**
  use the *n*-th argument formatted according to *fmt*

You can either have none of the arguments specified by an index, or all of them:

```
std::format("{}: {}", key, value);          // OK
std::format("{1}: {0}", value, key);        // OK
std::format("{}: {} or {0}", value, key);   // ERROR
```

Additional arguments are ignored.

The syntax of the format specifier depends on the type of the passed argument.

- For arithmetic types, strings, and raw pointers, there is a standard format defined.
- C++20 specifies also a standard format for chrono types (durations, time points, and calendrical types).

13.2.2 Standard Format Specifiers

As standard format specifiers you can use the following format (each specifier is optional):

fill align sign `#` `0` *width* `.`*prec* `L` *type*

- *fill* is the character to fill the value up to *width* (default: space). It can only be specified if *align* is also specified.
- *align* is
  - `<` left-aligned
  - `>` right-aligned
  - `^` centered

  The default alignment depends on the type.
- *sign* is
  - `-` only negative sign for negative values (default)
  - `+` positive or negative sign
  - *space* negative sign or space
- `#` forces to use an *alternative form* of some notations:
  - Writes a prefix such as `0b`, `0`, and `0x` the binary, octal, and hexadecimal notation of integral values
  - Forces always to write a dot (and keep trailing zeros) for floating-point notations
- `0` in front of *width* pads arithmetic values with zeros.
- *width* specified a minimum field width.
- *prec* after a dot specifies the precision:
  - For floating-point types, it specifies how many digits are printed after the dot or in total (depending on the notation).
  - For string types, it specifies the maximum number of characters processed from the string.
- `L` turns on locale-dependent formatting (matters for arithmetic types and `bool`)
- *type* specifies the general notation of the formatting. That way you can print characters as integral values (and vice versa) or choose the general notation of floating-point values.

13.2.3 Width, Precision, and Fill Characters

For all values printed, a positive integral value after the colon (without a leading dot) specifies a minimal field width for the output of the value as a whole (including sign etc.). It can be use together with a specification of the alignment: For example:

```
std::format("{:7}", 42);        // yields "     42"
std::format("{:7}", "hi");      // yields "hi     "
std::format("{:^7}", "hi");     // yields "  hi   "
std::format("{:>7}", "hi");     // yields "     hi"
```

You can also specify padding zeros and a fill character. The padding `0` can only be used for arithmetic types (except `char` and `bool`) and is ignored if an alignment is specified:

```
std::format("{:07}", 42);       // yields "0000042"
```

```
std::format("{:^07}", 42);      // yields "  42   "
std::format("{:>07}", -1);      // yields "     -1"
```

A padding `0` is different from the general fill character that can be specified right behind the colon (in front of the alignment):

```
std::format("{:^07}", 42);      // yields "  42   "
std::format("{:0^7}", 42);      // yields "0042000"
std::format("{:07}", "hi");     // invalid (padding 0 not allowed for strings)
std::format("{:0<7}", "hi");    // yields "hi00000"
```

The precision is used for floating-point types and strings:

- For floating-point types, you can specify a different precision than the usual default 6:

  ```
  std::format("{}", 0.12345678);         // yields "0.12345678"
  std::format("{:.5}", 0.12345678);      // yields "0.12346"
  std::format("{:10.5}", 0.12345678);    // yields "   0.12346"
  std::format("{:^10.5}", 0.12345678);   // yields " 0.12346  "
  ```

  Note that depending on the floating-point notation the precision might apply to the value as a whole or the digits after the dot.

- For strings, you can use it to specify a maximum number of characters:

  ```
  std::format("{}", "counterproductive");         // yields "counterproductive"
  std::format("{:20}", "counterproductive");      // yields "counterproductive   "
  std::format("{:.7}", "counterproductive");      // yields "counter"
  std::format("{:20.7}", "counterproductive");    // yields "counter             "
  std::format("{:^20.7}", "counterproductive");   // yields "      counter       "
  ```

Note that width and precision can be parameters themselves. For example, the following code:

```
int width = 10;
int precision = 2;
for (double val : {1.0, 12.345678, -777.7}) {
  std::cout << std::format("{:+{}.{}f}\n", val, width, precision);
}
```

has the following output:

```
     +1.00
    +12.35
   -777.70
```

Here, we specify at runtime that we have a minimum field width of 10 with two digits after the dot (using the fixed notation).

13.2.4 Format/Type Specifiers

By specifying a *format* or *type specifier*, you can force various notations for integral, floating-point types, and raw pointers.

Specifiers for Integer Types

Table *Formatting Options for Integral Types* lists the possible formatting type options for integral types (including `bool` and `char`)

Spec.	`42`	`'@'`	`true`	Meaning
none	`42`	`@`	`true`	Default format
`d`	`42`	`64`	`1`	Decimal notation
`b` / `B`	`101010`	`1000000`	`1`	Binary notation
`#b`	`0b101010`	`0b1000000`	`0b1`	Binary notation with prefix
`#B`	`0B101010`	`0B1000000`	`0B1`	Binary notation with prefix
`o`	`52`	`100`	`1`	Octal notation
`x`	`2a`	`40`	`1`	Hexadecimal notation
`X`	`2A`	`40`	`1`	Hexadecimal notation
`#x`	`0x2a`	`0x40`	`0x1`	Hexadecimal notation with prefix
`#X`	`0X2A`	`0X40`	`0X1`	Hexadecimal notation with prefix
`c`	`*`	`@`	`'\1'`	As character with the value
`s`	*invalid*	*invalid*	`true`	`bool` as string

Table 13.1. Formatting Options for Integral Types

For example:

```
std::cout << std::format("{:#b} {:#b} {:#b}\n", 42, '@', true);
```

will print:

```
0b101010 0b1000000 0b1
```

Note the following:

- The default notations are:
  - `d` (decimal) for integer types
  - `c` (as character) for character types
  - `s` (as string) for type `bool`
- If `L` is specified after the notation, the local-dependent character sequence for Boolean values and the locale-dependent thousands separator and decimal point characters for arithmetic values are used.

Specifiers for Floating-Point Types

Table *Formatting Options for Floating-Point Types* lists the possible formatting type options for floating-point types.

For example:

```
std::cout << std::format("{0} {0:#} {0:#g} {0:e}\n", -1.0);
```

will print:

```
-1 -1. -1.00000 -1.000000e+00
```

Spec.	-1.0	0.0009765625	1785856.0	Meaning
none	-1	0.0009765625	1.785856e+06	Default format
#	-1.	0.0009765625	1.785856e+06	Forces decimal point
f / F	-1.000000	0.000977	1785856.000000	Fixed notation (default precision after dot: 6)
g	-1	0.000976562	1.78586e+06	Fixed or exponential notation (default full precision: 6)
G	-1	0.000976562	1.78586E+06	Fixed or exponential notation (default full precision: 6)
#g	-1.00000	0.000976562	1.78586e+06	Fixed or exponential notation (forced dot and zeros)
#G	-1.00000	0.000976562	1.78586E+06	Fixed or exponential notation (forced dot and zeros)
e	-1.000000e+00	9.765625e-04	1.7858560e+06	Exponential notation (default precision after dot: 6)
E	-1.000000E+00	9.765625E-04	1.7858560E+06	Exponential notation (default precision after dot: 6)
a	-1p+0	1p-10	1.b4p+20	Hexadec. floating-point notation
A	-1P+0	1P-10	1.B4P+20	Hexadec. floating-point notation
#a	-1.p+0	1.p-10	1.b4p+20	Hexadec. floating-point notation
#A	-1.P+0	1.P-10	1.B4P+20	Hexadec. floating-point notation

Table 13.2. Formatting Options for Floating-Point Types

Note that passing the integer `-1` instead would be a formatting error.

Specifiers for Strings

For strings types, the default format specifier is **`s`**. However, because it is its default, you usually do not have to provide this specifier. Note also that for strings you can specify a certain precision, which is interpreted as the maximum number of characters used:

```
std::format("{}", "counter");        // yields "counter"
std::format("{:s}", "counter");      // yields "counter"
std::format("{:.5}", "counter");     // yields "count"
std::format("{:.5}", "hi");          // yields "hi"
```

Note that only the standard string types of the character types `char` and `wchar_t` are supported. There is no support for strings and sequences of the types `u8string` and `char8_t`, `u16string` and `char16_t`, or `u32string` and `char32_t`, In fact, the C++ standard library provides *formatters* for the following types:

- `char*` and `const char*`
- `const char[`*n*`]` (string literals)
- `std::string` and `std::basic_string<char,` *traits*`,` *allocator*`>`
- `std::string_view` and `std::basic_string_view<char,` *traits*`>`

- `wchar_t*` and `const wchar_t*`
- `const wchar_t[`*n*`]` (wide string literals)
- `std::wstring` and `std::basic_string<wchar_t, `*traits*`, `*allocator*`>`
- `std::wstring_view` and `std::basic_string_view<wchar_t, `*traits*`>`

Note that the format string and string arguments used there have to have the same character type:

```
auto ws1 = std::format("{}", L"K\u00F6ln");            // compile-time ERROR
std::wstring ws2 = std::format(L"{}", L"K\u00F6ln");   // OK
```

Specifiers for Pointers

For strings types, the default format specifier is **`p`**, which which usually writes the address in hexadecimal notation with the prefix `0x` (only on platforms that have no type `uintptr_t` the format is implementation-defined):

```
void* ptr = ...;
std::format("{}", ptr)          // usually yields 0x7ff688ee64 or so
std::format("{:p}", ptr)        // usually yields 0x7ff688ee64 or so
```

Note that only the following pointer types are supported:

- `void*` and `const void*`
- `std::nullptr_t`

Thus, you can either pass `nullptr` or a raw pointer, which you have to cast to type `(const) void*`:

```
int i = 42;
std::format("{}", &i)                                 // compile-time error
std::format("{}", static_cast<void*>(&i))             // OK (usually 0x7ff688ee64 or so)
std::format("{:p}", static_cast<void*>(&i))           // OK (usually 0x7ff688ee64 or so)
std::format("{}", static_cast<const void*>("hi"))     // OK (usually 0x7ff688ee64 or so)
std::format("{}", nullptr)                            // OK (usually 0x0)
std::format("{:p}", nullptr)                          // OK (usually 0x0)
```

13.3 Error Handling

Ideally, C++ compilers should detect bugs at compile time rather than at runtime. Because string literals are known at compile time, C++ can check for format violations when string literals are used as format strings and does so in `std::format()`:[2]

```
std::format("{:d}", 42)                               // OK
std::format("{:s}", 42)                               // compile-time ERROR
```

If you pass a format string that was initialized or computed before, the formatting library handles format errors as follows:

[2] The requirement to check format strings at compile time was proposed with http://wg21.link/p2216r3 and accepted as a fix against C++20 after it was standardized.

- `std::format()` `std::format_to()`, `format_to_n()`, and `std::formatted_size()` take only format strings known at compile time:
  - String literals
  - `constexpr` character pointers
  - Compile-time strings convertible to a compile-time string view
- To use format strings computed at runtime, use
  - `std::vformat()`
  - `std::vformat_to()`
- For `std::formatted_size()`, you can only use format strings known at compile time.

For example:

```
const char* fmt1 = "{:d}";                          // runtime format string
std::format(fmt1, 42);                              // compile-time ERROR
std::vformat(fmt1, std::make_format_args(42));      // OK

constexpr const char* fmt2 = "{:d}";                // compile-time format string
std::format(fmt2, 42);                              // OK
```

Using `fmt1` does not compile, because the passed argument is not a compile-time string and `std::format()` is used. However, using `fmt1` with `std::vformat()` works fine (but you have to convert all arguments with `std::make_format_args()`). Using `fmt2` does compile when passing it to `std::format()`, because it is initialized as a compile-time string.

If you want to use multiple arguments with `std::vformat()`, you have to pass them all to one call of `std::make_format_args()`:

```
const char* fmt3 = "{} {}";
std::vformat(fmt3, std::make_format_args(x, y))
```

If a format failure is detected at runtime, an exception of type `std::format_error` is thrown. This new standard exception type is derived from `std::runtime_error` and offers the usual API of standard exceptions to initialize them with a string and providing only a `what()` to get the error message.

For example:

```
try {
  const char* fmt4 = "{:s}";
  std::vformat(fmt4, std::make_format_args(42))  // throws std::format_error
}
catch (const std::format_error& e) {
  std::cerr << "FORMATTING EXCEPTION: " << e.what() << std::endl;
}
```

13.4 Internationalization

If `L` is specified for a format, the locale-specific notation is used:

- For `bool`, the locale strings on `std::numpunct::truename` and `std::numpunct::falsename` are used.

- For integral values, the locale-dependent thousands separator character is used.
- For floating-point values, the locale-dependent decimal point and thousands separator characters are used.
- For several notations of types in the chrono library (durations, timepoints, etc), their locale-specific formats are used.

To turn the locale specific notation on, you can pass a locale to `std::format()`. For example:

```
// initialize a locale for "German in Germany":
#ifdef _MSC_VER
std::locale locG{"deu_deu.1252"};
#else
std::locale locG{"de_DE"};
#endif

// use it for formatting:
std::format(locG, "{0} {0:L}", 1000.7)  // yields 1000.7 1.000,7
```

See *lib/formatgerman.cpp* for a complete example.

Note that the locale is only used, if you use the locale specifier `L`. Without, the default locale `"C"` is used, which uses the American formatting.

Alternatively, you can set the global locale and use the `L` specifier:

```
std::locale::global(locG);              // set German locale globally
std::format("{0} {0:L}", 1000.7)        // yields 1000.7 1.000,7
```

You might have to create your own locale (usually based on an existing locale with modified *facets*). For example:

lib/formatbool.cpp

```
#include <iostream>
#include <locale>
#include <format>

// define facet for German bool names:
class GermanBoolNames : public std::numpunct_byname<char> {
 public:
  GermanBoolNames (const std::string& name)
    : std::numpunct_byname<char>(name) {
  }
 protected:
  virtual std::string do_truename() const {
    return "wahr";
  }
  virtual std::string do_falsename() const {
    return "falsch";
  }
};
```

```
int main()
{
  // create locale with German bool names:
  std::locale locBool{std::cin.getloc(),
                      new GermanBoolNames{""}};

  // use locale to print Boolean values:
  std::cout << std::format(locBool, "{0} {0:L}\n", false);  //false falsch
}
```

The program has the following output:

```
false falsch
```

To print values with wide-character strings (which is especially an issue with Visual C++), both the format string and the arguments have to be wide-character strings. For example:

```
std::wstring city = L"K\u00F6ln";              // Köln
auto ws1 = std::format("{}", city);            // compile-time ERROR
std::wstring ws2 = std::format(L"{}", city);   // OK: ws2 is std::wstring
std::wcout << ws2 << '\n';                     // OK
```

Strings of types `char8_t` (UTF-8 characters), `char16_t`, and `char32_t` are not supported, yet.

13.5 User-Defined Formatted Output

The formatting library can defined formatting for user-defined types. What you need is a *formatter*, which is pretty straightforward to implement.

13.5.1 Basic Formatter API

A *formatter* is a specialization of the class template `std::formatter<>` for your type(s). Inside the formatter, two member functions have to be defined:

- `parse()` to implement how to parse the format string specifiers for your type
- `format()` to perform the actual formatting for an object/value of your type

Let us look at a first minimal example (we will improve it step by step) that specifies how to format an object/value that has a fixed value. Assume the type is defined like this (see *lib/format/always40.hpp*):

```
class Always40 {
 public:
  int getValue() const {
    return 40;
  }
};
```

For this type we can define a first formatter (which we should definitely improve) as follows:

lib/format/formatalways40.hpp

```
#include "always40.hpp"
#include <format>
#include <iostream>

template<>
struct std::formatter<Always40>
{
  // parse the format string for this type:
  constexpr auto parse(std::format_parse_context& ctx) {
    return ctx.begin();     // return position of } (hopefully there)
  }

  // format by always writing its value:
  auto format(const Always40& value, std::format_context& ctx) const {
    return std::format_to(ctx.out(), "{}", value.getValue());
  }
};
```

This is already good enough so that the following works:

```
Always40 val;
std::cout << std::format("Value: {}\n", val);
std::cout << std::format("Twice: {0} {0}\n", val);
```

The output would be:

```
Value: 40
Twice: 40 40
```

We define the formatter as specialization of type `std::formatter<>` for our type `Always40`:

```
template<>
struct std::formatter<Always40>
{
  ...
};
```

Because we only have public members we use `struct` instead of `class`.

Parsing the Format String

In `parse()` we implement the function to parse the format string:

```
// parse the format string for this type:
constexpr auto parse(std::format_parse_context& ctx) {
  return ctx.begin();     // return position of } (hopefully there)
}
```

The function takes a `std::format_parse_context`, provides an API to iterate over the remaining characters of the passed format string starting with the first character specific for the value to parse:

- If the format string is `"Value: {}"`
  `ctx.begin()` points to: `"}"`
- If the format string is `"Twice: {0} {0}"`
  `ctx.begin()` points to: `"} {0}"`
  when called for the first time

There is also a `ctx.end()`, which points to the end of the whole format string. If the format string is `"{1:7f} {}\n"`, `ctx.begin()` is the position of the `7` and `ctx.end()` is the end of the whole format string after the `\n`.

The task of `parse()` is to parse the specified format of the passed argument and then return the position of the end of our format specifier, which is the trailing `}`. In our implementation, we simply return the position of the first character we get, which will only work if the next character is really the `}` (this is the first thing we have to improve). Calling `std::format()` with any specified format character is an error:

```
Always40 val;
std::format("'{:7}", val)   // ERROR
```

Note that the `parse()` member function should be `constexpr` to support compile-time computing of the format string. This means that the code has to accept all restrictions of `constexpr` functions (which were relaxed with C++20).

However, you can see how this API allows us to parse whatever format we have specified for our type. Of course, we should follow the conventions of the standard specifiers to avoid programmer confusion.

Performing the Formatting

In `format()` we implement the function to format the passed value:

```
// format by always writing its value:
auto format(const Always40& value, std::format_context& ctx) const {
  return std::format_to(ctx.out(), "{}", value.getValue());
}
```

The function takes two parameters:

- Our value passed as argument to `std::format()` (or similar functions)
- A `std::format_context`, which provides the API to write the resulting characters of the formatting (according to the parsed format).

The most important function of the format context is `out()`, which yields an object you can pass to `std::format_to()` to write the actual characters of the formatting. The function has to return the new position for further output, which is returned by `std::format_to()`.

Note that it is currently not clear whether the `format()` member function of a formatter should be `const`. However, if you implement it you should better use `const` (unless the implementation modifies something). See `wg21.link/lwg3636` for details.

13.5.2 Improved Parsing

Let us improve the example we saw before. First, we should ensure that the parser deals with the format specifiers better:

- We should care about all character up to the closing }.
- We should throw an exception when illegal formatting arguments are specified.
- We should deal with valid formatting arguments (such as a specified field width).

Let us look at all of these topics by looking into an improved version of the previous formatter (this time dealing with a type that always has the value 41):

lib/format/formatalways41.hpp

```
#include "always41.hpp"
#include <format>

template<>
class std::formatter<Always41>
{
  int width = 0;  // specified width of the field
 public:
  // parse the format string for this type:
  constexpr auto parse(std::format_parse_context& ctx) {
    auto pos = ctx.begin();
    while (pos != ctx.end() && *pos != '}') {
      if (*pos < '0' || *pos > '9') {
        throw std::format_error{std::format("invalid format '{}'", *pos)};
      }
      width = width * 10 + *pos - '0';  // new digit for the width
      ++pos;
    }
    return pos;             // return position of }
  }

  // format by always writing its value:
  auto format(const Always41& value, std::format_context& ctx) const {
    return std::format_to(ctx.out(), "{:{}}", value.getValue(), width);
  }
};
```

Our formatter now has a member to store the specified field width:

```
template<>
class std::formatter<Always41>
{
  int width = 0;  // specified width of the field
  ...
```

```
};
```

The field width is initialized with `0`, but can be specified by the format string.

The parser now has a loop that processes all characters up to the trailing `}`:

```
constexpr auto parse(std::format_parse_context& ctx) {
  auto pos = ctx.begin();
  while (pos != ctx.end() && *pos != '}') {
    ...
    ++pos;
  }
  return pos;                 // return position of }
}
```

Note that the loop has to check for both whether there is still a character and whether it is the trailing `}`, because the programmer calling `std::format()` might have forgotten the trailing `}`.

Inside the loop we multiply the current width with the integral value of the digit character:

```
width = width * 10 + *pos - '0';   // new digit for the width
```

If the character is not a digit, we throw a `std::format` exception initialized with `std::format()`:

```
if (*pos < '0' || *pos > '9') {
  throw std::format_error{std::format("invalid format '{}'", *pos)};
}
```

Note that we cannot use `std::isdigit()` here, because it is not a functions we could call at compile time.

You can test the formatter as follows: *lib/format/always41.cpp*

The program has the following output:

```
41
Value: 41
Twice: 41 41
With width: '     41'
Format Error: invalid format 'f'
```

Note that the value is right-aligned because this is the default alignment for integral values.

13.5.3 Parsing with the Help of Standard Formatters

We can still improve the formatter implemented above:

- We can allow alignment specifiers.
- We can support fill characters.

Fortunately, we can get this by not implementing everything of the formatter ourselves; we can use other formatters to use the specifiers they already provide.

So, here is our final version of the formatter example (this time implemented for type `Always42`):

lib/format/formatalways42.hpp

```
#include "always42.hpp"
#include <format>
```

```
template<>
struct std::formatter<Always42> : std::formatter<int>
{
  auto format(const Always42& value, std::format_context& ctx) {
    // use standard int formatter for the integral value:
    return std::formatter<int>::format(value.getValue(), ctx);
  }
};
```

All we do here now is to create a string for the value that has to get formatted and pass it to the formatter for `int`'s. To let that formatter also do the parsing, we derive from `std::formatter<int>` and delegate the rest of the formatting to the `format()` member of this base class formatter.

We can test the formatter with the following program:

lib/format/always42.cpp

```
#include "always42.hpp"
#include "formatalways42.hpp"
#include <iostream>

int main()
{
  try {
    Always42 val;
    std::cout << val.getValue() << '\n';
    std::cout << std::format("Value: {}\n", val);
    std::cout << std::format("Twice: {0} {0}\n", val);
    std::cout << std::format("With width: '{:7}'\n", val);
    std::cout << std::format("With all:   '{:.^7}'\n", val);
  }
  catch (std::format_error& e) {
    std::cerr << "Format Error: " << e.what() << std::endl;
  }
}
```

The program has the following output:

```
42
Value: 42
Twice: 42 42
With width: '     42'
With all:   '..42...'
```

Note that the value is still right-aligned by default because that is the default alignment for `int`.

If your formatter only yields string literals, you can derive n the same way from the formatter for type `std::string_view`. This is expecially useful to implement a formatter for enumeration values:

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

template<>
struct std::formatter<Color> : std::formatter<std::string_view>
{
  auto format(Color c, format_context& ctx) {
    switch (c) {
      case Color::red:
        return std::formatter<std::string_view>::format("red", ctx);
      case Color::green:
        return std::formatter<std::string_view>::format("green", ctx);
      case Color::blue:
        return std::formatter<std::string_view>::format("blue", ctx);
    }
    return std::formatter<std::string_view>::format("<other Color>", ctx);
  }
};
```

Alternatively, you could implement the body of this `format()` member as follows:

```
std::string_view name = "<other Color>";
switch (c) {
  case Color::red:
    name = "red";
    break;
  case Color::green:
    name = "green";
    break;
  case Color::blue:
    name = "blue";
    break;
}
return formatter<string_view>::format(name, ctx);
```

You can use this formatter as follows:

```
std::cout << std::format("{} is {}\n", 42, color::red); // prints 42 is red
```

If you have to create more complicated strings, you can always create a `std::string` and use the formatter for strings.

Again note that it is currently not clear whether the `format()` member function of a formatter should be `const` (see wg21.link/lwg3636 for details). However, if you declare your `format()` function as `const` you cannot call a standard `format()` function that is not declared `const`. So, for the moment you might better not use `const`.

Open

******* This chapter/section is at work *******

13.6 Afternotes

Formatted output was first proposed by Victor Zverovich and Lee Howes in http://wg21.link/p0645r0. The finally accepted wording was formulated by Victor Zverovich in http://wg21.link/p0645r10.

After C++20 was standardized, checking format strings at compile time was accepted as a fix against C++20 as formulated by Victor Zverovich in http://wg21.link/p2216r3.

This page is intentionally left blank

Chapter 14

Dates and Time Zones for <chrono>

C++11 introduced the chrono library with basic support for durations and time points. You could specify and deal with durations of different units and time points of different clocks. However, there was no support yet for high-level durations and time points such as dates (days, months and years), weekdays, and dealing with different time zones.

C++20 extends the existing chrono library with support for dates, time zones, and several other features. This extension is described in this chapter.[1]

14.1 Overview by Example

Before we go into details, let us look at some motivating examples.

14.1.1 Schedule a Meeting on the 5th of Every Month

Consider a program where we want to to iterate over all months in a year to schedule a meeting on the 5th announced in different time zones. The program can be written as follows:

lib/chrono1.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono;       // shortcut for std::chrono
  using namespace std::literals;     // for h, min, y suffixes
```

[1] Many thanks to Howard Hinnant, the author of this library, for his incredible support in writing this chapter. He provided fast and elaborated answers, several code examples, and even some particular wording (with permission not marked as a quote to keep the chapter easy to read).

```
  // for each 5th of all months of 2021:
  chr::year_month_day first = 2021y / 1 / 5;
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << '\n';

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};                  // system/UTC time
    std::cout << "  UTC:    " << tp << '\n';

    // print date/time with specific time zones:
    chr::zoned_time timeBerlin{"Europe/Berlin", tp};          // Berlin time
    chr::zoned_time timeUsWest{"America/Los_Angeles", tp};    // L.A. time
    std::cout << "  Berlin: " << timeBerlin << '\n';
    std::cout << "  L.A.:   " << timeUsWest << '\n';
  }
}
```

The program has the following output:

```
2021-01-05
  UTC:    2021-01-05 18:30:00
  Berlin: 2021-01-05 19:30:00 CET
  L.A.:   2021-01-05 10:30:00 PST
2021-02-05
  UTC:    2021-02-05 18:30:00
  Berlin: 2021-02-05 19:30:00 CET
  L.A.:   2021-02-05 10:30:00 PST
2021-03-05
  UTC:    2021-03-05 18:30:00
  Berlin: 2021-03-05 19:30:00 CET
  L.A.:   2021-03-05 10:30:00 PST
2021-04-05
  UTC:    2021-04-05 18:30:00
  Berlin: 2021-04-05 20:30:00 CEST
  L.A.:   2021-04-05 11:30:00 PDT
2021-05-05
  UTC:    2021-05-05 18:30:00
  Berlin: 2021-05-05 20:30:00 CEST
  L.A.:   2021-05-05 11:30:00 PDT
...
2021-10-05
  UTC:    2021-10-05 18:30:00
  Berlin: 2021-10-05 20:30:00 CEST
  L.A.:   2021-10-05 11:30:00 PDT
2021-11-05
  UTC:    2021-11-05 18:30:00
  Berlin: 2021-11-05 19:30:00 CET
  L.A.:   2021-11-05 11:30:00 PDT
2021-12-05
```

```
UTC:    2021-12-05 18:30:00
Berlin: 2021-12-05 19:30:00 CET
L.A.:   2021-12-05 10:30:00 PST
```

Let us go step by step through this program.

Namespace Declarations

We start with namespace and `using` declarations to make the use of the chrono library a bit more convenient:

- First we introduce `chr::` as a shortcut for the standard namespace of the chrono library `std::chrono`:

  ```
  namespace chr = std::chrono;        // shortcut for std::chrono
  ```

  To make examples more readable, I will from time to time just use `chr::` instead of `std::chrono::`.
- Then, we ensure that we can use literal suffixes such as `y`, `min`, `h`, and `y` (the latter is new since C++20).

  ```
  using namespace std::literals;      // for min, h, y suffixes
  ```

To avoid any qualification you could use a `using` declaration instead:

```
using namespace std::chrono;        // skip chrono namespace qualification
```

However, as usual, you should limit the scope of such a using declaration to avoid unwanted side effects.

Date Type `year_month_day`

The first real statement is the initialization of `first` as an object of type `std::chrono::year_month_day`.

```
chr::year_month_day first = 2021y / 1 / 5;
```

The type `year_month_day` is a calendrical type providing attributes for all three fields of a date so that it is easy to deal with the year, the month, and the day of a particular date.

Because we want to iterate over all fifth days of each month in the current year, we initialize the object with January 5, 2021, passing first a year and then using operator **`/`** to combine it with a value for the month and a value for the day.

This way of initialization for date types is type safe and requires only specifying the type of the first operand. In fact, what happens here is as follows:

- First, we specify the year as an object of type `std::chrono::year`.
  Here, we use the new standard literal **`y`**:

  ```
  2021y
  ```

  To have this literal available, we have to provide one of the following `using` declarations:

  ```
  using std::literals;              // enable all standard literals
  using std::chrono::literals;      // enable all standard chrono literals
  using namespace std::chrono;      // enable all standard chrono literals
  using namespace std;              // enable all standard literals
  ```

  Without these literals we would need something like the following:

  ```
  std::chrono::year{2021}
  ```

- Then, we use operator / to combine a std::chrono::year with an integral value, which creates an object of type std::chrono::year_month.

  Because the first operand is a year it is clear that the second operand must be a month. You cannot specify a day there.
- Finally, we use operator / again to combine std::chrono::year_month object with an integral value to create a std::chrono::year_month_day.

Because the operator already yields the right type, we could also declare first just with auto. Without the namespace declarations we could and would have to write:

```
auto first = std::chrono::year{2021} / 1 / 5;
```

With the chrono literals available, we could simple write:

```
auto first = 2021y/1/5;
```

Other Ways to Initialize Date Types

There are other ways to initialize a date type like year_month_day:

```
auto d1 = std::chrono::years{2021}/1/31;      // January 31, 2021
auto d2 = std::chrono:month{1}/31/2021;       // January 31, 2021
auto d3 = std::chrono:day{31}/1/2021;         // January 31, 2021
```

That is, the type of the first argument for operator / specifies how to interpret the other ones.

With the chrono literals available, we could simple write:

```
using namespace std::literals;
auto d4 = 2021y/1/31;                         // January 31, 2021
auto d5 = 31d/1/2021;                         // January 31, 2021
```

There is no standard suffix for a month, but we have predefined standard objects:

```
auto d6 = std::chrono::January / 31 / 2021; // January 31, 2021
```

With the corresponding using declaration we could even write just the following:

```
using namespace std::chrono;
auto d6 = January/31/2021;                    // January 31, 2021
```

In all cases we initialize an object of type std::chrono::year_month_day.

New Duration Types

In our example, we then call a loop to iterate over all months of the year:

```
for (auto d = first; d.year() == first.year(); d += months{1}) {
  ...
}
```

The type std::chrono::months is a new duration type representing a month. You can use it for all calendrical types to deal with the specific month of a date such as adding one month as done here:

```
std::chrono::year_month_day d = ... ;
d += months{1};                               // add one month
```

However, note that when using it for ordinary durations and timepoints, the type `months` represents the ***average duration*** of a month, which is `30.436875` days. So, with ordinary timepoints you should use `months` and `years` with care.

Note that the default output format of `year_month_weekday` uses slashes instead of dashes as separator (only `year_month_day` uses hyphens because using dashes that is *the* establishes output format for dates of this format).

Output Operators for All Chrono Types

Inside the loop, we print the current date:

```
std::cout << d << '\n';
```

Since C++20, there is an output operator defined for more or less all possible chrono types.

```
std::chrono::year_month_day d = ...;
std::cout << "d: " << d << '\n';
```

This makes it easy to just print any chrono value. However, the output does not always fit specific needs. For `year_month_day` the output format is what we as programmers would expect for such a type: *year-month-day*. For example:

```
2021-01-05
```

The other default output formats are documented here. For user defined output the chrono library supports the the new library for formatted output, which is described here.

Combining Dates and Times

Next we combine the days of the loop with a specific time of the day:

```
auto tp{sys_days{d} + 18h + 30min};
```

For combined dates and times we have to use timepoints. However, there are new convenience types defined for that. So we first convert the `year_month_day` value to a `time_point<>` object:

```
std::chrono::year_month_day d = ...;
std::chrono::sys_days{d}    // convert to a time_point
```

The type `std::chrono::sys_days` is a a new shortcut for system timepoints with the granularity of days. It is equivalent to: `time_point<system_clock, days>`.

By adding some durations (18 hours and 30 minutes), we compute a new value, which as usual in the chrono library, has a type with a granularity that is good enough for the result of the computation. Because we combine days with hours and minutes the resulting type is a system timepoint with the granularity of minutes. However, we do not have to know the type. Just using `auto` works fine.

If we want to specify the type of `tp` we could use two other alias types:

- A system timepoint type for any specified duration:
  ```
  std::chrono::sys_time<std::chrono::minutes> tp;
  ```
- A system timepoint type for seconds:
  ```
  std::chrono::sys_seconds tp;
  ```

In all these cases the default output operator prints the timepoint with seconds:

```
2021-02-01 18:30:00
```

Note that when dealing with system time, the output by default is UTC.

A more fine grained timepoint would also use whatever is necessary to output the exact value (such as milliseconds). See later.

Using Time Zones

Finally, we print the timepoint using different time zones, one for Berlin and one for Los Angeles:

```
chr::zoned_time timeBerlin{"Europe/Berlin", tp};         // Berlin time
chr::zoned_time timeUsWest{"America/Los_Angeles", tp};   // L.A. time
std::cout << "  Berlin: " << timeBerlin << '\n';
std::cout << "  L.A.:   " << timeUsWest << '\n';
```

A `std::chrono::zoned_time` object is an object representing a timepoint and taking a time zone into account. To specify the time zone we have to use the IANA timezone database, which is usually based on cities, not on the usual timezone codes. Using the default output operator of these objects adds the corresponding time zone code, whether it is in "winter time:"

```
UTC:    2021-01-05 18:30:00
Berlin: 2021-01-05 19:30:00 CET
L.A.:   2021-01-05 10:30:00 PST
```

or whether it is in "summer time:"

```
UTC:    2021-07-05 18:30:00
Berlin: 2021-07-05 20:30:00 CEST
L.A.:   2021-07-05 11:30:00 PDT
```

Sometimes some time zones have summer time while others do not (like at the beginning of November in the US only):

```
UTC:    2021-11-05 18:30:00
Berlin: 2021-11-05 19:30:00 CET
L.A.:   2021-11-05 11:30:00 PDT
```

14.1.2 Schedule a Meeting Every First Monday

Let us modify the first example program a bit as follows:

- Iterate over all months of the **current year**
- Schedule the meeting **each first Monday** of a month
- Schedule the meeting at 18:30 of our **local time**

Consider a program where we want to to iterate over all first Mondays of each month in a year and schedule a meeting in different time zones. The program can now be written as follows:

lib/chrono2.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono;       // shortcut for std::chrono
  using namespace std::literals;     // for min, h, y suffixes

  // initialize today as current local date:
  auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
  chr::year_month_day today{chr::floor<chr::days>(localNow)};
  std::cout << "today: " << today << '\n';

  // for each first Monday of all months of the current year:
  auto first = today.year() / 1 / chr::Monday[1];
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << '\n';

    // init and print 18:30 local time of those days:
    auto tp{chr::local_days{d} + 18h + 30min};                    // no timezone

    // apply this local time to the current time zone:
    chr::zoned_time timeLocal{chr::current_zone(), tp};           // local time
    std::cout << "  local:  " << timeLocal << '\n';

    // print out date with other time zones:
    chr::zoned_time timeUTC{"UTC", timeLocal};                    // UTC time
    chr::zoned_time timeUsWest{"America/Los_Angeles", timeLocal}; // L.A. time
    std::cout << "  UTC:    " << timeUTC << '\n';
    std::cout << "  L.A.:   " << timeUsWest << '\n';
  }
}
```

Starting the program in Europe on March 29, 2021, results in the following output:

```
today: 2021-03-29
2021/Jan/Mon[1]
  local:  2021-01-04 18:30:00 CET
  UTC:    2021-01-04 17:30:00 UTC
  L.A.:   2021-01-04 09:30:00 PST
2021/Feb/Mon[1]
  local:  2021-02-01 18:30:00 CET
  UTC:    2021-02-01 17:30:00 UTC
  L.A.:   2021-02-01 09:30:00 PST
2021/Mar/Mon[1]
```

```
  local:  2021-03-01 18:30:00 CET
  UTC:    2021-03-01 17:30:00 UTC
  L.A.:   2021-03-01 09:30:00 PST
2021/Apr/Mon[1]
  local:  2021-04-05 18:30:00 CEST
  UTC:    2021-04-05 16:30:00 UTC
  L.A.:   2021-04-05 09:30:00 PDT
2021/May/Mon[1]
  local:  2021-05-03 18:30:00 CEST
  UTC:    2021-05-03 16:30:00 UTC
  L.A.:   2021-05-03 09:30:00 PDT
...
2021/Oct/Mon[1]
  local:  2021-10-04 18:30:00 CEST
  UTC:    2021-10-04 16:30:00 UTC
  L.A.:   2021-10-04 09:30:00 PDT
2021/Nov/Mon[1]
  local:  2021-11-01 18:30:00 CET
  UTC:    2021-11-01 17:30:00 UTC
  L.A.:   2021-11-01 10:30:00 PDT
2021/Dec/Mon[1]
  local:  2021-12-06 18:30:00 CET
  UTC:    2021-12-06 17:30:00 UTC
  L.A.:   2021-12-06 09:30:00 PST
```

Look at the output of October and November: In Los Angeles, the meeting is scheduled now at different times although the same time zone PDT is used. That happens, because the source for the meeting time (central Europe) changed from summer to winter/standard time.

Again, let us go step by step through the modifications of this program.

Dealing with Today

The first new statement is the initialization of `today` as an object of type `year_month_day`:

```
auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
chr::year_month_day today = chr::floor<chr::days>(localNow);
```

Support for `std::chrono::system_clock::now()` was already provided since C++11 and yields a `std::chrono::time_point<>` with the granularity of the system clock. However, the system clocks uses UTC (since C++20, the `system_clock` is guaranteed to use *Unix time*, which is based on UTC). Therefore, we first have to adjust the current UTC time and date to the time and date of the current/local time zone, which `current_zone()->to_local()` does. Otherwise, our local date might not fit the UTC date (because we already passed midnight but UTC did, or the other way around).

Using it directly to initialize a `year_month_day` value does not compile, because we would "narrow" the value (lose the hours, minutes, seconds, milliseconds, ... parts of the value). By using a convenience function such as `floor()` (available since C++17), we round the value down according to our requested granularity.

In case you need the current date according UTC, the following would be enough:

```
chr::year_month_day today = chr::floor<chr::days>(chr::system_clock::now());
```

Date Types for Weekdays

Because we want to iterate over all first Mondays of each month in the current year, we initialize an object of the very special calendrical type `std::chrono::month_year_weekday` and initialize it with the first Monday of January of the current year:

```
auto first = today.year() / 1 / chr::Monday[1];
```

Again we use `operator/` to combine different date fields. However, this time types for weekdays come into play:

- First, we call `today.year() / 1` to combine a `std::chrono::year` with an integral value to create a `std::chrono::year_month`.
- Then, we use `Monday`, a standard object of type `std::chrono::weekday` with `operator[]` to create an object `std::chrono::weekday_indexed` representing the *n*th weekday.
- Finally, `operator/` is used to combine the `std::chrono::month_year` with the `std::chrono::weekday_indexed` object, which creates an object of type `std::chrono::year_month_weekday`.

Therefore, a fully specified declarations would look as follows:

```
std::chrono::year_month_weekday first = today.year() / 1 / std::chrono::Monday[1];
```

Note that the default output format of `year_month_weekday` uses slashes instead of dashes as separator (only `year_month_day` uses hyphens in its default output format). For example:

```
2021/Jan/Mon[1]
```

Local Dates and Times

Again, we combine the days we iterate over with a specific time of day. However, this time we convert each day to type `std::chrono::local_days` first:

```
auto tp{local_days{d} + 18h + 30min};
```

The type `std::chrono::local_days` is a shortcut of `time_point<local_t, days>`. Here the pseudo clock `std::chrono::local_t` is used, which means that we have a *local timepoint*, a time point with no associated time zone (not even UTC) yet.

The next statement combines the local timepoint with the current timezone so that we get a time-zone specific point of time of type `std::chrono::zoned_time<>`.

```
chr::zoned_time timeLocal{chr::current_zone(), tp};   // local time
```

The difference between time points and zoned times is subtle:

- A *timepoint **may*** be associated with a defined epoch so that it defines a unique point in time. However, it might also be associated with an undefined or pseudo epoch, for which the meaning is not clear yet.
- A *zoned time **is*** associated with a time zone so that the epoch finally has a defined meaning and it always represents a unique point in time.

The default output operators demonstrate the different types:

```
auto tpLocal{chr::local_days{d} + 18h + 30min};            // local timepoint
std::cout << "timepoint: " << tpLocal << '\n';

chr::zoned_time timeLocal{chr::current_zone(), tpLocal};   // apply to local timezone
```

```
std::cout << "zonetime:  " << timeLocal << '\n';
```

This code outputs, for example:

```
timepoint: 2021-01-04 18:30:00
zonetime:  2021-01-04 18:30:00 CET
```

You see that the timepoint is printed without a time zone while the zoned time has it. However, both outputs print the same time because we apply this time to our local timezone.

However, the following code demonstrates why the difference is a bit tricky:

```
auto tpSys{chr::sys_days{d} + 18h + 30min};              // system timepoint
std::cout << "timepoint: " << tpSys << '\n';

chr::zoned_time timeSys{chr::current_zone(), tpSys};     // convert to local timezone
std::cout << "zonetime:  " << timeSys << '\n';
```

This code outputs for the same day:

```
timepoint: 2021-01-04 18:30:00
zonetime:  2021-01-04 19:30:00 CET
```

Again the default output operator of the system timepoint prints no time zone. However, the timepoint is already associated with the system clock, which means that it already associates the time with `UTC`. Combining this timepoint with a different timezone *converts* the time to the other timezone, so that we get a difference (here one hour because I was one hour ahead of UTC).

Using Zoned Times

At the end of the loop, we convert the time-zone specific points in time to other time zones. For this, we simply initialize other `zoned_time`'s with the local zoned time and a different time zone:

```
chr::zoned_time timeLocal{chr::current_zone(), tp};              // local time
chr::zoned_time timeUTC{"UTC", timeLocal};                       // UTC time
chr::zoned_time timeUsWest{"America/Los_Angeles", timeLocal};    // L.A. time
```

So the output for a particular point in time caused by the following code:

```
std::cout << "  local:  " << timeLocal << '\n';
...
std::cout << "  UTC:    " << timeUTC << '\n';
std::cout << "  L.A.:   " << timeUsWest << '\n';
```

looks as follows:

```
local:  2021-01-04 18:30:00 CET
UTC:    2021-01-04 17:30:00 UTC
L.A.:   2021-01-04 09:30:00 PST
```

14.2 Basic Chrono Concepts and Terminology

The chrono library was designed to be able to deal with the fact that timers and clocks might be different on different systems and improve over time in precision. To avoid having to introduce a new time type every 10 years or so, the basic goal established with C++11 was to provide a precision-neutral concept by separating duration and point of time ("timepoint"). C++20 extends these basic concepts with support for dates and time zones and a few other extensions.

As a result, the core of the chrono library consists of the following types or concepts:

- A ***duration*** of time is defined as a specific number of ticks over a time unit. One example is a duration such as "3 minutes" (3 ticks of a "minute"). Other examples are "42 milliseconds" or "86,400 seconds," which represents the duration of 1 day. This concept also allows specifying something like "1.5 times a third of a second," where 1.5 is the number of ticks and "a third of a second" the time unit used.
- A ***timepoint*** is defined as combination of a duration and a beginning of time (the so-called ***epoch***).

  A typical example is a ***system timepoint*** that represents New Year's Midnight 2000. Because the system epoch is specified as Unix/POSIX birth, the timepoint would be defined as "946,684,800 seconds since January 1, 1970." (or 946,684,822 seconds when taking leap seconds into account, which some timepoints do).

  However, the epoch might also be an unspecified or pseudo epoch. For example, a ***local timepoint*** is associated with whatever local time we have. It still needs a certain value for the epoch, but which point in time it represents becomes clear when we apply this timepoint to a specific timezone. A typical example is Midnight on New Year's Eve. The celebration and fireworks does not start start simultaneously; when it starts depends on the timezone we are in.
- A ***clock*** is the object that defines the epoch of a timepoint. Thus, different clocks have different epochs. Each timepoint is parameterized by a clock.

  C++11 introduced two basic clocks (a `system_clock` to deal with the system time and a `steady_clock` for measurements and timers that should not be influenced by changes of the system clock). C++20 adds new clocks to deal with UTC timepoints (supporting leap seconds), GPS timepoints, TAI (international atomic time) timepoints, and timepoints of the filesystem,

  To deal with *local timepoints*, C++20 also adds a pseudo clock `local_t`, which is not bound to a specific epoch/origin.

  Operations dealing with multiple timepoints, such as processing the duration/difference between two timepoints, usually require using the same epoch/clock. However, conversions between different clocks are possible.

  A clock (if not local) provides a convenience function `now()` to yield the current point in time.
- A ***calendrical*** type (introduced with C++20) allows us to deal with the attributes of calendars using the common terminology of days, months, and years. These types can be used to represent a single attribute of a date (`day`, `month`, `year`, and `weekday`) and combinations of them (such as `year_month` or `year_month_day` for a full date).

  Different symbols such as `Wednesday`, `November`, and `last` allow us to use common terminology for partial and full dates, such as "last Wednesday of November."

  Fully specified calendrical dates (having a year, month, and day or specific weekday of the month) can be converted to or from timepoints using the system clock or the pseudo clock for the local time.

- A ***time zone*** (introduced with C++20) allows us to deal with the fact that simultaneous events result in different times when different time zones come into play. If we meet online at 18:30 UTC, the local time of the meeting would be significant later in Asia, but significant earlier in America.

  Thus, time zones give timepoints a (different) meaning by applying or converting it to a different local time.
- A ***zoned time*** (introduced with C++20) is the combination of a timepoint with a time zone. It can be used to apply a local timepoint to a specific time zone (which might be the "current" time zone) or convert timepoints to different time zones.

  A zoned time can be seen as a date and time object, which is the combination of an epoch (the origin of a time point), a duration (the distance from the origin), and a time zone (to adjust the resulting time).

For all of these concepts and types C++20 adds support for output (even formatted) and parsing. That way you can decide whether you print date/time values in formats like:

```
Nov/24/2011
24.11.2011
2011-11-24 16:30:00 UTC
Thursday, November 11, 2011
```

14.3 Basic Chrono Extensions with C++20

Before we discuss how to use it the chrono library in detail, let us introduce all basic types and symbols.

14.3.1 Duration Types

C++20 introduces a additional duration types for days, weeks, months, and years. Table *Standard duration types since C++20* lists all duration types C++ has now, together with the standard literal suffix you can use to create a value and the default output suffix used when printing values.

Type	Defined as	Literal	Output Suffix
nanoseconds	1/1,000,000,000 Seconds	**ns**	**ns**
microseconds	1,000 Nanoseconds	**us**	**µs** or **us**
milliseconds	1,000 Microseconds	**ms**	**ms**
seconds	1,000 Milliseconds	**s**	**s**
minutes	60 Seconds	**min**	**min**
hours	60 Minutes	**h**	**h**
days	24 Hours	**d**	**d**
weeks	7 Days		**[604800]s**
months	30.436875 Days		**[2629746]s**
years	365.2425 Days	**y**	**[31556952]s**

Table 14.1. Standard duration types since C++20

Be careful when using `std::chrono::months` and `std::chrono::years`. `months` and `years` represent the ***average*** duration of a month or year, which is a fractional day. The duration of an average year is computed by taking leap years into account:

- Every 4 years we have one more day (366 instead of 365 days)
- However, every 100 years we do not have one more day
- However, every 400 years we have one more day again

So the value is $400 * 365 + 100 - 4 + 1$ divided by 400. The duration of an average month is a 12th of it.

14.3.2 Clocks

C++20 introduces a couple of new clocks (remember, clocks define the origin/epoch of a timepoint).

Table *Standard clock types since C++20* describes their names and meanings.

Type	Meaning	Epoch
`system_clock`	Associated with the lock of the system (C++11)	UTC time
`utc_clock`	Clock for UTC time values	UTC time
`gps_clock`	Clock for GPS values	GPS time
`tai_clock`	Clock for international atomic time values	TAI time
`file_clock`	Clock for timepoints of the filesystem library	impl.spec.
`local_t`	Pseudo clock for *local timepoints*	open
`steady_clock`	Clock for measurements (C++11)	impl.spec.

Table 14.2. Standard clock types since C++20

The column **Epoch** specifies whether the clock specifies a unique point in time so that you always have a specific UTC time defined. This requires a stable specified epoch. For the `file_clock` the epoch is system specific but will be stable across multiple runs of a program. The epoch of the `steady_clock` might change from one run of your application to the next (e.g., when the system rebooted). For the pseudo clock `local_t`, the epoch is interpreted as "local time," which means you have to combine it with a timezone to know which point in time it represents.

The C++20 standard also provides a `high_resolution_clock`. However, its use can introduce subtle issues when porting code from one platform to another. In practice, `high_resolution_clock` is an alias for either `system_clock` or `steady_clock`, which means that the clock sometimes is steady and sometimes is not and that this clock might or might not support conversions to other clocks or `time_t`. As the `high_resolution_clock` is on no platform finer than the `steady_clock`, you should use the `steady_clock` instead.[2]

We discuss later in detail

- The difference of the clocks in detail
- Conversions between clocks
- Which way clocks deal with leap seconds

[2] Thanks to Howard Hinnant for pointing that out.

14.3.3 Timepoint Types

C++20 introduces a couple of new types for timepoints. Based on the general definition available since C++11:

```
template<typename Clock, typename Duration = typename Clock::duration>
class time_point;
```

the new types provide a more convenient way of using timepoints with the different clocks.

Table *Standard timepoint types since C++20* describes the names and meanings of the convenient timepoint types.

Type	Meaning	Defined as
local_time<*Dur***>**	Local timepoint	time_point<*LocalTime*, *Dur*>
local_seconds	Local timepoint in seconds	time_point<*LocalTime*, seconds>
local_days	Local timepoint in days	time_point<*LocalTime*, days>
sys_time<*Dur***>**	System timepoint	time_point<system_clock, *Dur*>
sys_seconds	System timepoint in seconds	time_point<system_clock, seconds>
sys_days	System timepoint in days	time_point<system_clock, days>
utc_time<*Dur***>**	UTC timepoint	time_point<utc_clock, *Dur*>
utc_seconds	UTC timepoint in seconds	time_point<utc_clock, seconds>
tai_time<*Dur***>**	TAI timepoint	time_point<tai_clock, *Dur*>
tai_seconds	TAI timepoint in seconds	time_point<tai_clock, seconds>
gps_time<*Dur***>**	GPS timepoint	time_point<gps_clock, *Dur*>
gps_seconds	GPS timepoint in seconds	time_point<gps_clock, seconds>
file_time<*Dur***>**	Filesystem timepoint	time_point<file_clock, *Dur*>

Table 14.3. Standard timepoint types since C++20

For each standard clock (except the steady clock), we have a corresponding `_time<>` type, which allows us to declare objects that represent a date and the time of it. In addition, a `_seconds` type allows us to define date/time objects of the corresponding type with the granularity of seconds. For system and local time, a `_days` type allows us to define date/time objects of the corresponding type with the granularity of days:

```
std::chrono::sys_days d;    // shortcut for time_point<system_clock, days>
```

Please note that objects of this type represent still one timepoint, although the type unfortunately has a plural name. For example, `sys_days` means a single day defined as a system timepoint (the name comes from "system time point with granularity days").

14.3.4 Calendrical Types

As extension to timepoint types, C++20 introduces types for the civil (Gregorian) calendar to the chrono library.

While time points are specified as durations from an epoch, calendrical types have distinct and combined types and values for years, months, weekdays, and days of a month. Both are useful to use:

- Timepoint types are great as long as we only compute with seconds, hours and days (such as doing something on each day of a year).
- Calendrical types are great for date arithmetic where the fact that months and years have a different number of days comes into account. In addition, you can deal with something like "the third Monday of the month" or "the last day of the month."

Table *Standard calendrical types* lists the new calendrical types together with their default output format.

Type	Meaning	Output Format
day	Day	05
month	Month	Feb
year	Year	1999
weekday	Weekday	Mon
weekday_indexed	*n*th weekday	Mon[2]
weekday_last	Last weekday	Mon[last]
month_day	Day of a month	Feb/05
month_day_last	Last day of a month	Feb/last
month_weekday	*n*th weekday of a month	Feb/Mon[2]
month_weekday_last	Last weekday of a month	Feb/Mon[last]
year_month	Month of a year	1999/Feb
year_month_day	A full date (day of a month of a year)	1999-02-05
year_month_day_last	Last day of a month of a year	1999/Feb/last
year_month_weekday	*n*th weekday of a month of a year	1999/Feb/Mon[2]
year_month_weekday_last	Last weekday of a month of a year	1999/Feb/Mon[last]

Table 14.4. Standard calendrical types

Remember that these types names do not imply in which order the elements of the date can be passed for initialization or are written to formatted output. For example, type `std::chrono::year_month_day` can be used as follows:

```
using namespace std::chrono;

year_month_day d = January/31/2021;          // January 31, 2021
std::cout << std::format("{:%D}", d);        // writes 01/31/21
```

Calendrical types such as `year_month_day` allow us to compute precisely the date of the same day next month:

```
std::chrono::year_month_day start = ...;             // 2021/2/5
auto end = start + std::chrono::months{1};           // 2021/3/5
```

If you used timepoints such as `sys_days` the corresponding code would not really work, because it uses the ***average*** duration of a month:

```
std::chrono::sys_days start = ...;                   // 2021/2/5
auto end = start + std::chrono::months{1};           // 2021/3/7 10:29:06
```

Note that in this case `end` has a different type because with `months` fractional days come into play.

When adding 4 weeks or 28 days timepoints types are better, because for them this is a simple arithmetic operation and does not have to take the different length of months or years into account:

```
std::chrono::sys_days start = ... ;                    // 2021/1/5
auto end = start + std::chrono::weeks{4};              // 2021/2/2
```

Details of using `months` and `years` are discussed later.

As you can see, there are specific types to deal with weekdays and the *n*th and last weekday within a month. This allows us to iterate over all second Mondays or jump to the last day of the next month:

```
std::chrono::year_month_day_last start = ... ;         // 2021/2/28
auto end = start + std::chrono::months{1};             // 2021/3/31
```

Every type has a default output format, which is a fixed English sequence of characters. For other formats, use the formatted chrono output. Please note that only `year_month_day` uses dashes as separator in its default output format. All other types separate their tokens by default with a slash.

To deal with the values of the calendrical types a couple of calendrical constants are defined:

- `std::chrono::`**`last`**
  to specify the last day/weekday of a month. The constant has type `std::chrono::`**`last_spec`**.
- `std::chrono::`**`Sunday`**, `std::chrono::`**`Monday`**, ... `std::chrono::`**`Saturday`**
  to specify a weekday (Sunday has value 0, Saturday has value 6).
  These constants have type `std::chrono::`**`weekday`**.
- `std::chrono::`**`January`**, `std::chrono::`**`February`**, ... `std::chrono::`**`December`**
  to specify a month (January has value 1, December has value 12).
  These constants have type `std::chrono::`**`month`**.

Restricted Operations for Calendrical Types

The calendrical types were designed to detect at compile-time when operations are not useful or do not have the best performance. As a result, some "obvious" operations do not compile as table *Standard operations for calendrical types* lists:

- Day and month arithmetic depends on the year. You cannot add days/months or compute the difference of all month types that do not have a year.
- Day arithmetic for fully specified dates takes a bit of time to deal with the different length of months and leap years. You can add/subtract days or compute the difference of these types.
- Because chrono makes no assumption about the first day of a week, you cannot get an order between weekdays. When comparing types with weekdays, only `operator==` and `operator!=` are supported.

Weekday arithmetic is modulo 7 so that it does not really matter which day is the first day of a week. You can compute the difference of any two weekdays and the result is always a value between 0 and 6. Adding the difference to the first weekday will always be the second weekday. For example:

```
std::cout << chr::Friday - chr::Tuesday << '\n';       // 3d (Tuesday thru Friday)
std::cout << chr::Tuesday - chr::Friday << '\n';       // 4d (Friday thru Tuesday)

auto d1 = chr::February / 25 / 2021;
```

Type	++/--	add/subtract	- (diff.)	==	</<=>
day	yes	days	yes	yes	yes
month	yes	months, years	yes	yes	yes
year	yes	years	yes	yes	yes
weekday	yes	days	yes	yes	-
weekday_indexed	-	-	-	yes	-
weekday_last	-	-	-	yes	-
month_day	-	-	-	yes	yes
month_day_last	-	-	-	yes	yes
month_weekday	-	-	-	yes	-
month_weekday_last	-	-	-	yes	-
year_month	-	months, years	yes	yes	yes
year_month_day	-	months, years	-	yes	yes
year_month_day_last	-	months, years	-	yes	yes
year_month_weekday	-	months, years	-	yes	-
year_month_weekday_last	-	months, years	-	yes	-

Table 14.5. Standard operations for calendrical types

```
auto d2 = chr::March / 3 / 2021;
std::cout << chr::sys_days{d1} - chr::sys_days{d2} << '\n';   // -6d (date diff)
std::cout << chr::weekday(d1) - chr::weekday(d2) << '\n';     // 3d  (weekday diff)
```

That way you can always easily compute something like the difference to "the next Monday" as "Monday minus the current weekday:"

```
d1 =  chr::sys_days{d1} + (chr::Monday - chr::weekday(d1));   // set d1 to next Monday
```

Note that if `d1` is a calendrical date type you first have to convert it to type `std::chrono::sys_days` so that day arithmetic is supported (if this happens often, you better declare `d1` with this type).

The same way month arithmetic is modulo 12 so that the next month after December is January. If a year is part of the type, it is adjusted accordingly:

```
auto m = chr::December;
std::cout << m + chr::months{10} << '\n';         // Oct
std::cout << 2021y/m + chr::months{10} << '\n';   // 2022/Oct
```

14.3.5 Time Type `hh_mm_ss`

According to the calendrical types, C++20 introduces a new time type, which converts a timepoint or duration to a kind of data structure with corresponding time fields. This type is most useful to deal with the different attributes of a time specified as duration. It allows us to split times into its attributes and serves as a formatting aid.

Table `hh_mm_ss` *members* describes the names and meanings ob the `hh_mm_ss` members.

Member	Meaning
hours()	Hour value
minutes()	Minutes value
seconds()	Seconds value
subseconds()	Value for partial seconds with appropriate granularity
is_negative()	True if the value is negative
to_duration()	Conversion to value with corresponding precision
precision	duration type of the subseconds
operator precision()	Conversion to value with corresponding precision
fractional_width	precision of the subseconds (static member)

Table 14.6. `hh_mm_ss` *members*

For example, you can check for a specific hour or pass the hour and minute as integral values to another function:

```
auto dur = measure();                  // process and yield some duration
std::chrono::hh_mm_ss hms{dur};        // convert to data structure for attributes
process(hms.hours(), hms.minutes());   // pass hours and minutes
```

If you have a timepoint, you have to convert it to a duration first. For this, you usually just compute the difference of the timepoint with midnight of the day (computed by rounding it down to the granularity of days). For example:

```
auto tp = getStartTime();              // process and yield some timepoint
// convert time to data structure for attributes:
std::chrono::hh_mm_ss hms{tp - std::chrono::floor<std::chrono:days>(tp)};
process(hms.hours(), hms.minutes());   // pass hours and minutes
```

As another example, we can use `hh_mm_ss` to print the attributes of a duration in a different form:

```
auto t0 = std::chrono::system_clock::now();
...
auto t1 = std::chrono::system_clock::now();
std::chrono::hh_mm_ss hms{t1 - t0}
std::cout << "minutes:  " << hms.hours() + hms.minutes() << '\n';
std::cout << "seconds:  " << hms.seconds() << '\n';
std::cout << "subsecs:  " << hms.subseconds() << '\n';
```

might print:

```
minutes:  63min
seconds:  19s
subsecs:  502998000ns
```

There is no way to directly initialize the different attributes of `hh_mm_ss` object with specific values. In general, you better use duration types to deal with times:

```
using namespace std::literals;
...
```

```
auto t1 = 18h + 30min;    // 18 hours and 30 minutes
```

The most powerful functionality of `hh_mm_ss` is that it takes *any* precision duration and transforms it into the usual attributes and duration types `hours`, `minutes`, and `seconds`. In addition, `subseconds()` yields the rest of the value with an appropriate duration type such as `milliseconds` or `nanoseconds`. Even if the unit is a not a power of 10 (e.g., thirds of a second), `hh_mm_ss` transforms that into subseconds with a power of 10 with up to 18 digits. If that does not represent the exact value, a precision of 6 digits is used. You can use the standard output operator to print the result as a whole. For example:

```
std::chrono::duration<int, std::ratio<1,3>> third{1};
auto manysecs = 10000s;
auto manydbl = 10000.0s;

std::cout << "third:    " << third << '\n';
std::cout << "          " << std::chrono::hh_mm_ss{third} << '\n';
std::cout << "manysecs: " << manysecs << '\n';
std::cout << "          " << std::chrono::hh_mm_ss{manysecs} << '\n';
std::cout << "manydbl:  " << manydbl << '\n';
std::cout << "          " << std::chrono::hh_mm_ss{manydbl} << '\n';
```

This code has the following output:

```
third:    1[1/3]s
          00:00:00.333333
manysecs: 10000s
          02:46:40
manydbl:  10000.000000s
          02:46:40.000000
```

For output of specific attributes, you can also use formatted output with specific conversion specifiers:

```
auto manysecs = 10000s;
std::cout << "manysecs: " << std::format("{:%T}", manysecs) << '\n';
```

This also writes:

```
manysecs: 02:46:40
```

14.4 Time Zones

In large countries or for international communication it is not enough to say “let us meet at noon” because we have different time zones. The new chrono library supports this fact by having an API to deal with the different time zones we have on Earth including dealing with *standard* (“winter”) and *daylight saving* (“summer”) time.

14.4.1 Characteristics of Time Zones

Dealing with time zones is a bit tricky because it is a topic that can become pretty complex. For example, you have to take into account:

- Time zone differences are not necessarily multiples of hours. In fact we also have differences of 30 or even 15/45 minutes. For example, a significant part of Australia (the Northern Territory and South Australia with Adelaide) has the standard time zone of UTC+9:30 and Nepal has the time zone of UTC+5:45. Two time zones may also have 0 minutes difference (such as time zones in north and south America).
- Time zone abbreviations might refer to different time zones. For example, ***CST*** might stand for *Central Standard Time* (the standard time zone of Chicago, Mexico City, and Costa Rica) or *China Standard Time* (the international name of the time zone of Beijing and Shanghai). or *Cuba Standard Time* (the standard time zone of Havana).
- Time zones change. Countries decide to change their zone(s) or when daylight saving time starts. As a consequence, you might have to take multiple updates per year into account when dealing with time zones.

The Time Zone Database

To deal with time zones, the C++ standard library uses the IANA time zone database, which is available at `http://www.iana.org/time-zones`. This database has multiple updates each year to be up-to-date when time zones change so that programs can react accordingly.

The key entry for the time zone database is a time zone name, which is usually

- a ***city*** in a certain area.
  For example: `America/Chicago`, `Asia/Hong_Kong`, `Europe/Berlin`, `Pacific/Honolulu`
- a ***GMT entry*** with the negated offset from UTC time.
  For example: `Etc/GMT`, `Etc/GMT+6`, which stands for `UTC-6`, or *Etc/GMT-8*, which stands for `UTC+8`

Yes, the UTC time zone offsets are intentionally inverted as GMT entries; you cannot search for something like `UTC+6` or `UTC+5:45`.

A few additional canonical and alias names are supported (e.g., `UTC` or `GMT`) and also some deprecated entries are available (e.g., `PST8PDT`, `US/Hawaii`, `Canada/Central`, or just `Japan`). However, you cannot search for a single time zone abbreviation entry such as `CST` or `PST`. We discuss later how to deal with time zone abbreviations.

See also `http://en.wikipedia.org/wiki/List_of_tz_database_time_zones` for a list of time zone names.

Updating the Time Zone Database

The way systems handle the time zone database is implementation defined. The operating systems have to decide how to provide the necessary data and how to keep it up-to-date. Freestanding systems will usually not support this API. In that case, the request to a time zone (e.g., with `locate_zone()`) throws an exception.

An update of the timezone database is usually done as part of an operating system update. Such updates generally require a reboot of the machine, and so no C++ application runs during an update.

For systems that support to update the IANA time zone database without rebooting the machine, `std::chrono::reload_tzdb()` is provided. An update does not delete memory from the old database because the application may still have (`time_zone`) pointers into it. Instead, the new database is atomically pushed onto the front of a linked list, which you can get with `std::chrono::get_tzdb_list()`. `get_tzdb()` always yields the front of this list, so that any new calls to functions such as `locate_zone()` return pointers into the new database.

You can check the current version of your timezone database with the string member `version` of the type `tzdb` returned by `get_tzdb()`. For example:

```
std::cout << "tzdb version: " << chr::get_tzdb().version << '\n';
```

Usually the output is a year and an ascending alphabetic character for the update of that year (e.g., `"2021b"`). `remote_version()` provides the version of the latest available timezone database, which you can use to decide whether to call `reload_tzdb()`.

If a long-running program is using the chrono timezone database, but never calls `reload_tzdb()`, that program will not be aware of any updates of the database. It will continue to use whatever version of the database existed when the program first accessed it.

See http://github.com/HowardHinnant/date/wiki/Examples-and-Recipes#tzdb_manage for more details and examples about reloading the IANA timezone database for long-running programs.

14.4.2 Using Time Zones

To deal with time zones, two types play a basic role:

- `std::chrono::time_zone`,
  a type that represents a specific time zone.
- `std::chrono::zoned_time`,
  a type that represents a specific point of time associated with a specific time zone.

Let us look at them in detail.

Type `time_zone`

All possible time zone values are predefined by the IANA timezone database. For this reason, you cannot create a `time_zone` object by just declaring it. The values come from the timezone database and what you usually deal with is with pointers to these objects:

- `std::chrono::current_zone()` yields a pointer to the current time zone.
- `std::chrono::locate_zone(`*name*`)` yields a pointer to the time zone *name*.
- The timezone database returned by `std::chrono::get_tzdb()` has non-pointer collections with all timezone entries
  - Member `zones` has all canonical entries.
  - Member `links` has all alias entries with links to them.

  This you can use to find timezones by characteristics such as their abbreviated name.

For example:

```
auto tzHere = std::chrono::current_zone();      // type const time_zone*
auto tzUTC = std::chrono::locate_zone("UTC");   // type const time_zone*
```

```
...
std::cout << tzHere->name() << '\n';
std::cout << tzUTC->name() << '\n';
```

The output depends on your current timezone. For me in Germany, it looks like this:

```
Europe/Berlin
Etc/UTC
```

As you can see, you can search for any entry in the timezone database (such as `"UTC"`), but what you get is the canonical entry of it if any (`"UTC"` is just a time zone link to `"Etc/UTC"`). If `locate_zone()` finds no corresponding entry with that name, a `std::runtime_error` exception is thrown.

You cannot do much with a `time_zone`. The most important thing is to combine it with a system timepoint or with a local timepoint.

If you output a `time_zone` you get some implementations defined output only for debugging purposes:

```
std::cout << *tzHere << '\n';     // some implementation-defined debugging output
```

The chrono library also allows you to define and use types for custom timezones.

Type `zoned_time`

Objects of type `std::chrono::zoned_time` apply timepoints to time zones. You have two options to perform this conversion:

- Apply a system timepoint (a timepoint that belongs to the system clock) to a time zone. In that case we convert the timepoint of an event to happen simultaneously the local time of the other time zone.
- Apply a *local timepoint* (a timepoint that belongs to the pseudo clock `local_t`) to a time zone. In that case we apply a timepoint as it is as local time to another time zone.

In addition, you can convert the point of time of a `zoned_time` to a different time zone by initializing a new `zoned_time` object with another `zoned_time` object.

For example, let us schedule both a local party at every office at 18:00 and a company party over multiple time zones at the end of September 2021:

```
auto day = 2021y/9/chr::Friday[chr::last];                    // last Friday of month
chr::local_seconds tpOfficeParty{chr::local_days{day} - 6h}; // 18:00 the day before
chr::sys_seconds tpCompanyParty{chr::sys_days{day} + 17h};    // 17:00 that day

std::cout << "Berlin Office and Company Party:\n";
std::cout << "  " << chr::zoned_time{"Europe/Berlin", tpOfficeParty} << '\n';
std::cout << "  " << chr::zoned_time{"Europe/Berlin", tpCompanyParty} << '\n';

std::cout << "New York Office and Company Party:\n";
std::cout << "  " << chr::zoned_time{"America/New_York", tpOfficeParty} << '\n';
std::cout << "  " << chr::zoned_time{"America/New_York", tpCompanyParty} << '\n';
```

The output of this code is as follows:

```
Berlin Office and Company Party:
  2021-09-23 18:00:00 CEST
```

```
  2021-09-24 19:00:00 CEST
New York Office and Company Party:
  2021-09-23 18:00:00 EDT
  2021-09-24 13:00:00 EDT
```

Details matter when combining timepoints and time zones. For example, consider this code:

```
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now());  // system timepoint
auto locTime = chr::zoned_time{chr::current_zone(), sysTp};       // local time
...
std::cout << "sysTp:       " << sysTp << '\n';
std::cout << "locTime:     " << locTime << '\n';
```

First, we initialize `sysTp` as a the current system timepoint in seconds and combine this timepoint with the current timezone. The output shows the system and the local timepoint of the same point in time:

```
sysTp:       2021-04-13 13:40:02
locTime:     2021-04-13 15:40:02 CEST
```

Now let initialize a local timepoint. One way to do this is just to convert a system timepoint to a local timepoint. For this we need a time zone. If we use the current timezone, the local time is converted to UTC:

```
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now());  // system timepoint
auto curTp = chr::current_zone()->to_local(sysTp);                // local timepoint
std::cout << "sysTp:       " << sysTp << '\n';
std::cout << "locTp:       " << locTp << '\n';
```

The corresponding output is as follows:

```
sysTp:       2021-04-13 13:40:02
curTp:       2021-04-13 13:40:02
```

However, if we use UTC as timezone, the local time is *used* as local time without an associated timezone:

```
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now());  // system timepoint
auto locTp = std::chrono::locate_zone("UTC")->to_local(sysTp);    // take local time as it is
std::cout << "sysTp:          " << sysTp << '\n';
std::cout << "locTp:          " << locTp << '\n';
```

According to the output, both timepoints look the same:

```
sysTp:       2021-04-13 13:40:02
locTp:       2021-04-13 13:40:02
```

However, they are not. `sysTp` has an associated UTC epoch but `locTp` has not. If we now apply the local timepoint to a time zone, we do not convert, we specify the missing timezone leaving the time as it is:

```
auto timeFromSys = chr::zoned_time{chr::current_zone(), sysTp};   // converted time
auto timeFromLoc = chr::zoned_time{chr::current_zone(), locTp};   // applied time
std::cout << "timeFromSys: " << timeFromSys << '\n';
std::cout << "timeFromLoc: " << timeFromLoc << '\n';
```

Thus, the output is as follows:

```
timeFromSys: 2021-04-13 15:40:02 CEST
timeFromLoc: 2021-04-13 13:40:02 CEST
```

Now let us combine all 4 objects with the timezone of New York:

```
std::cout << "NY sysTp:       "
          << std::chrono::zoned_time{"America/New_York", sysTp} << '\n';
std::cout << "NY locTP:       "
          << std::chrono::zoned_time{"America/New_York", locTp} << '\n';
std::cout << "NY timeFromSys: "
          << std::chrono::zoned_time{"America/New_York", timeFromSys} << '\n';
std::cout << "NY timeFromLoc: "
          << std::chrono::zoned_time{"America/New_York", timeFromLoc} << '\n';
```

The output is as follows:

```
NY sysTp:       2021-04-13 09:40:02 EDT
NY locTP:       2021-04-13 13:40:02 EDT
NY timeFromSys: 2021-04-13 09:40:02 EDT
NY timeFromLoc: 2021-04-13 07:40:02 EDT
```

The system timepoint and the local time derived from it both convert the time now to the time zone of New York. As usual, the local timepoint is applied to New York so that we have now the same value as the original time with the timezone removed. `timeFromLoc` is the initial local time 13:40:02 in Central Europe applied to the timezone of New York.

14.5 I/O with Chrono Types

C++20 provides new support to directly output and parse almost all chrono types.

14.5.1 Default Output Formats

For more or less all chrono types the standard output operator is defined since C++20. If useful, it does not only print the value but also uses an appropriate format and the unit. Locale-dependent formatting is possible,

All **calendrical types** print out the value as listed with the output formats of calendrical types.

All **duration types** print out the value with the unit type as listed in the table of table *Output units for durations*. This fits any literal operator if provided.

Unit	Output Suffix
`atto`	`as`
`femto`	`fs`
`pico`	`ps`
`nano`	`ns`
`micro`	`µs` or `us`
`milli`	`ms`
`centi`	`cs`
`deci`	`ds`
`ratio<1>`	`s`
`deca`	`das`
`hecto`	`hs`
`kilo`	`ks`
`mega`	`Ms`
`giga`	`Gs`
`tera`	`Ts`
`peta`	`Ps`
`exa`	`Es`
`ratio<60>`	`min`
`ratio<3600>`	`h`
`ratio<86400>`	`d`
`ratio<`*num*`,1>`	`[`*num*`]s`
`ratio<`*num*`,`*den*`>`	`[`*num*`/`*den*`]s`

Table 14.7. Output units for durations

For all standard **timepoint** types an output operator prints date and optionally time in the following format:

- *year*-*month*-*day* for an integral granularity unit implicitly convertible to days
- *year*-*month*-*day hour*:*minutes*:*seconds* for an integral granularity unit equal or less than days

If the type of the timepoint value (member rep) has a floating-point type, no output operator is defined.[3]

For the time part, the output operator of type hh_mm_ss is used. This corresponds with the %F %T conversion specifier for formatted output.

For example:

```
auto tpSys = std::chrono::system_clock::now();
std::cout << tpSys << '\n';                          //2021-04-25 13:37:02.936314000

auto tpL = chr::zoned_time{chr::current_zone(), tpSys}.get_local_time();
std::cout << tpL;                                    //2021-04-25 15:37:02.936314000
std::cout << chr::floor<chr::milliseconds>(tpL);     //2021-04-25 15:37:02.936
std::cout << chr::floor<chr::seconds>(tpL);          //2021-04-25 15:37:02
std::cout << chr::floor<chr::minutes>(tpL);          //2021-04-25 15:37:02
std::cout << chr::floor<chr::days>(tpL);             //2021-04-25
std::cout << chr::floor<chr::weeks>(tpL);            //2021-04-22

auto tp3 = std::chrono::floor<chr::duration<long long, std::ratio<1, 3>>>(tpSys);
std::cout << tp3 << '\n';                            //2021-04-25 13:37:02.666666

chr::sys_time<chr::duration<double, std::milli>> tpD{tpSys};
std::cout << tpD << '\n';                            // ERROR: no output operator defined

std::chrono::gps_seconds tpGPS;
std::cout << tpGPS << '\n';                          //1980-01-06 00:00:00

auto tpStd = std::chrono::steady_clock::now();
std::cout << "tpStd: " << tpStd;                     // ERROR: no output operator defined
```

The zoned_time<> type outputs like the timepoint type extended with the abbreviated time zone name. The following time zone abbreviations are used for the standard clocks:

- UTC for sys_clock, utc_clock, and file_clock
- TAI for tai_clock
- GPS for gps_clock

Finally, note that sys_info and local_info have output operators with an undefined format. It should only be used for debugging purposes.

14.5.2 Formatted Output

Chrono supports the new library for formatted output. That means you can use date/time types for arguments of std::format() and std::format_to().

For example:

```
auto t0 = std::chrono::system_clock::now();
```

[3] This is a gap that will hopefully be fixed with C++23.

```
...
auto t1 = std::chrono::system_clock::now();

std::cout << std::format("From {} to {}\nit took {}\n", t0, t1, t1-t0);
```

This would use the default output formats of the date/time types. For example, we might have the following output:

```
From 2021-04-03 15:21:33.197859000 to 2021-04-03 15:21:34.686544000
it took 1488685000ns
```

To improve the output you can constrain the types or use specific conversions specifiers. Most of them (but not all) correspond with the POSIX `date` and `strftime()` commands. For example:

```
std::cout << std::format("From {:%T} to {:%T} it took {:%S}s\n", t0, t1, t1-t0);
```

This might have the following output then:

```
From 15:21:34.686544000 to 15:21:34.686544000 it took 01.488685000s
```

The format specifiers for chrono types are a part of the syntax for standard format specifiers (each specifier is optional):

fill align width **.***prec* **L** *spec*

- *fill*, *align*, *with*, and *prec* mean the same as for standard format specifiers.
- *spec* specifies the general notation of the formatting starting with **%**.
- **L** also as usual turns on locale-dependent formatting, for specifiers that support it.

Table *Conversion specifiers for chrono types* lists all conversion specifiers for formatted output of date/time types with an example based on Sunday, June 9, 2019 17:33:16 and 850 Milliseconds.

Without using a specific conversion specifier, the default output operator is used, which marks invalid dates. When using specific conversion specifiers this is not the case:

```
std::chrono::year_month_day ymd{2021y/2/31};    // February 31, 2021
std::cout << std::format("{}", ymd);            // 2021-02-31 is not a valid ...
std::cout << std::format("{:%F}", ymd);         // 2021-02-31
std::cout << std::format("{:%Y-%m-%d}", ymd);   // 2021-02-31
```

If the date/time value type does not provide the necessary information for a conversion specifier, a `std::format_error` exception is thrown. For example, an exception is thrown when:

- A year specifier is used for a `month_day`.
- A weekday specifier is used for a duration.
- A month or weekday name should be printed and the value is not valid.
- A time zone specifier is used for a local timepoint.

In addition, note the following about conversion specifiers:

- A negative duration or `hh_mm_ss` value print the minus sign in front of the whole value. For example:

  ```
  std::cout << std::format("{:%H:%M:%S}", -10000s);  // outputs: -02:46:40
  ```

- Different week number and year formats might result different output values.
  For example, Sunday, January 1, 2023 yields:

  - Week 00 with %W (week before first Monday)
  - Week 01 with %U (week with first Monday)
  - Week 52 with %V (ISO week: week before Monday of the week 01, which has January 4th)

  Because the ISO week might be the last week of the previous year, the ISO year, which is the year of that week, may be one less:
  - Year 2023 with %Y
  - Year 23 with %y
  - Year 2022 with %G (ISO year of the ISO week %V, which is the last week of the previous month)
  - Year 22 with %g (ISO year of the ISO week %V, which is the last week of the previous month)
- The following time zone abbreviations are used for the standard clocks:
  - UTC for sys_clock, utc_clock, and file_clock
  - TAI for tai_clock
  - GPS for gps_clock

All conversion specifiers except %q and %Q can also be used for formatted parsing.

14.5.3 Locale Dependent Output

The default output operator for the various types uses a locale-dependent format if the output stream is imbued by a locale that has its own format. For example:

```
using namespace std::literals;
auto dur = 42.2ms;

std::cout << dur << '\n';     // 42.2ms

#ifdef _MSC_VER
  std::locale locG("deu_deu.1252");
#else
  std::locale locG("de_DE");
#endif
std::cout.imbue(locG);        // switch to German locale
std::cout << dur << '\n';     // 42,2ms
```

Formatted output with std::format() is handled as usual:[4]

- By default, formatted output uses the locale independent "C" locale.
- By specifying L you can switch to a locale dependent output specified via a locale parameter or as global locale.

That means, to use a locale-dependent notation, you have to use the L specifier and pass either pass the locale as first argument to std::format() or set the global locale before calling it. For example:

```
using namespace std::literals;
```

[4] This behavior was specified as a bug fix to C++20 with http://wg21.link/p2372 so that the original wording of C++20 does not specify this behavior.

```
auto dur = 42.2ms;                                  // duration to print

#ifdef _MSC_VER
  std::locale locG("deu_deu.1252");
#else
  std::locale locG("de_DE");
#endif

std::string s1 = std::format("{:%S}", dur);         // "00.042s" (not localized)
std::string s3 = std::format(locG, "{:%S}", dur);   // "00.042s" (not localized)
std::string s2 = std::format(locG, "{:L%S}", dur);  // "00,042s" (localized)

std::locale::global(locG);                          // set German locale globally
std::string s4 = std::format("{:L%S}", dur);        // "00,042s" (localized)
```

In several cases, you can even use an alternative locale's representation according to `strftime()` and ISO 8601:2004, which you can specify with a leading `O` or `E` in front of the conversion specifier:

- `E` can be used as the locale's alternate representations in front of `c`, `C`, `x`, `X`, `y`, `Y`, and `z`
- `O` can be used as the locale's alternate numeric symbols in front of `d`, `e`, `H`, `I`, `m`, `M`, `S`, `u`, `U` `V`, `w`, `W`, `y`, and `z`

Spec.	Example	Meaning
%c	Sun Jun 9 17:33:16 2019	Locale's date and time representation
Dates:		
%x	06/09/19	Locale's date representation
%F	2019-06-09	*year*-*month*-*day* with four and two digits
%D	06/09/19	*month*/*day*/*year* with two digits
%e	9	Day with leading space if single digit
%d	09	Day as two digits
%b	Jun	Locale's abbreviated month name
%h	Jun	same
%B	June	Locale's month name
%m	06	Month with two digits
%Y	2019	Year with four digits
%y	19	Year without century as two digits
%G	2019	ISO-week-based year as four digits (week according to %V)
%g	19	ISO-week-based year as two digits (week according to %V)
%C	20	Century as two digits
Weekdays and weeks:		
%a	Sun	Locale's abbreviated weekday name
%A	Sunday	Locale's weekday name
%w	0	Weekday as decimal number (Sunday as 0 until Saturday as 6)
%u	7	Weekday as decimal number (Monday as 1 until Sunday as 7)
%W	22	Week of the year (00 ... 53, week 01 starts with first Monday)
%U	23	Week of the year (00 ... 53, week 01 starts with first Sunday)
%V	23	ISO week of the year (01 ... 53, week 01 has Jan. 4th)
Times:		
%X	17:33:16	Locale's time representation
%r	05:33:16 PM	Locale's 12-hour clock time
%T	17:33:16.850	*hour*:*minutes*:*seconds* (local-dependent subseconds as needed)
%R	17:33	*hour*:*minutes* with two digits each
%H	17	24-hour clock hour as two digits
%I	05	12-hour clock hour as two digits
%p	PM	AM or PM according to the 12-hour clock
%M	33	Minute with two digits
%S	16.850	Seconds as decimal number (locale-specific subseconds as needed)
Other:		
%Z	CEST	Time zone abbreviation (may also be UTC, TAI, or GPS)
%z	+0200	Offset (hours and minutes) from UTC (%Ez or %Oz for +02:00)
%j	160	Day of the year with three digits (Jan. 1st is 001)
%q	ms	Unit suffix according to the time's duration
%Q	63196850	Value according to the time's duration
%n	\n	Newline character
%t	\t	Tabulator character
%%	%	% character

Table 14.8. Conversion specifiers for chrono types

14.5.4 Formatted Input

The chrono library also supports formatted input. You have two options:

- A free-standing function `std::chrono::from_stream()` is provided by certain date/time types to read in a specific value according to a passed format string.
- A manipulator `std::chrono::parse()` allows us to use `from_stream()` as part of a bigger parsing with the input operator `>>`.

Using `from_stream()`

The following code demonstrates how to use `from_stream()` by parsing a full time point:

```
std::chrono::sys_seconds tp;
std::istringstream sstrm{"2021-2-28 17:30:00"};

std::chrono::from_stream(sstrm, "%F %T", tp);
if (sstrm) {
  std::cout << "tp: " << tp << '\n';
}
else {
  std::cerr << "reading into tp failed\n";
}
```

The code generates the following output:

```
tp: 2021-02-28 17:30:00
```

As another example, you can parse a `year_month` from a sequence of characters specifying the full month name and the year among other things as follows:

```
std::chrono::year_month m;
std::istringstream sstrm{"Monday, April 5, 2021"};
std::chrono::from_stream(sstrm, "%A, %B %d, %Y", m);
if (sstrm) {
  std::cout << "month: " << m << '\n';  // prints: month: 2021/Apr
}
```

The format string accepts all conversion specifiers of formatted output except `%q` and `%Q` with improved flexibility. For example:

- `%d` stands for 1 or two characters to specify the day and with `%4d` you can specify that even up to 4 characters are parsed.
- `%n` stands for exactly one whitespace character
- `%t` stands for zero or one whitespace character
- A whitespace character such as a space represents an arbitrary number of whitespaces (including zero whitespaces).

`from_stream()` is provided for the following types:

- a `duration<>` of any type

- a sys_time<>, utc_time<>, gps_time<>, tai_time<>, local_time<>, or file_time<> of any duration
- a day, a month, or a year
- a year_month, a month_day, or a year_month_day
- a weekday

The format has to be a C string of type const char*. It must match the characters in the input stream and the value to parse. The parsing fails if

- the input sequence of characters does not match the required format
- the format provides not enough information for the value
- the parsed date is not valid.

In that case, the failbit of the stream is set, which you can test with calling fail() or using the stream as Boolean value.

A Generic Function to Parse Date/Times

In practice, dates and times are rarely hard coded. However, when testing code, there is often a need for an easy way to specify a date/time value. Here is a little helper function I used to test the examples of this book:

lib/chronoparse.hpp

```
#include <chrono>
#include <string>
#include <sstream>
#include <cassert>

// parse year-month-day with optional hour:minute and optional :sec
// - returns a time_point<> of the passed clock (default: system_clock)
//   in seconds
template<typename Clock = std::chrono::system_clock>
auto parseDateTime(const std::string& s)
{
  // return value:
  std::chrono::time_point<Clock, std::chrono::seconds> tp;

  // string stream to read from:
  std::istringstream sstrm{s};   // no string_view support

  auto posColon = s.find(":");
  if (posColon != std::string::npos) {
    if (posColon != s.rfind(":")) {
      // multiple colons:
      std::chrono::from_stream(sstrm, "%F %T", tp);
    }
    else {
```

```
      // one colon:
      std::chrono::from_stream(sstrm, "%F %R", tp);
    }
  }
  else {
    // no colon:
    std::chrono::from_stream(sstrm, "%F", tp);
  }

  // handle invalid formats:
  assert((!sstrm.fail()));

  return tp;
}
```

You can use it as follows:

lib/chronoparse.cpp

```
#include "chronoparse.hpp"
#include <iostream>

int main()
{
  auto tp1 = parseDateTime("2021-1-1");
  std::cout << std::format("{:%F %T %Z}\n", tp1);

  auto tp2 = parseDateTime<std::chrono::local_t>("2021-1-1");
  std::cout << std::format("{:%F %T}\n", tp2);

  auto tp3 = parseDateTime<std::chrono::utc_clock>("2015-6-30 23:59:60");
  std::cout << std::format("{:%F %T %Z}\n", tp3);

  auto tp4 = parseDateTime<std::chrono::gps_clock>("2021-1-1 18:30");
  std::cout << std::format("{:%F %T %Z}\n", tp4);
}
```

The program has the following output:

```
2021-01-01 00:00:00 UTC
2021-01-01 00:00:00
2015-06-30 23:59:60 UTC
2021-01-01 18:30:00 GPS
```

Note that for a local timepoint you cannot use %Z to print its timezone (using it would raise an exception).

Using the parse() Manipulator

Instead of calling from_stream()

```
std::chrono::from_stream(sstrm, "%F %T", tp);
```

you could also call:

```
sstrm >> std::chrono::parse("%F %T", tp);
```

Please note that the original C++20 standard does not formally allow you to pass the format directly as a string literal, so that you have to call

```
sstrm >> std::chrono::parse(std::string{"%F %T"}, tp);
```

However, this should be fixed with http://wg21.link/lwg3554.

std::chrono::parse() is an I/O stream manipulator. It allows you to parse multiple values inside one statement reading from an input stream. In addition, thanks to move semantics you can even pass a temporary input stream. For example:

```
chr::sys_days tp;
chr::hours h;
chr::minutes m;
// parse date into tp, hour into h and minute into m:
std::istringstream{"12/24/21 18:00"} >> chr::parse("%D", tp)
                                     >> chr::parse(" %H", h)
                                     >> chr::parse(":%M", m);
std::cout << tp << " at " << h << ' ' << m << '\n';
```

This code outputs:

```
2021-12-24 at 18h 0min
```

Again, note that you might have to explicitly convert the string literals "%D", " %H", and ":%M" to strings.

Parsing Time Zones

Parsing time zones is a bit tricky, because time zone abbreviations are not unique:

To help here, from_stream() has the following formats:

istream **from_stream(***istream*, *format*, *value***)**
istream **from_stream(***istream*, *format*, *value*, *abbrevPtr***)**
istream **from_stream(***istream*, *format*, *value*, *abbrevPtr*, *offsetPtr***)**

Optionally you can pass the address of a std::string to store a parsed time zone abbreviation into the string and you can pass the address of a std::chrono::minutes object to store a parsed time zone offset into that string (in both cases nullptr can be passed).

However, still you have to be careful:

- This works:

```
chr::sys_seconds tp;
std::istringstream sstrm{"2021-4-13 12:00 UTC"};
chr::from_stream(sstrm, "%F %R %Z", tp);
```

```
std::cout << std::format("{:%F %R %Z}\n", tp);  // 2021-04-13 12:00 UTC
```

However, it works only because system timepoints use `UTC` anyway.

- This does not work, because it ignores the time zone:

```
chr::sys_seconds tp;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp);
std::cout << chr::format("{:%F %R %Z}", tp);  // 2021-04-13 12:00 UTC
```

`%Z` is used to parse `MST` but there is no parameter to store the value.

- This seems to work:

```
chr::sys_seconds tp;
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << tp << '\n';       // 2021-04-13 12:00
std::cout << tzAbbrev << '\n'; // MST
```

However, if you compute the zoned time you see that you are converting a UTC time to a different time zone:

```
chr::zoned_time zt{tzAbbrev, tp};     // OK: MST exists
std::cout << zt << '\n';       // 2021-04-13 05:00:00 MST
```

- This really seems to work:

```
chr::local_seconds tp;  // local time
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << tp << '\n';       // 2021-04-13 12:00
std::cout << tzAbbrev << '\n'; // MST
chr::zoned_time zt{tzAbbrev, tp};     // OK: MST exists
std::cout << zt << '\n';       // 2021-04-13 12:00:00 MST
```

However, we were lucky that `MST` is one of the few abbreviations available as a deprecated entry in the timezone database. The moment you use this code with `CEST` or `CST` it throws an exception when initializing the `zoned_time`.

- So, either you use only `tzAbbrev` instead of `zoned_time` and `%Z`:

```
chr::local_seconds tp;  // local time
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 CST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << chr::format("{:%F %R} {}", tp, tzAbbrev);  // 2021-04-13 12:00 CST
```

or you have to deal with code to map a time zone abbreviation to a time zone.

Note that `%Z` cannot parse the pseudo time zones `GPS` and `TAI`.

14.6 Using the Chrono Extensions in Practice

After introducing the new features and types of the chrono library, this section discusses how to use them in practice. See also http://github.com/HowardHinnant/date/wiki/Examples-and-Recipes for more examples and recipes.

14.6.1 Invalid Dates

Values of calendrical types may not be valid. This can happen in two ways:

- By an initialization with an invalid value. For example:

```
std::chrono::day d{0};                          // invalid day
std::chrono::year_month ym{2021y/13};           // invalid year_month
std::chrono::year_month_day ymd{2021y/2/31};    // invalid year_month_day
```

- By a computation that results in an invalid date. For example:

```
auto ymd1 = std::chrono::year{2021}/1/31;       // January 31, 2021
ymd1 += std::chrono::months{1};                 // February 31, 2021  (invalid)

auto ymd0 = std::chrono::year{2020}/2/29;       // February 29, 2020
ymd1 += std::chrono::years{1};                  // February 29, 2021  (invalid)
```

Table *Valid values of standard date members* lists the internal types and possible values of the different attributes of a date.

Member	Internal Type	Valid Values
day	`unsigned char`	1 until 31
month	`unsigned char`	1 until 12
year	`short`	-32767 until 32767
weekday	`unsigned char`	0 (Sunday) until 6 (Saturday) and 7 (again Sunday), which is converted to 0
weekday index	`unsigned char`	1 until 5

Table 14.9. Valid values of standard date members

All combined types are valid if the individual components are valid (e.g., a valid `month_weekday` is valid if both the month and the weekday are valid). However, additional checks may apply:

- A full date of type `year_month_day` has to exists. By taking leap years into account, For example:

```
2020y/2/29;          // valid (there is a February 29 in 2020)
2021y/2/29;          // invalid (there is no February 29 in 2021)
```

- A `month_day` is only valid if the day could be valid in that month For February, 29 is valid, but 30 is not. For example:

```
February/29;         // valid (a February can have 29 days)
February/30;         // invalid (no February can have 30 days)
```

- A `year_month_weekday` is only valid if the weekday index can exist in the specified month of a year. For example:

```
2020y/1/Thursday[5];   // valid (there is a fifth Thursday in January 2020)
2020y/1/Sunday[5];     // invalid (there is no fifth Sunday in January 2020)
```

Each calendrical type provides a member function `ok()` to check whether the value is valid. Default output operators signal invalid dates.

The way to handle invalid dates depends on your programming logic. For the typical scenario where you have created a day too high for a month, you have the following options:

- Round down to the last day of the month:

```
auto ymd = std::chrono::year{2021}/1/31;
ymd += std::chrono::months{1};
if (!ymd.ok()) {
  ymd = ymd.year()/ymd.month()/std::chrono::last;                // February 28, 2021
}
```

  Note that the expression on the right creates a `year_month_last` which is then converted to the `year_month_day`.

- Round up to the first day of the next month:

```
auto ymd = std::chrono::year{2021}/1/31;
ymd += std::chrono::months{1};
if (!ymd.ok()) {
  ymd = ymd.year()/ymd.month()/1 + std::chrono::months{1};      // March 1, 2021
}
```

  Do not add just 1 to the month because for December you create an invalid month.

- Round up to the date according to all overflown days:

```
auto ymd = std::chrono::year{2021}/1/31;
ymd += std::chrono::months{1};                                  // March 3, 2021
if (!ymd.ok()) {
  ymd = std::chrono::sys_days(ymd);
}
```

  This uses a special feature of the conversion of a `year_month_day` where all overflown days are logically added to the next month. However, this does only work for a few days (you cannot add 1000 days that way).

If the date is not a valid value the default output format will signal that with "is not a valid *type*." For example:

```
std::chrono::day d{0};                          // invalid day
std::chrono::year_month_day ymd{2021y/2/31};    // invalid year_month_day
std::cout << "day: " << d << '\n';
std::cout << "ymd: " << ymd << '\n';
```

will output:[5]

[5] Note that the specification in the C++20 standard is a bit inconsistent here and is currently fixed. For example, it states that for an invalid `year_month_days` "*is not a valid **date***" is the output.

```
day: 00 is not a valid day
ymd: 2021-02-31 is not a valid year_month_day
```

The same happens when using the default formatting (just {}) of formatted output. By using specific conversion specifiers, you can disable the "*is not a valid*" output (in software for banking or quarterly processing a day such as June 31 is sometimes even used):

```
std::chrono::year_month_day ymd{2021y/2/31};
std::cout << ymd2 << '\n';                          //2021-02-31 is not a valid year_month_day
std::cout << std::format("{:%F}\n", ymd);           //2021-02-31
std::cout << std::format("{:%Y-%m-%d}\n", ymd);     //2021-02-31
```

14.6.2 Dealing with `months` and `years`

Because `std::chrono::months` and `std::chrono::years` are not integral multiples of days, you have to be careful when using them.

- For standard types that have their own value just for the year (`month`, `year`, `year_month`, `year_month_day`, `year_month_weekday_last`, ...) they work fine when adding a specific number of months or years to a date.
- For standard time point types that have one value for the date as a whole (`time_point`, `sys_time`, `sys_seconds`, and `sys_days`) they add the corresponding average fractional period to a date, which might not result in the date you expect.

For example, let us look at the different effects of adding 4 months or 4 years to December 31, 2020:

- When dealing with `year_month_day`:

```
chr::year_month_day ymd0 = chr::year{2020}/12/31;
auto ymd1 = ymd0 + chr::months{4};              // OOPS: April 31, 2021
auto ymd2 = ymd0 + chr::years{4};               // OK: December 31, 2024

std::cout << "ymd:         " << ymd0 << '\n';   //2020-12-31
std::cout << " +4months:  " << ymd1 << '\n';   //2021-04-31 is not a valid ...
std::cout << " +4years:   " << ymd2 << '\n';   //2024-12-31
```

- When dealing with `year_month_day_last`:

```
chr::year_month_day_last yml0 = chr::year{2020}/12/chr::last;
auto yml1 = yml0 + chr::months{4};              // OK: last day of April 2021
auto yml2 = yml0 + chr::years{4};               // OK: last day of Dec. 2024

std::cout << "yml:         " << yml0 << '\n';   //2020/Dec/last
std::cout << " +4months:  " << yml1 << '\n';   //2021/Apr/last
std::cout << " as date:   "
          << chr::sys_days{yml1} << '\n';       //2021-04-30
std::cout << " +4years:   " << yml2 << '\n';   //2024/Dec/last
```

- When dealing with `sys_days`:

```
chr::sys_days day0 = chr::year{2020}/12/31;
auto day1 = day0 + chr::months{4};              // OOPS: May 1, 2021 17:56:24
```

```
auto day2 = day0 + chr::years{4};                // OOPS: Dec. 30, 2024 23:16:48

std::cout << "day:         " << day0 << '\n';    // 2020-12-31
std::cout << " with time: "
          << chr::sys_seconds{day0} << '\n';     // 2020-12-31 00:00:00
std::cout << " +4months:   " << day1 << '\n';    // 2021-05-01 17:56:24
std::cout << " +4years:    " << day2 << '\n';    // 2024-12-30 23:16:48
```

This feature is only supported to be able to model things such as physical or biological processes that do not care about the intricacies of human calendars (weather, gestation periods, etc.) and should not be used for other purposes.

Note that both the output values and the default output formats differ. That is a clear sign that different types are used:

- When adding months or years to calendrical types, they process the right logical date (the same day or again the last day of the next month or year). Please note that this might result in invalid dates such as April 31, which the default output operator even signals in its output as follows:[6]

  ```
  2021-04-31 is not a valid year_month_day
  ```

- When adding months or years to a day of type `std::chrono::sys_days` the result is not of type `sys_days`. As usual in the chrono library, the result has the best type being able to represent any possible result:
  - Adding `months` yields a type with a unit of 54 seconds.
  - Adding `years` yields a type with a unit of 216 seconds.

  Both units are a multiple of a second so that they can be used as `std::chrono::sys_seconds`. And the corresponding output operator by default prints both the day and the time in seconds, which as you can see is not the same day and time of a later month or year. Both time points are no longer the 31st or last day of a month and the time is no longer midnight:

  ```
  2021-05-01 17:56:24
  2024-12-30 23:16:48
  ```

Both can be useful. However, using `months` and `years` with timepoints is usually only useful to compute the rough day after many months and/or years.

14.6.3 Parsing Time Points and Durations

If you have a time point or duration, you can access the different fields as demonstrated by the following program:

lib/chronoattr.cpp

```
#include <chrono>
#include <iostream>
```

[6] Due to some inconsistencies, the exact format of invalid dates might not match the C++20 standard, which specifies "`is not a valid date`" here.

```
int main()
{
  namespace chr = std::chrono;       // shortcut for std::chrono

  auto now = chr::system_clock::now();       // sys_time<>
  auto today = chr::floor<chr::days>(now);   // sys_days
  chr::year_month_day ymd{today};
  chr::hh_mm_ss hms{now - today};
  chr::weekday wd{today};
  chr::sys_info info{chr::current_zone()->get_info(now)};

  std::cout << "now:      " << now << '\n';
  std::cout << "today:    " << today << '\n';
  std::cout << "ymd:      " << ymd << '\n';
  std::cout << "hms:      " << hms << '\n';
  std::cout << "year:     " << ymd.year() << '\n';
  std::cout << "month:    " << ymd.month() << '\n';
  std::cout << "day:      " << ymd.day() << '\n';
  std::cout << "hours:    " << hms.hours() << '\n';
  std::cout << "minutes:  " << hms.minutes() << '\n';
  std::cout << "seconds:  " << hms.seconds() << '\n';
  std::cout << "subsecs:  " << hms.subseconds() << '\n';
  std::cout << "weekday:  " << wd << '\n';
  std::cout << "timezone: " << info.abbrev << '\n';
}
```

The output of the program might is, for example, as follows:

```
now:      2021-04-02 13:37:34.059858000
today:    2021-04-02
ymd:      2021-04-02
hms:      13:37:34.059858000
year:     2021
month:    Apr
day:      02
hours:    13h
minutes:  37min
seconds:  34s
subsecs:  59858000ns
weekday:  Fri
timezone: CEST
```

The function `now()` yields a timepoint with the granularity of the system clock:

```
auto tpNow = chr::system_clock::now();       // sys_time<>
```

If you want to deal with the local time, you have to do this:

```
auto tpLoc = chr::zoned_time{chr::current_zone(),
                             chr::system_clock::now()}.get_local_time();
```

To deal with the date part of the timepoint we need the granularity of days, which we get with:

```
auto today = chr::floor<chr::days>(tpNow);   // sys_days
```

The result `today` has the type `sys_days`. This, you can assign to a `year_month_day`, so that you can access year, month, and day by the corresponding member functions:

```
chr::year_month_day ymd{today};
std::cout << "year:     " << ymd.year() << '\n';
std::cout << "month:    " << ymd.month() << '\n';
std::cout << "day:      " << ymd.day() << '\n';
```

To deal with the time part of the timepoint we need a duration, which we get when we compute the difference of our original timepoint and its value at midnight. We use the duration to initialize a `hh_mm_ss` object:

```
chr::hh_mm_ss hms{tpNow - today};
```

From this you can directly get the hour, minute, second, subseconds:

```
std::cout << "hours:    " << hms.hours() << '\n';
std::cout << "minutes:  " << hms.minutes() << '\n';
std::cout << "seconds:  " << hms.seconds() << '\n';
std::cout << "subsecs:  " << hms.subseconds() << '\n';
```

For the subseconds, `hh_mm_ss` finds out the necessary granularity and uses the appropriate unit. In our case it is printed as nanoseconds:

```
hms:      13:37:34.059858000
hours:    13h
minutes:  37min
seconds:  34s
subsecs:  59858000ns
```

For the weekday, you only have to initialize it with a type of granularity day. This works for `sys_days`, `local_days`, and `year_month_day` (because the latter implicitly converts to `std::sys_days`):

```
chr::weekday wd{today};   // OK (today has day granularity)
chr::weekday wd{ymd};     // OK due to implicit conversion to sys_days
```

For timezone aspects, you have to combine the timepoint (here `tpNow`) with a time zone (here the current time zone). The resulting `sys_info` object contains information such as the abbreviated timezone name:

```
chr::sys_info info{chr::current_zone()->get_info(tpNow)};
```

In our case we get my timezone when running the program, the central European summer time (CEST):

```
timezone: CEST
```

14.6.4 Dealing with Time Zone Abbreviations

Because time zone abbreviations might refer to different time zones, you cannot define a unique time zone by its abbreviation. Instead, you have to map the abbreviation to one of multiple IANA timezone entries.

The following program demonstrates this for the timezone abbreviation CST:

lib/chronocst.cpp

```
#include <iostream>
#include <chrono>

int main()
{
  auto abbrev = "CST";
  auto day = std::chrono::sys_days{2021_y/1/1};
  auto& db = std::chrono::get_tzdb();

  // print time and name of all time zones with abbrev:
  std::cout << abbrev << " at "
            << std::chrono::zoned_time{"UTC", day} << ":\n";
  for (const auto& z : db.zones) {
    if (z.get_info(day).abbrev == abbrev) {
      std::chrono::zoned_time zt{&z, day};
      std::cout << "  " << zt << "  " << z.name() << '\n';
    }
  }
}
```

The program has the following output:

```
CST at 2021-01-01 00:00:00 UTC:
  2020-12-31 18:00:00 CST  America/Bahia_Banderas
  2020-12-31 18:00:00 CST  America/Belize
  2020-12-31 18:00:00 CST  America/Chicago
  2020-12-31 18:00:00 CST  America/Costa_Rica
  2020-12-31 18:00:00 CST  America/El_Salvador
  2020-12-31 18:00:00 CST  America/Guatemala
  2020-12-31 19:00:00 CST  America/Havana
  2020-12-31 18:00:00 CST  America/Indiana/Knox
  2020-12-31 18:00:00 CST  America/Indiana/Tell_City
  2020-12-31 18:00:00 CST  America/Managua
  2020-12-31 18:00:00 CST  America/Matamoros
  2020-12-31 18:00:00 CST  America/Menominee
  2020-12-31 18:00:00 CST  America/Merida
  2020-12-31 18:00:00 CST  America/Mexico_City
  ...
  2020-12-31 18:00:00 CST  America/Winnipeg
```

```
2021-01-01 08:00:00 CST  Asia/Macau
2021-01-01 08:00:00 CST  Asia/Shanghai
2021-01-01 08:00:00 CST  Asia/Taipei
2020-12-31 18:00:00 CST  CST6CDT
```

Because `CST` may stand for *Central Standard Time*, *China Standard Time*, or `Cuba Standard Time`, you can see a 14 hour difference between most of the American and China's entries. In addition, Havana in Cuba has a 1 or 13 hour difference to them.

Note also that the output is significantly smaller when we search for `"CST"` on a day in the summer, because the US entries and Cuba's entry then switch to `"CDT"` (the corresponding daylight saving time). However, we still have some entries because, for example, China and Costa Rica do not have daylight saving time.

14.6.5 Custom Timezones

The chrono library allows you to use custom timezones. One common example is the need to have a time zone that has an offset from UTC that is not known until run time.

Here is an example, which supplies a custom time zone `OffsetZone` that can hold a UTC offset with minutes precision:[7]

lib/offsetzone.hpp

```
#include <chrono>
#include <iostream>
#include <type_traits>

class OffsetZone
{
  private:
    std::chrono::minutes offset;   // UTC offset
  public:
    explicit OffsetZone(std::chrono::minutes offs)
     : offset{offs} {
    }

    template <typename Duration>
    auto to_local(std::chrono::sys_time<Duration> tp) const {
      // define helper type for local time:
      using LT
       = std::chrono::local_time<std::common_type_t<Duration,
                                                    std::chrono::minutes>>;
      // convert to local time:
      return LT{(tp + offset).time_since_epoch()};
    }
```

[7] Thanks to Howard Hinnant for providing this example.

```
    template <class Duration>
    auto to_sys(std::chrono::local_time<Duration> tp) const {
      // define helper type for system time:
      using ST
       = std::chrono::sys_time<std::common_type_t<Duration,
                                                  std::chrono::minutes>>;
      // convert to system time:
      return ST{(tp - offset).time_since_epoch()};
    }

    template <class Duration>
    auto get_info(const std::chrono::sys_time<Duration>& tp) const {
      return std::chrono::sys_info{};
    }
};
```

As you can see, all you have to define are conversions between the local time and the system time.

You can use the time zone just like any other `time_zone` pointer:

lib/offsetzone.cpp

```
#include "offsetzone.hpp"
#include <iostream>

int main()
{
  using namespace std::literals;  // for h and min suffix

  // timezone with 3:45 offset:
  OffsetZone p3_45{3h + 45min};

  // convert now to timezone with offset:
  auto now = std::chrono::system_clock::now();
  std::chrono::zoned_time<decltype(now)::duration, OffsetZone*> zt{&p3_45, now};

  std::cout << "UTC:    " << zt.get_sys_time() << '\n';
  std::cout << "+3:45:  " << zt.get_local_time() << '\n';
  std::cout << zt << '\n';
}
```

The program might have the following output:

```
UTC:    2021-05-31 13:01:19.0938339
+3:45:  2021-05-31 16:46:19.0938339
```

14.7 Clocks in Detail

C++20 now has support for a couple of clocks. This section discusses their differences and how to use special clocks.

14.7.1 Clocks with a Specified Epoch

C++20 now provides the following clocks associated with an epoch (so that they define a unique point in time):

- The **system clock** is the clock of your operating system. Since C++20 it is specified to be Unix Time,[8] which counts time since the epoch January 1, 1970 00:00:00 UTC.

  Leap seconds are handled in a way that some seconds might take a little bit longer. Therefore, you never have hours with 61 seconds and all years with 365 days have the same number of 31,536,000 seconds.
- The **UTC clock** is the clock representing the *Coordinated Universal Time*, popularly known as *GMT* (*Greenwich Mean Time*), or *Zulu* time. Local time differs from UTC by the UTC offset of your timezone.

  It uses the same epoch as the system clock (January 1, 1970 00:00:00 UTC).

  Leap seconds are handled in a way that some minutes might have 61 seconds. For example, there is a timepoint `1972-06-30 23:59:60` because a leap second was added at the last minutes of June 1972. Therefore, years with 365 days might sometimes have 31,536,001 or even 31,536,002 seconds.
- The **GPS clock** uses the time of the *Global Positioning System*, which is the atomic time scale implemented by the atomic clocks in the GPS ground control stations and satellites. GPS time starts with the epoch of January 6, 1980 00:00:00 UTC.

  Each hour has 60 seconds, but GPS takes leap seconds into account by switching earlier to the next hour. As a result, GPS is more and more seconds ahead of UTC (or more and more seconds behind before 1980). For example, the time point `2021-01-01 00:00:00 UTC` is represented as GPS time point as `2021-01-01 00:00:18 GPS`. So, while writing this book GPS time points are 18 seconds ahead.

  All *GPS years* with 365 days (the difference between midnight of a GPS date and midnight of the GPS date one year later) have the same number of 31,536,000 seconds, but might be one or two seconds shorter than the "real" year.
- The **TAI clock** uses the *International Atomic Time*, which is the international atomic time scale based on a continuous counting of the SI second. TAI time starts with the epoch of January 1, 1958 00:00:00 UTC.

  TAI is currently ahead of UTC by 37 seconds. TAI is always ahead of GPS by 19 seconds. which is the atomic time scale implemented by the atomic clocks in the GPS ground control stations and satellites.

  As it is the case for GPS time, each hour has 60 seconds, and leap seconds are taken into account by switching earlier to the next hour. As a result, TAI is more and more seconds ahead of UTC, but always has a constant offset of 19 seconds to GPS. For example, the time point `2021-01-01 00:00:00 UTC` is represented as TAI time point as `2021-01-01 00:00:37 TAI`. So, while writing this book TAI time points are 37 seconds ahead.

[8] For details about Unix Time see, for example, `http://en.wikipedia.org/wiki/Unix_time`.

14.7.2 The Pseudo Clock `local_t`

As introduced already, there is a special clock of type `std::chrono::local_t`. This clock allows us to specify *local timepoints*, which have no time zone (not even UTC) yet. Its epoch is interpreted as "local time" which means you have to combine it with a timezone to know which point in time it represents.

`local_t` is a "pseudo clock" because it does not fulfill all requirements of clocks. In fact, it does not provide a member function `now()`:

```
auto now1 = std::chrono::local_t::now();    // ERROR: now() not provided
```

Instead you need a system clock timepoint and a time zone. Then, you can convert the timepoint to a local timepoint using a time zone such as the current timezone or UTC:

```
auto sysNow = chr::system_clock::now();                  // NOW as system timepoint
...
chr::local_time now1
  = chr::locate_zone("UTC")->to_local(sysNow);           // NOW in UTC as local time-
point
chr::local_time now2
  = chr::locate_zone("Asia/Tokyo")->to_local(sysNow);    // Now in Tokyo as local time-
point
chr::local_time now3
  = chr::current_zone()->to_local(sysNow);               // NOW locally as local timepoint
```

Another way to get the same result is to call `get_local_time()` for a zoned time (a timepoint with associated timezone):

```
chr::local_time now4 = chr::zoned_time{chr::current_zone(), sysNow}.get_local_time();
```

A different approach is to parse a string into a local timepoint:

```
chr::local_seconds tp;   //time_point<local_t, seconds>
std::istringstream{"2021-1-1 18:30"} >> chr::parse(std::string{"%F %R"}, tp);
```

Remember the subtle difference of local timepoints compared to other timepoints:

- A system/UTC/GPS/TAI timepoint represents a specific point in time. Applying it to a timezone will convert the time value it represents.
- A local timepoint represents a local time. Its global point in time becomes clear once it is combined with a time zone.

For example:

```
auto now = chr::current_zone()->to_local(chr::system_clock::now());
std::cout << now << '\n';
std::cout << "Berlin: " << chr::zoned_time("Europe/Berlin", now) << '\n';
std::cout << "Sydney: " << chr::zoned_time("Australia/Sydney", now) << '\n';
std::cout << "Cairo:  " << chr::zoned_time("Africa/Cairo", now) << '\n';
```

This applies the local time of `now` to three different timezones:

```
2021-04-14 08:59:31.640004000
Berlin: 2021-04-14 08:59:31.640004000 CEST
Sydney: 2021-04-14 08:59:31.640004000 AEST
```

```
Cairo:  2021-04-14 08:59:31.640004000 EET
```

Note that you cannot use conversion specifiers for the timezone when using a local timepoint:

```
chr::local_seconds tp;   // time_point<local_t, seconds>
...
std::cout << chr::format("{:%F %T %Z}\n", tp);   // throws std::format_error exception
std::cout << chr::format("{:%F %T}\n", tp);      // OK
```

14.7.3 Dealing with Leap Seconds

The previous discussion of clocks with a specified epoch already introduced basic aspects of dealing with leap seconds.

To make the handling of leap seconds more clear let us iterate over timepoints of a leap second using different clocks:[9]

lib/chronoclocks.cpp

```
#include <iostream>
#include <chrono>

int main()
{
  using namespace std::literals;
  namespace chr = std::chrono;

  auto tpUtc = chr::clock_cast<chr::utc_clock>(chr::sys_days{2017y/1/1} - 1000ms);
  for (auto end = tpUtc + 2500ms; tpUtc <= end; tpUtc += 200ms) {
    auto tpSys = chr::clock_cast<chr::system_clock>(tpUtc);
    auto tpGps = chr::clock_cast<chr::gps_clock>(tpUtc);
    auto tpTai = chr::clock_cast<chr::tai_clock>(tpUtc);
    std::cout << std::format("{:%F %T} SYS  ", tpSys);
    std::cout << std::format("{:%F %T %Z}  ",  tpUtc);
    std::cout << std::format("{:%F %T %Z}  ",  tpGps);
    std::cout << std::format("{:%F %T %Z}\n",  tpTai);
  }
}
```

The program has the following output:

```
2016-12-31 23:59:59.000 SYS  2016-12-31 23:59:59.000 UTC  2017-01-01 00:00:16.000 GPS  2017-01-01 00:00:35.000 TAI
2016-12-31 23:59:59.200 SYS  2016-12-31 23:59:59.200 UTC  2017-01-01 00:00:16.200 GPS  2017-01-01 00:00:35.200 TAI
2016-12-31 23:59:59.400 SYS  2016-12-31 23:59:59.400 UTC  2017-01-01 00:00:16.400 GPS  2017-01-01 00:00:35.400 TAI
2016-12-31 23:59:59.600 SYS  2016-12-31 23:59:59.600 UTC  2017-01-01 00:00:16.600 GPS  2017-01-01 00:00:35.600 TAI
2016-12-31 23:59:59.800 SYS  2016-12-31 23:59:59.800 UTC  2017-01-01 00:00:16.800 GPS  2017-01-01 00:00:35.800 TAI
```

[9] Thanks to Howard Hinnant for providing this example.

```
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.000 UTC  2017-01-01 00:00:17.000 GPS  2017-01-01 00:00:36.000 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.200 UTC  2017-01-01 00:00:17.200 GPS  2017-01-01 00:00:36.200 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.400 UTC  2017-01-01 00:00:17.400 GPS  2017-01-01 00:00:36.400 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.600 UTC  2017-01-01 00:00:17.600 GPS  2017-01-01 00:00:36.600 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.800 UTC  2017-01-01 00:00:17.800 GPS  2017-01-01 00:00:36.800 TAI
2017-01-01 00:00:00.000 SYS  2017-01-01 00:00:00.000 UTC  2017-01-01 00:00:18.000 GPS  2017-01-01 00:00:37.000 TAI
2017-01-01 00:00:00.200 SYS  2017-01-01 00:00:00.200 UTC  2017-01-01 00:00:18.200 GPS  2017-01-01 00:00:37.200 TAI
2017-01-01 00:00:00.400 SYS  2017-01-01 00:00:00.400 UTC  2017-01-01 00:00:18.400 GPS  2017-01-01 00:00:37.400 TAI
```

The leap second we look at is the last leap second we had when this book was written (we do not know ahead when leap seconds will happen in future). We print the timepoint with the corresponding timezone (for system timepoints we print `SYS` instead of its default timezone `UTC`). You can observe the following:

- During the leap second:
  - The UTC time uses the value 60 as seconds.
  - The system clock uses the last representable value of `sys_time` prior to the insertion of the leap second. That behavior is guaranteed by the C++ standard.
- Before the leap second:
  - The GPS time is 17 seconds ahead of the UTC time.
  - The TAI time is 36 seconds ahead of the UTC time (as always, 19 seconds ahead of the GPS time).
- After the leap second:
  - The GPS time is 18 seconds ahead of the UTC time.
  - The TAI time is 37 seconds ahead of the UTC time (still, 19 seconds ahead of the GPS time).

******* This chapter/section is at work *******

To do:

```
struct leap_second_info;
template<typename Duration>
  leap_second_info get_leap_second_info(const utc_time<Duration>& ut);
```

14.7.4 Conversions between Clocks

You can convert timepoint between clocks provided such a conversion makes sense. For this, a `clock_cast<>` is provided by the chrono library. It is defined in a way that you can only convert between the timepoints of clocks that have a specified stable epoch (`sys_time<>`, `utc_time<>`, `gps_time<>`, `tai_time<>`) and filesystem timepoints.

The cast needs the destination clock and you can optionally pass a different duration.

The following program creates the output of a UTC leap seconds to a couple of other clocks:

lib/chronoconv.cpp

```
#include <iostream>
#include <sstream>
#include <chrono>

int main()
{
```

```
  namespace chr = std::chrono;

  // initialize a utc_time<> with a leap second:
  chr::utc_time<chr::utc_clock::duration> tp;
  std::istringstream{"2015-6-30 23:59:60"}
    >> chr::parse(std::string{"%F %T"}, tp);

  // convert it to other clocks and print that out:
  auto tpUtc = chr::clock_cast<chr::utc_clock>(tp);
  std::cout << "utc_time:  " << std::format("{:%F %T %Z}", tpUtc) << '\n';
  auto tpSys = chr::clock_cast<chr::system_clock>(tp);
  std::cout << "sys_time:  " << std::format("{:%F %T %Z}", tpSys) << '\n';
  auto tpGps = chr::clock_cast<chr::gps_clock>(tp);
  std::cout << "gps_time:  " << std::format("{:%F %T %Z}", tpGps) << '\n';
  auto tpTai = chr::clock_cast<chr::tai_clock>(tp);
  std::cout << "tai_time:  " << std::format("{:%F %T %Z}", tpTai) << '\n';
  auto tpFile = chr::clock_cast<chr::file_clock>(tp);
  std::cout << "file_time: " << std::format("{:%F %T %Z}", tpFile) << '\n';
}
```

The program has the following output:

```
utc_time:  2015-06-30 23:59:60.0000000 UTC
sys_time:  2015-06-30 23:59:59.9999999 UTC
gps_time:  2015-07-01 00:00:16.0000000 GPS
tai_time:  2015-07-01 00:00:35.0000000 TAI
file_time: 2015-06-30 23:59:59.9999999 UTC
```

Note that for all of these clocks we have pseudo timezones for formatted output.

The rules of conversion are as follows:

- Any conversion from a local timepoint adds just the epoch. The time value remains the same.
- Conversion between UTC, GPS, and TAI timepoints add or subtract the necessary offsets.
- Conversion between UTC and system time do not change the time value except that for the time point of a UTC leap second the last time point ahead is used as system time.

Conversions from local timepoints to other clocks are also supported. However, for conversions to local timepoints you have to use `to_local()` (after converting to a system timepoint).

Conversions to or from timepoints of the `steady_clock` are not supported.

Clock Conversions Internally

Under the hood, `clock_cast<>` is a "two-hub spoke and wheel system."[10] The two hubs are `system_clock` and `utc_clock`. Every convertible clock has to convert to and from one of these hubs (but not both). The

[10] Thanks to Howard Hinnant for pointing this out.

conversions *to* the hubs are done with the `to_sys()` or `to_utc()` static member functions of the clocks. For conversions *from* the hubs `from_sys()` or `from_utc()` are provided. `clock_cast<>` strings these member functions together to convert from any clock to any other clock.

Clocks that cannot deal with leap seconds should convert to/from `system_clock` For example, `utc_clock` and `local_t` provide `to_sys()`.

Clocks that can deal leap seconds in some way (does not necessarily mean that they have a leap second value) should convert to/from `utc_clock`. This applies to `gps_clock` and `tai_clock`, because even though GPS and TAI do not have leap seconds, they have unique, bidirectional mappings to UTC leap seconds.

For the `file_clock` it is implementation defined whether conversions to/from `system_clock` or `utc_clock` are provided.

14.7.5 Dealing with the File Clock

The clock `std::chrono::`**`file_clock`** is the clock the filesystem library uses for timepoints of files system entries (files, directories, etc.). It is an implementation-specific clock type that reflects the resolution and range of time values of the filesystem.

For example, you can use the file clock to update the time of last access to a file as follows:

```
// touch file with path p (update last write access to file):
std::filesystem::last_write_time(p,
                                 std::chrono::file_clock::now());
```

You could use the clock of file system entries also in C++17 by using the type `std::filesystem::file_time_type`:

```
std::filesystem::last_write_time(p,
                                 std::filesystem::file_time_type::clock::now());
```

Since C++20, the filesystem type name `file_time_type` is defined as follows:

```
namespace std::filesystem {
  using file_time_type = chrono::time_point<chrono::file_clock>;
}
```

In C++17 it was only defined to use an unspecified *trivial-clock*.

For time points of the file system also the type `file_time` is defined now:

```
namespace std::chrono {
  template<typename Duration>
  using file_time = time_point<file_clock, Duration>;
}
```

A type like `file_seconds` (as the other clocks have) is not defined.

The new definition of the `file_time` type allows programmers now to portably convert filesystem timepoints to system timepoints. For example, you can print the time when a passed file was accessed last as follows:

```
void printFileAccess(const std::filesystem::path& p)
{
```

```
    std::cout << "\"" << p.string() << "\":\n";

    auto tpFile = std::filesystem::last_write_time(p);
    std::cout << std::format("  Last write access: {0:%F} {0:%X}\n", tpFile);

    auto diff = std::chrono::file_clock::now() - tpFile;
    auto diffSecs = std::chrono::round<std::chrono::seconds>(diff);
    std::cout << std::format("  It is {} old\n", diffSecs);
}
```

This code might output:

```
"chronoclocks.cpp":
  Last write access: 2021-07-12 16:50:08
  It is 18s old
```

If you would use the default output operator of time points, which prints subseconds according to the granularity of the file clock:

```
std::cout << "  Last write access: " << diffSecs << '\n';
```

the output might be as follows:

```
  Last write access: 2021-07-12 16:50:08.3680536
```

To deal with the file access time as system or local time, you have to use a `clock_cast<>()` (which internally might call the static `file_clock` member functions `to_sys()` or `to_utc()`). For example:

```
auto tpFile = std::filesystem::last_write_time(p);
auto tpSys = std::chrono::file_clock::to_sys(tpFile);
auto tpSys = std::chrono::clock_cast<std::chrono::system_clock>(tpFile);
```

14.8 Other New Chrono Features

In addition to what I described so far, the new chrono library adds the following features:

- To check whether a type is a clock, a new type trait `std::chrono::`**`is_clock<>`** is provided together with its corresponding variable template `std::chrono::`**`is_clock_v<>`**. For example:

  ```
  std::chrono::is_clock_v<std::chrono::system_clock>   //true
  std::chrono::is_clock_v<std::chrono::local_t>        //false
  ```

  The pseudo clock `local_t` yields `false` here, because it does not provide the member function `now()`.
- For durations and timepoints, `operator<=>` is defined.

******* This chapter/section is at work *******

```
constexpr bool is_am(const hours& h) noexcept;
constexpr bool is_pm(const hours& h) noexcept;
constexpr hours make12(const hours& h) noexcept;
constexpr hours make24(const hours& h, bool is_pm) noexcept;
```

```
class nonexistent_local_time;
class ambiguous_local_time;

enum class choose {earliest, latest};
class time_zone;
bool operator==(const time_zone& x, const time_zone& y) noexcept;
strong_ordering operator<=>(const time_zone& x, const time_zone& y) noexcept;

template<typename T> struct zoned_traits;
```

14.9 Afternotes

The chrono library was developed by Howard Hinnant. When the basic part for durations and timepoints was standardized with C++11, it was already the plan to extend this with support for dates, calendars, and time zones.

The chrono extension for C++20 was first proposed by Howard Hinnant in http://wg21.link/p0355r0. The finally accepted wording for this extension was formulated by Howard Hinnant and Tomasz Kaminski in http://wg21.link/p0355r7.

A few minor fixes were added by Howard Hinnant in http://wg21.link/p1466r3 and by Tomasz Kaminski in http://wg21.link/p1650r0. The finally accepted wording for the integration of the chrono library with the formatting library was formulated by Victor Zverovich, Daniela Engert, and Howard Hinnant in http://wg21.link/p1361r2.

After finishing C++20, a few fixes were applied to fix the chrono extensions of the C++20 standard:

- http://wg21.link/p2372 clarified the behavior of locale dependent formatted output.
- http://wg21.link/lwg3554 ensures that you can pass string literals as format to `parse()`.

Chapter 15

Coroutines

This chapter introduces *coroutines*.

Coroutines (invented in 1958 by Mel Conway) are functions you can suspend.

C++20 provides first basic support for coroutines by introducing some language features and data types to deal with them. However, some glue code has to be written to be able to deal with coroutines, which makes using coroutines very flexible but also requires some effort even in simple cases. The plan is to provide standard types and functions to deal with coroutines later with C++23 in the C++ standard library.

15.1 What Are Coroutines?

While ordinary functions (or procedures) are called and then run from begin to the end or until a return statement is reached, coroutines can run in multiple steps. At certain moments you can *suspend* a coroutine which means that the function pauses its computation until it gets *resumed*. You might suspend it because the function has to wait for something, there are other (more important things to do), or you have an intermediate result to yield to the caller.

This behavior it a bit like having multiple function calls. However, the key benefit is that a coroutine keeps its state from when it gets suspended to when it gets resumed.

Note that this concept is independent from from the question where to coroutine and a function calling it run. This is not an asynchronous event mechanism and you do not necessarily need multi-threading. It is an orthogonal feature which however might make sense with the other features.

Figure XXX demonstrates the basic API when dealing with coroutines:

1. A function might call a coroutine and wait until it gets suspended for the first time.

   The coroutine might perform some initial code just to finish the setup or to start performing some computation and then suspend for the first time.
2. The caller gets a *coroutine interface object* to deal with the running coroutine.

   With this object the caller gets a flexible interface to resume a suspended coroutine, to deal with intermediate or final values, and to handle special situations such as exceptions. The generator might just provide a function `resume()` to resume the coroutine or it might provide an iterator-based interface,

where `begin()` and `operator++` resume and ask for the next intermediate result and `operator*` grants access to it.

3. A coroutine might finally return something to the caller, which again the caller gets via the coroutine interface.

A few things are special in C++ regarding coroutines:

- Coroutines are stackless. When suspended their state is stored in an object separately from the stack so that it can be resumed in a totally different context (in a different call stack, in another thread, etc.).
- Coroutines are defined just by using one of the following keywords in a function:
  - **`co_await`**
  - **`co_yield`**
  - **`co_return`**
- With these keywords a coroutine returns a handle to the caller that can be used to deal with the coroutine. Depending on the purpose and use of the coroutine, that handle can represent a running task that suspends or switches context from time to time, a generator yielding values from time to time, or a factory returning one or more values lazily and on demand.

15.2 A First Coroutine Example

Let us look at a first coroutine example.

15.2.1 Defining a Coroutine

Here is a first coroutine that from time to time suspends its execution:

coro/coro.hpp

```
#include <iostream>
#include <coroutine>      // for std::suspend_always()
#include "corotask.hpp"   // for CoroTask

CoroTask sayHello()
{
  std::cout << "Hello" << '\n';
  co_await std::suspend_always();        // SUSPEND
  std::cout << "World" << '\n';
  co_await std::suspend_always();        // SUSPEND
  std::cout << "!" << '\n';
}
```

What we define here is a function that calls three statements to print a string. However, between the statement there is a `co_await` expression, which suspends the coroutine and blocks until it gets resumed.

The exact behavior of the suspend call is defined by the expression after `co_await`. It allows an expression to accept the request for suspension (giving control back to the caller), requesting the request to suspend

(resume immediately), or make this decision depending on the situation, immediately or with a delay, or by switching contexts (such as starting a new thread to continue there).

For the moment we use a default-constructed object of type `std::suspend_always`, which accepts the suspension and gives control back to the caller.

The coroutine has no return statement but returns a type: `CoroTask` (note that we cannot declare it with `auto`). This is because we need an interface and some glue code to be able to deal with the coroutine (such as resuming it). In C++20, you have to use a user-defined type here, which we will discuss later (the plan is to have some standard types here in C++23 to deal with coroutines).

The code calling the coroutine might look very simple:

coro/coro.cpp

```
#include <iostream>
#include <thread>
#include <chrono>
#include "coro.hpp"

int main()
{
  using namespace std::literals;

  // start coroutine:
  CoroTask sayHelloTask = sayHello();
  std::cout << "coroutine sayHello() started\n";

  // loop to resume the coroutine until it is done:
  while (sayHelloTask.resume()) {         // resume
    std::this_thread::sleep_for(500ms);
  }
}
```

We call the coroutine like calling a function:

```
CoroTask sayHelloTask = sayHello();
```

However, this call does not block until `sayHello()` is done. The call returns after the coroutine is initialized (which might or might not include running all statements up to the first suspension). The call returns what we have declared as the return type of the coroutine: a handle of type `CoroTask` to deal with the coroutine whenever we get back control. (here, we could also use `auto` to declare it). We store this return value in `sayHelloTask` for later use.

The exact behavior of the initialization and the interface of the coroutine depends on class `CoroTask`. It might just start the coroutine or provide a context with some initializations such as opening a file or starting a thread. And as written it defines whether the coroutine is immediately suspended or whether it starts to perform the statement until it reaches the first suspension. We will see that in this case it only performs some minimal initializations, so that this call does *not* perform any statement of `sayHello()`. We kind of suspend the coroutine immediately, which means nothing is printed yet when we call `sayHello()`.

Afterwards, we start a loop that again and again resumes the coroutine after it was suspended:

```
// loop to resume the coroutine until it is done:
while (sayHelloTask.resume()) {         // resume
  std::this_thread::sleep_for(500ms);
}
```

As you can see, class `CoroTask` provides a member function `resume()`, which resumes the coroutine. Each call of `resume()` lets the coroutine continue to run until the next suspension or its end is reached. So with the loop we

- resume to print `"Hello"` and then suspend and sleep 500 milliseconds
- resume to print `"World"` and then suspend and sleep 500 milliseconds
- resume to print `"!"`

Note that there is no asynchronous communication or data flow. It is more that `resume()` is like a function call for the next part of `sayHello()`.

The loop ends when we have reached the end of the coroutine. For this purpose, `resume()` is implemented so that it returns whether the coroutine is not done (yet). Thus, while `resume()` returns `true` we continue the loop (sleep a bit and then resume it again).

The output of the program is as follows:

```
coroutine sayHello() started
Hello
World
!
```

Again, note that the output depends on how class `CoroTask` (which specifies several details for dealing with the coroutine) is implemented.

Implementing the Interface to Coroutines

I wrote a couple of times that in our example, class `CoroTask` plays an important role to deal with the coroutine. In fact it brings together a couple of requirements to deal with coroutines and provides the interface for the caller of the coroutine. Let us elaborate on that in detail.

To deal with coroutines in C++ we need two things:

- A ***promise type***

  This type is used to define certain customization points for dealing with a coroutine. Specific member functions define callbacks that are called on certain situations. The basic customization points are:
  - How to create a coroutine handle that can keep its state (from suspension to resumption)
  - Whether the coroutine should be suspended at the beginning (the initial suspend point)
  - Whether the coroutine should be suspended at the end (the final suspend point)
  - How to deal with unhandled exceptions
  - How to deal with intermediate and final results

  The type can also define members to store intermediate and final results so that the caller of the coroutine can access them.
- An internal ***coroutine handle*** of type `std::coroutine_handle<>`

This object is created when calling a coroutine (using one of the standard callbacks of the promise type above) and provides the low-level interfaces to resume a coroutine as well as to check whether it is done.

The usual purpose of the type dealing with the return type of a coroutine is to bring these requirements together:

- It has to define which promise type is used (it is usually defined as type member `promise_type`).
- It has to define where the coroutine handle is stored (it is usually defined as a data member).
- It has to provide the interface for the caller to deal with the coroutine (the member function `resume()` in this case).

Here is a basic promise type we can use:

coro/coropromise.hpp

```
#include <iostream>
#include <coroutine>
#include <exception>  // for terminate()

// API to control the basic handling of a coroutine
// and to deal with a result or an exception
template <typename CoroType>
struct CoroPromise {
  auto get_return_object() {          // create the coroutine handle
    std::cout << "- promise: get_return_object()\n";
    return std::coroutine_handle<CoroPromise<CoroType>>::from_promise(*this);
  }
  auto initial_suspend() {            // start immediately?
    std::cout << "- promise: initial_suspend()\n";
    return std::suspend_always{};     // - suspend immediately
  }
  auto final_suspend() noexcept {     // clean-ups / postprocessing
    std::cout << "- promise: final_suspend()\n";
    return std::suspend_always{};     // - suspend at the end
  }
  void unhandled_exception() {        // deal with exceptions
    std::terminate();                 // - terminate the program
  }
  void return_void() {                // deal with the end or with return;
    std::cout << "- promise: return_void()\n";
  }
};
```

It just defines more or less the minimum interface the promise type of such a coroutine needs:

- `get_return_object()` is called to initialize the coroutine handle. You have to create it using the type of the promise (we will pass `CoroTask` as `CoroType` so that the handle knows the promise type and the external type the caller of the coroutine uses).

- `initial_suspend()` defines whether the coroutine should immediately start or be suspended. In this case we request to suspend it immediately (more complex logic is possible here).
- `final_suspend()` defines whether the coroutine should be finally suspended. Here we specify to do so, which usually is the right thing to do. Please note that this member function has to guarantee not to throw exceptions.
- `unhandled_exception()` defines how to deal with exceptions not handled locally in the coroutine. Here we specify that this results in an abnormal termination of the program.
- `return_void()` defines anything when reaching the end or a `return;` statement. By having this member function the coroutine should never return a value. In case the coroutine yields or returns data, you have to use another member function instead.

Now let us look how class `CoroTask` combines the promise type with all other things we need to deal with the coroutine:

coro/corotask.hpp

```
#include <iostream>
#include "coropromise.hpp"

// handle to deal with a simple coroutine task
// - providing resume() to resume it
class CoroTask {
 public:
  // helper type for state and customization:
  using promise_type = CoroPromise<CoroTask>;

 private:
  // internal handle to allocated state:
  std::coroutine_handle<promise_type> coroHdl;

 public:
  // constructor to initialize the coroutine:
  CoroTask(auto h) : coroHdl{h} {
    std::cout << "- CoroTask: construct\n";
  }
  ~CoroTask() {
    std::cout << "- CoroTask: destruct\n";
    if (coroHdl) {
      coroHdl.destroy();
    }
  }
  // don't copy or move:
  CoroTask(const CoroTask&) = delete;
  CoroTask& operator=(const CoroTask&) = delete;

  // API to resume the coroutine
```

```
  // - returns whether there is still something to process
  bool resume() const {
    std::cout << "- CoroTask: resume()\n";
    if (!coroHdl) {
      return false;       // nothing (more) to process
    }
    coroHdl.resume();     // RESUME (just coroHdl() is also possible)
    return !coroHdl.done();
  }
};
```

The first thing we have to do is to define the promise type of the coroutine. As written, it has to be provided as a type member with the name `promise_type`:

```
class CoroTask {
 public:
  // helper type for state and customization:
  using promise_type = CoroPromise<CoroTask>;
  ...
};
```

As you see, we pass our own type to the promise type so that `get_return_object()` there can create an internal coroutine handle using this type.

Then, we need a place for the internal coroutine handle, which is usually just a private member:

```
class CoroTask {
  ...
 private:
  // internal handle to allocated state:
  std::coroutine_handle<promise_type> coroHdl;
 ...
};
```

As you can see, the type is parameterized with the promise type (which fits with the type `get_return_object()` of the promise type returns).

Then, we need a constructor and a destructor, which are called when the coroutine is called and when the lifetime of the return value ends. The constructor gets the internal handle and can store it at the right place. The destructor should simply destroy the internal handle.

Here, for simplicity, we also disable copying and moving. Providing them is possible, but you have to be careful to deal with the resources right.

Finally, we define our only interface for the caller, `resume()`:

```
class CoroTask {
  ...
  bool resume() const {
    std::cout << "- Task: resume()\n";
    if (!coroHdl) {
      return false;       // nothing (more) to process
```

```
    }
    coroHdl.resume();    // RESUME (just coroHdl() is also possible)
    return !coroHdl.done();
  }
};
```

`resume()` is simply propagated to the internal coroutine handle. By calling `resume()` there, we continue the coroutine.

In addition, a few things have to be done:

- We check whether there is a coroutine handle at all (just a safety net in case we have move semantics and the internal handle was moved away).
- We return whether it makes sense to resume the coroutine again. Here we use the member function `done()` of the internal coroutine handle, which yields whether the coroutine is done.

As you can see, I have provided print statements inside most of the additional functions. Therefore, with the output you can see now how data flows:

```
- promise: get_return_object()
- promise: initial_suspend()
- CoroTask: construct
coroutine sayHello() started
- CoroTask: resume()
Hello
- CoroTask: resume()
World
- CoroTask: resume()
!
- promise: return_void()
- promise: final_suspend()
- CoroTask: destruct
```

The first thing that happens when we call a coroutine is that the internal coroutine handle is created (with the help of `get_return_object()`). Then, `initial_suspend()` is called to see whether it should be immediately suspended, which is the case here. So, we have to return a `CoroTask`, which is created here. Later, whenever we call `resume()` the coroutine resumes so that we process our statements up to the next suspension or the end. At the end we call `return_void()` and call the function for the final suspension, `final_suspend()`. At the end of the lifetime of the `CoroTask` its destructor is called, which destroys the coroutine handle.

We could also use a promise type where we do not initially suspend the coroutine instead:

```
template<typename CoroType>
struct CoroPromise {
  ...
  auto initial_suspend() {          // initializations
    return std::suspend_never{};    // - don't suspend initially
  }
  ...
```

```
};
```

In this case we would get the following output:

```
- promise: get_return_object()
- promise: initial_suspend()
Hello
- CoroTask: construct
coroutine sayHello() started
- CoroTask: resume()
World
- CoroTask: resume()
!
- promise: return_void()
- promise: final_suspend()
- CoroTask: destruct
```

Again we initially start create the internal coroutine handle. However, `initial_suspend()` does not suspend. Therefore, the initial call of the coroutine now executes the first statements of the coroutine before a first `co_await` suspends it for the first time. This is the moment where we create the `CoroTask` object and return it. Then as before we loop over resumptions.

The code of the promise types is pretty straight forward and usually the promise type is specific for a specific coroutine return type. For that reason, both is often implemented together. It might look for example as follows:

coro/corotask1.hpp

```
#include <iostream>
#include <coroutine>
#include <exception>  // for terminate()

// handle to deal with a simple coroutine task
// - providing resume() to resume it
class CoroTask {
 public:
  // helper type for state and customization:
  struct promise_type {
    auto get_return_object() {         // create the coroutine handle
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    auto initial_suspend() {           // start immediately?
      return std::suspend_always{};    // - suspend immediately
    }
    auto final_suspend() noexcept {    // clean-ups / postprocessing
      return std::suspend_always{};    // - suspend at the end
    }
    void unhandled_exception() {       // deal with exceptions
```

```
      std::terminate();                 // - terminate the program
    }
    void return_void() {                // deal with the end or with return;
    }
  };

 private:
  // internal handle to allocated state:
  std::coroutine_handle<promise_type> coroHdl;

 public:
  // constructor to initialize the coroutine:
  CoroTask(auto h) : coroHdl{h} {
    std::cout << "- CoroTask: construct\n";
  }
  ~CoroTask() {
    std::cout << "- CoroTask: destruct\n";
    if (coroHdl) {
      coroHdl.destroy();
    }
  }
  // don't copy or move:
  CoroTask(const CoroTask&) = delete;
  CoroTask& operator=(const CoroTask&) = delete;

  // API to resume the coroutine
  // - returns whether there is still something to process
  bool resume() const {
    std::cout << "- CoroTask: resume()\n";
    if (!coroHdl) {
      return false;       // nothing (more) to process
    }
    coroHdl.resume();     // RESUME (just coroHdl() is also possible)
    return !coroHdl.done();
  }
};
```

15.3 Further Coroutine Examples

After introducing a first example using `co_await`, we should also introduce the other two keywords for coroutines:

- `co_yield` allows coroutines to yield a value each time it is suspended.
- `co_return` allows coroutines to return a value at their end.

15.3.1 Coroutine with `co_yield`

By using `co_yield` a coroutine can yield intermediate results on suspension.

Consider the following example:

coro/coyield.cpp

```
#include "intgen.hpp"
#include <iostream>
#include <vector>
#include <thread>

template <typename T>
IntGen loopOver(const T& coll)
{
  // coroutine that iterates over the elements of a collection:
  for (int elem : coll) {
    std::cout << "- yield " << elem  << '\n';
    co_yield elem;  // calls yield_value(elem) on promise
    std::cout << "- resume\n";
  }
}

int main()
{
  using namespace std::literals;

  // define generator that yields the elements of a collection:
  std::vector<int> coll{0, 8, 15, 33, 42, 77};
  IntGen gen = loopOver(coll);

  // loop to resume the coroutine until there is no more value:
  std::cout << "start loop:\n";
  while (gen.resume()) {    // resume until we have the next value
    std::cout << "main(): value: " << gen.getValue() << '\n';
    std::this_thread::sleep_for(1s);
  }
}
```

In this program, `main()` creates a container of integers and lets the coroutine `loopOver()` iterate over its values. Whenever we call `resume()` for the return value of the coroutine, the integer generator `IntGen`, it "computes" and yields the next element. However, `resume()` does not return it. To access the next value, `getValue()` is provided.

Again, the return type of the coroutine, class `IntGen` defines basic customization points to deal with the coroutine and the interface to deal with it. It may be defined as follows:

coro/intgen.hpp

```
#include <coroutine>
#include <exception>  // for terminate()

class IntGen {
 public:
  // customization points:
  struct promise_type {
    int currentValue;                 // last value from co_yield

    auto yield_value(int value) {     // reaction on co_yield
      currentValue = value;           // - store value locally
      return std::suspend_always{};   // - suspend coroutine
    }

    // the usual callbacks:
    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    auto initial_suspend() { return std::suspend_always{}; }
    auto final_suspend() noexcept { return std::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
    void return_void() { }
  };

 private:
  // internal coroutine handle:
  std::coroutine_handle<promise_type> coroHdl;

 public:
  // constructors and destructor:
  IntGen(auto h) : coroHdl{h} {
  }
  ~IntGen() {
    if (coroHdl) {
      coroHdl.destroy();
    }
  }

  // no copying, but moving is supported:
  IntGen(IntGen const&) = delete;
  IntGen(IntGen&& rhs)
   : coroHdl(rhs.coroHdl) {
       rhs.coroHdl = nullptr;
```

```
  }
  IntGen& operator=(IntGen const&) = delete;

  // API:
  // - resume() to resume the coroutine
  // - getValue() to get the last value from co_yield
  bool resume() const {
    if (!coroHdl) {
      return false;     // nothing (more) to process
    }
    coroHdl.resume();      // RESUME
    return !coroHdl.done();
  }

  int getValue() const {
    return coroHdl.promise().currentValue;
  }
};
```

The definition follows the general principles of coroutine interfaces.

However, this time we have support for intermediate results. When the coroutine reaches `co_yield` it suspends the coroutine providing the value of the expression behind `co_yield`:

```
co_yield elem;   // calls yield_value(elem) on promise
```

This calls `yield_value()` on the promise type of the coroutine, which can define how to handle this intermediate result:

```
struct promise_type {
  int currentValue;                 // last value from co_yield

  auto yield_value(int value) {     // reaction on co_yield
    currentValue = value;           // - store value locally
    return std::suspend_always{};   // - suspend coroutine
  }
  ...
};
```

In this case, the promise defines a member for so that `yield_value()` can store the value there and then conforms to really suspend the coroutine.

The coroutine interface then just returns this value whenever `getValue()` is called:

```
class IntGen {
 public:
  ...
  int getValue() const {
    return coroHdl.promise().currentValue;
  }
```

```
};
```

Therefore, the output of the whole program is as follows:

```
start loop:
- yield 0
main(): value: 0
- resume
- yield 8
main(): value: 8
- resume
- yield 15
main(): value: 15
- resume
- yield 33
main(): value: 33
- resume
- yield 42
main(): value: 42
- resume
- yield 77
main(): value: 77
- resume
```

We could also map the coroutine to a coroutine interface that acts like a range (providing an API to iterate over the intermediate values):

coro/coyield2.cpp

```
#include "intrange.hpp"
#include <iostream>
#include <vector>
#include <thread>

template <typename T>
IntRange loopOver(const T& coll)
{
  // coroutine that iterates over the elements of a collection:
  for (auto elem : coll) {
    std::cout << "- suspend" << '\n';
    co_yield elem;  // calls yield_value(elem) on promise
    std::cout << "- resume" << '\n';
  }
}

int main()
{
  using namespace std::literals;
```

```
  // define generator that yields the elements of a collection:
  std::vector<int> coll{0, 8, 15, 33, 42, 77};
  IntRange gen = loopOver(coll);

  // loop to resume the coroutine until there is no more value:
  std::cout << "start loop:\n";
  for (const auto& val : gen) {
    std::cout << "value: " << val << '\n';
    std::this_thread::sleep_for(1s);
  }
}
```

Here, we use a range-based `for` loop to iterate over the values:

```
IntRange gen = loopOver(coll);

for (const auto& val : gen) {
  ...
}
```

For that interface, the coroutine interface might be defined as follows:

coro/intrange.hpp

```
#include <coroutine>
#include <exception>  // for terminate()
#include <cassert>    // for assert()

class IntRange {
 public:
  // customization points:
  struct promise_type {
    int currentValue;                 // last value from co_yield

    auto yield_value(int value) {     // reaction on co_yield
      currentValue = value;           // - store value locally
      return std::suspend_always{};   // - suspend coroutine
    }

    // the usual callbacks:
    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    auto initial_suspend() { return std::suspend_always{}; }
    auto final_suspend() noexcept { return std::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
```

```
    void return_void() { }
  };

private:
  // internal coroutine handle:
  std::coroutine_handle<promise_type> coroHdl;

public:
  // constructors and destructor:
  IntRange(auto h) : coroHdl{h} {
  }
  ~IntRange() {
    if (coroHdl) {
      coroHdl.destroy();
    }
  }

  // no copying, but moving is supported:
  IntRange(IntRange const&) = delete;
  IntRange(IntRange&& rhs)
   : coroHdl(rhs.coroHdl) {
       rhs.coroHdl = nullptr;
  }
  IntRange& operator=(IntRange const&) = delete;

  // iterator interface with begin() and end()
  struct iterator {
    std::coroutine_handle<promise_type> hdl;  // nullptr on end
    iterator(auto p) : hdl{p} {
    }
    void getNext() {
      if (hdl) {
        hdl.resume();          // RESUME
        if (hdl.done()) {
          hdl = nullptr;
        }
      }
    }
    int operator*() const {
      assert(hdl != nullptr);
      return hdl.promise().currentValue;
    }
    iterator operator++() {
      getNext();               // resume for next value
      return *this;
```

```
    }
    bool operator== (const iterator& i) const = default;
  };
  iterator begin() const {
    if (!coroHdl || coroHdl.done()) {
      return iterator{nullptr};
    }
    iterator itor{coroHdl};  // initialize iterator
    itor.getNext();          // resume for first value
    return itor;
  }
  iterator end() const {
    return iterator{nullptr};
  }
};
```

15.3.2 Coroutine with `co_return`

By using `co_return` a coroutine can return a result at its end to the caller.

Consider the following example:

coro/coreturn.cpp

```
#include <iostream>
#include <thread>
#include <chrono>
#include <coroutine>        // for std::suspend_always()
#include "stringtask.hpp"   // for StringTask

StringTask computeInSteps()
{
  std::string ret;
  ret += "Hello";
  co_await std::suspend_always();         // SUSPEND
  ret += " World";
  co_await std::suspend_always();         // SUSPEND
  ret += "!";
  co_return ret;
}

int main()
{
  using namespace std::literals;

  // start coroutine:
```

```
  StringTask task = computeInSteps();

  // loop to resume the coroutine until it is done:
  while (task.resume()) {                  // RESUME
    std::this_thread::sleep_for(500ms);
  }

  // print return value of coroutine:
  std::cout << "result: " << task.getResult() << '\n';
}
```

In this program, `main()` starts a coroutine `computeInSteps()`, which suspends from time to time and returns a string at the end. As usual, we have a coroutine interface class, `StringTask` that provides the interface to `resume()` the coroutine whenever it is suspended. In addition, it provides `getResult()` to ask for the return value of the coroutine after it is done:

coro/stringtask.hpp

```
#include <string>
#include <coroutine>
#include <exception>  // for terminate()

class StringTask {
 public:
  // customization points:
  struct promise_type {
    std::string result;                    // value from co_return

    void return_value(const auto& value) { // reaction on co_return
      result = value;                      // - store value locally
    }

    // the usual callbacks:
    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    auto initial_suspend() { return std::suspend_always{}; }
    auto final_suspend() noexcept { return std::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
  };

 private:
  // internal coroutine handle:
  std::coroutine_handle<promise_type> coroHdl;

 public:
```

```
  // constructors and destructor:
  StringTask(auto h) : coroHdl{h} {
  }
  ~StringTask() {
    if (coroHdl) {
      coroHdl.destroy();
    }
  }

  // no copying or moving is supported:
  StringTask(StringTask const&) = delete;
  StringTask& operator=(StringTask const&) = delete;

  // API:
  // - resume() to resume the coroutine
  // - getValue() to get the last value from co_yield
  bool resume() const {
    if (!coroHdl) {
      return false;      // nothing (more) to process
    }
    coroHdl.resume();      // RESUME
    return !coroHdl.done();
  }

  std::string getResult() const {
    return coroHdl.promise().result;
  }
};
```

Again, the definition follows the general principles of coroutine interfaces.

However, this time we have support for a return value. Therefore, in the promise type the customization point `return_void()` is no longer provided. Instead, `return_value()` is provided which is called when the coroutine reaches the `co_return` expression:

```
struct promise_type {
  std::string result;                    // value from co_return

  void return_value(const auto& value) { // reaction on co_return
    result = value;                      // - store value locally
  }
  ...
};
```

The coroutine interface then just returns this value whenever `getResult()` is called:

```
class StringTask {
 public:
```

```
    ...
    std::string getResult() const {
      return coroHdl.promise().result;
    }
  };
```

15.4 Coroutines in Detail

Coroutines have the following attributes and constraints:

- They cannot be used as `constexpr` or `consteval` functions.

******* This chapter/section is at work *******

Open

******* This chapter/section is at work *******

15.5 Afternotes

The request to support coroutines was first proposed as a pure library extension by Oliver Kowalke and Nat Goodspeed in http://wg21.link/n3708. Due to the complexity of the feature a *Coroutine TS* (experimental technical specification) was established with http://wg21.link/n4403 to work on details. The wording to finally merge coroutines into the C++20 standard was formulated by Gor Nishanov in http://wg21.link/p0912r5

Chapter 16

`std::jthread` and Stop Tokens

C++20 introduces a new type to represent with threads: `std::jthread`. It fixes a severe design problem of `std::thread` and benefits from a new feature to signal cancellation.

This chapter explains both the feature to signal cancellation in asynchronous scenarios and the new class `std::jthread`.

16.1 Motivation for `std::jthread`

C++11 introduced a type `std::thread` to map one-to-one with threads provided by the operating system. However, the type has a severe design flaw: it is not an RAII type.

16.1.1 The Problem of `std::thread`

`std::thread` requires that at the end of its lifetime representing a running thread either `join()` (to wait for the end for the running thread) or `detach()` (to let it run in the background) is called. If neither has been called, the destructor immediately causes an abnormal program termination with a core dump.

Thus, the following is usually an error (unless an abnormal program termination is not a problem):

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  ...   // neither t.join() nor t.detach() called

} // std::terminate() called
```

When the destructor of `t` representing the running thread is called without having called `join()` or `detach()`, the program calls `std::terminate()`, which calls `std::abort()` and usually causes a core dump.

However, even when calling `join()` to wait for the running thread to end, you still have a significant problem:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  ...      // calls std::terminate() on exception
  // wait for tasks to finish:
  t.join();
  ...
}
```

As `t.join()` is not called on an exception (or any other reason to leave the scope without reaching the call to `join()`), this code may also cause an abnormal program termination.

The way you have to program this is as follows:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  try {
    ...
  }
  catch (...) { // if we have an exception
    // clean-up the started thread:
    t.join();    // wait for thread (blocks until done)
    throw;       // and rethrow the caught exception
  }
  // wait for thread to finish:
  t.join();
  ...
}
```

Here, we react to an exception without resolving it by making sure that `join()` is called when we leave the scope. Unfortunately, this might block (forever). However, calling `detach()` is also a problem, because the thread continues in the background of the program using CPU time and resources which might now become destroyed.

Using multiple threads in more complex contexts, the problem gets even worse and creates really nasty code. For example, just when starting two threads, you have to program something like this:

```
void foo()
{
  ...
  // start thread calling task1() with name and val as arguments:
  std::thread t1{task1, name, val};
  std::thread t2;
  try {
```

```
    // start thread calling task2() with name and val as arguments:
    t2 = std::thread{task2, name, val};
    ...
  }
  catch (...) { // if we have an exception
    // clean-up the started threads:
    t1.join();      // wait for first thread
    if (t2.joinable()) {  // might or might no be started
      t2.join();  // wait for second thread
    }
    throw;          // and rethrow the caught exception
  }
  // wait for threads to finish:
  t1.join();
  t2.join();
  ...
}
```

On one hand after starting the first thread, calling the second thread might have already thrown, so this has to happen in the `try` clause. On the other hand we want to wait for them in the same scope, so we have to forward declare the second thread. And when an exception occurs we have to check whether the second thread was started or not, because joining a thread that was not started causes another exception.

And because calling `join()` for both threads might take significant time (or even take forever), you risk that on an exception you have no fast reaction. Note that you cannot "kill" started threads (these are not processes). A thread can only end by ending itself or ending the program as a whole.

So, before calling `join()` you better make sure that the thread you wait for more or less immediately cancels its execution. However, with `std::thread` there is no mechanism for that. You have to implement the request for cancellation in a way that, for example, the started thread checks from time to time a flag that is raised by the caller.

16.1.2 Using `std::jthread`

`std::jthread` solves these problems. First, it is an RAII type. The destructor of the type calls `join()` if the thread is joinable (the "j" stands for "joining"). Thus, the complex code above simply looks as follows to have the same effect:

```
void foo()
{
  ...
  // start thread calling task1() with name and val as arguments:
  std::thread t1{task1, name, val};
  // start thread calling task2() with name and val as arguments:
  t2 = std::thread{task2, name, val};
  ...
  // wait for threads to finish:
  t1.join();
```

```
    t2.join();
    ...
}
```

Just by using `std::jthread` instead of `std::thread`, the danger of causing an abnormal program termination is gone although no exception handling is implemented. To support this, class `std::jthread` provides the same API as `std::thread` including:

- Using the same header file `<thread>`
- Also returning a `std::thread::id` when calling `get_id()` (the type `std::jthread::id` is just the same)
- Also having the `static` member `hardware_concurrency()`

That means: **just replace `std::thread` by `std::jthread` and recompile** and your code gets safer (if you do not handle exceptions correctly yet).[1]

16.1.3 Stop Tokens and Stop Callbacks

Class `std::jthread` does even more: it provides a mechanism to signal cancellation, which is used by the destructor in case it is calling `join()`. However, the mechanism needs support by the callable of the started thread:

- If the callable started as a thread just has parameters for all passed arguments, the request to end would be ignored:

```
void task (std::string s, double value)
{
  ...   // join() waits until this code ends
}
```

- To react to the request, the callable has to add a new optional first parameter of type `std::stop_token` and check from time to time whether a stop was requested:

```
void task (std::stop_token st,
           std::string s, double value)
{
  while (!st.stop_requested()) {   // stop requested (e.g., by the destructor)?
    ...   // ensure we check from time to time
  }
}
```

That means, `std::jthread` provides a *cooperative* mechanism to signal that a thread should no longer run. It is "cooperative" because the mechanism does not kill the running thread (again, in general you cannot kill threads because it might not be supported at all or can easily leave your program in a corrupt state). Instead, the started thread has to check from time to time.

You can also manually request a started thread to stop. For example:

[1] You might wonder, why we did not just fix `std::thread` instead of introducing a new type `std::jthread`. The reason is backward compatibility. There might be a few applications that want to terminate the program when leaving the scope of a running thread. And for some new functionality discussed next we would also break binary compatibility.

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::jthread t{task, name, val};
  ...
    if (...) {
      t.request_stop();   // explicitly request task() to stop its execution
    }
  ...
  // wait for thread to finish:
  t.join();
  ...
}
```

There is also another way to react to a stop request: you can register callbacks for a stop token, which are automatically called when a stop is requested. For example:

```
void task (std::stop_token st,
           std::string s, double value)
{
  std::stop_callback cb{st, [] {
                              ...   // called on a stop request
                            }};
  ...
}
```

In this case, a request to stop the thread performing `task()` (whether it is an explicit call of `request_stop()` or caused by the destructor) calls the lambda you have registered as stop callback. The callback is called by the thread requesting the stop.

At the end of the lifetime of `cb` the destructor unregisters the callback automatically so that it will no longer be called if a stop is signaled afterwards. You can register an arbitrary number of callables (functions, function objects, or lambdas) that way.

Note that the stop mechanism is more flexible than it looks at first:

- You can pass around handles to request a stop and tokens to check for a requested stop.
- There is support for condition variables so that a signaled stop can interrupt a wait there.
- You can use the mechanism to request and check for stops independent from `std::jthread`.

In the following sections we discuss the mechanism to request stops and its application in threads.

16.2 Stop Sources and Stop Tokens

C++20 provides a new basic library with a mechanism to asynchronously signal stop, and provides various ways to react to this signal.

The basic mechanism is as follows:

- The C++20 standard library allows us to establish a *shared stop state*. By default, a stop is not signaled.

- *Stop sources* of type `std::stop_source` can *request* a stop its associated shared stop state.
- *Stop tokens* of type `std::stop_token` can be used to react to a stop request for its associated shared stop state. You can actively poll whether there is a stop requested or register a callback of type `std::stop_callback` which will be called when a stop is/was requested.
- Once a stop request has been made, it cannot be withdrawn (a subsequent stop request has no effect).
- Stop sources and stop tokens can be copied and moved around to be able to signal or react to a stop at multiple locations. Copying a source or token is relatively cheap so that you usually pass them by value to avoid any lifetime issues (copying is not as cheap as passing an integral value or raw pointer around; it is more like passing a shared pointer; so, when passing them often to a sub-function it might better to pass them by reference).
- The mechanism is thread safe and can be used in concurrent situations: calls to request a stop check for a requested stop, calls to register or unregister callbacks are properly synchronized and the associated shared stop state is automatically destroyed when the last user (stop source or stop token) gets destroyed.

The following example shows how to establish both a stop source and a stop token:

```
#include <stop_token>
...

// create stop_source and stop_token:
std::stop_source ssrc;                      // creates a shared stop state
std::stop_token stok{ssrc.get_token()};     // creates a token for the stop state
```

You simply first create the `stop_source` object which, provides the API to request a stop. The constructor also creates the associated shared stop state. Then, you can ask the stop source for the `stop_token` object, which provides the API to react to a stop requests (by polling or registering callbacks).

You can then pass the token (and/or the source) to locations/threads to establish the asynchronous communication between the places that might request a stop and those that might react to a stop.

There is no other way to create a stop token with an associated shared stop state. The default constructor has no associated stop state.

16.2.1 Stop Sources and Stop Tokens in Detail

Let us look at the API's of stop sources, stop tokens, and stop callbacks in detail.

Stop Sources in Detail

Table *Operations of Objects of Class* `stop_source` lists the API of `std::stop_source`.

To make it possible to create stop sources with no associated stop state (which might be useful because the stop state needs resources), you can create a stop source with a special constructor and assign a stop source later:

```
std::stop_source ssrc{nostopstate};   // no associated shared stop state
...
ssrc = std::stop_source{};            // assign new shared stop state
```

Operation	Effect
stop_source *s*	Default constructor; creates a stop source with no associated stop state
stop_source *s*{nostopstate}	Creates a stop source with no associated stop state
stop_source *s*{*s2*}	Copy constructor; creates a stop source that shares the associated stop state of *s2*
stop_source *s*{move(*s2*)}	Move constructor; creates a stop source that gets the associated stop state of *s2* (*s2* no longer has an associated stop state)
s.~***stop_source***()	Destructor; destroys the associated shared stop state if this is the last user of it
s = *s2*	Copy assignment; copy assigns the state of *s2* so that *s* now also shares the stop state of *s2* (any former stop state of *s* is released)
s = move(*s2*)	Move assignment; move assigns the state of *s2* so that *s* now shares the stop state of *s2* (*s2* no longer has a stop state and any former stop state of *s* is released)
s.**get_token()**	Yields a stop_token for the associated stop state (returns a stop token with no associated stop state if there is no stop state to share)
s.**request_stop()**	Requests a stop on the associated stop state if any if it is not done yet (returns whether a stop was requested)
s.stop_possible()	Yields whether *s* has an associated stop state
s.stop_requested()	Yields whether *s* has an associated stop state, for which a stop was requested
s1 == *s2*	Yields whether *s1* and *s2* share the same stop state (or both share none)
s1 != *s2*	Yields whether *s1* and *s2* do not share the same stop state
s1.swap(*s2*)	Swaps the states of *s1* and *s2*
swap(*s1*, *s2*)	Swaps the states of *s1* and *s2*

Table 16.1. Operations of Objects of Class stop_source

Stop Tokens in Detail

Table *Operations of Objects of Class* stop_token lists the API of std::stop_token.

Note that stop_possible() yields not only false if there is no associated stop state. If there is a stop state, but there is no stop source anymore *and* a stop was never requested, it also yields false. This can be used to avoid unnecessary dealing with possible stops if they are impossible to occur. For example, a request to register a callback is ignored.

16.2.2 Using Stop Callbacks

A stop callback is an object of the RAII type std::stop_callback. The constructor registers a callable (function, function object, or lambda) to be called when a stop is requested for a specified stop token:

```
void task(std::stop_token st)
```

Operation	Effect
stop_token *t*	Default constructor; creates a stop token with no associated stop state
stop_token *t*{*t2*}	Copy constructor; creates a stop token that shares the associated stop state of *t2*
stop_token *t*{move(*t2*)}	Move constructor; creates a stop token that gets the associated stop state of *t2* (*t2* no longer has an associated stop state)
t.~***stop_token***()	Destructor; destroys the associated shared stop state if this is the last user of it
t = *t2*	Copy assignment; copy assigns the state of *t2* so that *t* now also shares the stop state of *t2* (any former stop state of *t* is released)
t = move(*t2*)	Move assignment; move assigns the state of *t2* so that *t* now shares the stop state of *t2* (*t2* no longer has a stop state and any former stop state of *t* is released)
t.stop_possible()	Yields whether *t* has an associated stop state and a stop was or can (still) be requested
t.**stop_requested()**	Yields whether *t* has an associated stop state, for which a stop was requested
t1 == *t2*	Yields whether *t1* and *t2* share the same stop state (or both share none)
t1 != *t2*	Yields whether *t1* and *t2* do not share the same stop state
t1.swap(*t2*)	Swaps the states of *t1* and *t2*
swap(*t1*, *t2*)	Swaps the states of *t1* and *t2*
stop_token *cb*{*t*, *f*}	Registers *cb* as stop callback of *t* calling *f*

Table 16.2. Operations of Objects of Class stop_token

```
{
  // register temporary callback:
  std::stop_callback cb{st, []{
    std::cout << "stop requested\n";
    ...
  }};
  ...
} // unregisters callback
```

Assume, we have created the shared stop state and create an asynchronous situation where one thread might request for a stop and another thread might run task(). We might create this situation with code like the following:

```
// create stop_source with associated stop state:
std::stop_source ssrc;

// register/start task() and pass the corresponding stop token to it:
registerOrStartInBackgound(task, ssrc.get_token());
```

...

registerOrStartInBackgound() could be any approach to start task() asynchronously such as immediately or later calling std::async(), initializing a std::thread, calling a coroutine, registering an event handler.

Now whenever we request a stop:

```
ssrc.request_stop();
```

one of the following things can happen:

- If task() was already started (and its callback was initialized) and is still running (and the destructor of the callback was not called yet), the registered callable is called immediately in the thread where request_stop() was called. request_stop() blocks until all registered callables are called. The order of the calls is not defined.
- If task() was not started yet (or its callback was not initialized yet), request_stop() changes the stop state to signal that a stop was requested and returns. If later func() is started and the callback gets initialized, the callable is called immediately in the thread where the callback gets initialized The constructor of the callback blocks until the callable returns.
- If task() has already finished (or at least the destructor of the callback was called), the callable will never be called. The end of the lifetime of the callback signals that there is no longer the need to call the callable.

These scenarios are carefully synchronized. So even if we are in the middle of initializing a stop_callback so that the callable gets registered, one of the above scenarios will happen. The same applies if a stop is requested during unregistering a callable due to destroying a stop callback. If the callable was started already by another thread, the destructor blocks until the callable has finished.

For your programming logic this means that from the moment you initialize the callback until the end of its destruction the registered callable might be called. Up to the end of the constructor the callback runs in the thread of the initialization, afterwards in the thread requesting the stop. For the code that requests a stop it *might* immediately call the registered callable, might call it later (if the callback is initialized later), or might never call it (if it is too late to call the callback).

For example, consider the following program:

lib/stop.cpp

```
#include <iostream>
#include <stop_token>
#include <future>     // for async()
#include <thread>     // for sleep_for()
using namespace std::literals;  // for duration literals

void func(std::stop_token st, int num)
{
  auto id = std::this_thread::get_id();
  std::cout << "call func(" << num << ")\n";

  // register a first callback:
```

```
  std::stop_callback cb1{st, [num, id]{
    std::cout << "- STOP1 requested in func(" << num
              << (id == std::this_thread::get_id() ?  ")\n"
                                                   :  ") in main thread\n");
  }};
  std::this_thread::sleep_for(9ms);

  // register a second callback:
  std::stop_callback cb2{st, [num, id]{
    std::cout << "- STOP2 requested in func(" << num
              << (id == std::this_thread::get_id() ?  ")\n"
                                                   :  ") in main thread\n");
  }};
  std::this_thread::sleep_for(2ms);
}

int main()
{
  // create stop_source and stop_token:
  std::stop_source ssrc;
  std::stop_token stok{ssrc.get_token()};

  // register callback:
  std::stop_callback cb{stok, []{
    std::cout << "- STOP requested in main()\n" << std::flush;
  }};

  // in the background call func() a couple of times:
  auto fut = std::async([stok] {
                          for (int num = 1; num < 10; ++num) {
                            func(stok, num);
                          }
                        });

  // after a while, request stop:
  std::this_thread::sleep_for(120ms);
  ssrc.request_stop();
}
```

The output might for example be as follows:

```
call func(1)
call func(2)
...
call func(7)
call func(8)
```

```
  - STOP2 requested in func(8) in main thread
  - STOP1 requested in func(8) in main thread
  - STOP requested in main()
  call func(9)
  - STOP1 requested in func(9)
  - STOP2 requested in func(9)
```

or:

```
  call func(1)
  call func(2)
  call func(3)
  call func(4)
  - STOP2 requested in func(4) in main thread
  call func(5)
  - STOP requested in main()
  - STOP1 requested in func(5)
  - STOP2 requested in func(5)
  call func(6)
  - STOP1 requested in func(6)
  - STOP2 requested in func(6)
  call func(7)
  - STOP1 requested in func(7)
  - STOP2 requested in func(7)
  ...
```

or:

```
  call func(1)
  call func(2)
  call func(3)
  call func(4)
  - STOP requested in main()
  call func(5)
  - STOP1 requested in func(5)
  - STOP2 requested in func(5)
  call func(6)
  - STOP1 requested in func(6)
  - STOP2 requested in func(6)
  ...
```

or just:

```
  call func(1)
  call func(2)
  ...
  call func(8)
  call func(9)
  - STOP requested in main()
```

The output pretty often has even interleaved characters as the output from the main thread and the thread running `func()` might be completely mixed. (to avoid that we could have to use synchronized streams).

Stop Callbacks in Detail

The type of stop callback, `stop_callback` is a class template with a very limited API. Effectively, it only provides a constructor to register a callable for a stop token and a destructor that unregisters the callable. Copying and moving is deleted and no other member function is provided.

The template parameter is the type of the callable and is usually deduced when the constructor is initialized:

```
auto func = [] { ... };

std::stop_callback cb{myToken, func};   // deduces stop_callback<decltype(func)>
```

The type has a type member `callback_type` to deal with the type of the callable.

The constructors accept both lvalues (objects with a name) and rvalues (temporary objects or objects marked with `std::move()`):

```
auto func = [] { ... };

std::stop_callback cb1{myToken, func};              // copies func
std::stop_callback cb2{myToken, std::move(func)};   // moves func
std::stop_callback cb3{myToken, [] { ... }};        // moves the lambda
```

16.2.3 Constraints and Guarantees of Stop Tokens

The implemented mechanism is pretty robust regarding a couple of scenarios as they might occur in asynchronous contexts, but you cannot do everything.

The C++20 standard library guarantees the following:

- All `request_stop()`, `stop_requested()`, and `stop_possible()` calls are synchronized.
- The callback registration is guaranteed to be performed atomically. If there is a concurrent call to `request_stop()` from another thread, then either the current thread will see the request to stop and immediately invoke the callback on the current thread or the other thread will see the callback registration and will invoke the callback before returning from `request_stop()`.
- The callable of a `stop_callback` is guaranteed not to be called after the destructor of the `stop_callback` returns.
- The destructor of a callback waits for its callable to finish if it is just called by another thread (it does not wait for other callables to finish).

However, note the following constraints:

- A callback should not throw. If an invocation of its callable exits via an exception then `std::terminate()` is called.
- Do not destroy a callback in its own callable. The destructor does not wait for the callback to finish.

16.3 std::jthread In Detail

Table *Operations of Objects of Class* jthread lists the API of std::thread. The column **Diff** marks member function that **Mod**ified the behavior or are **New** compared to std::thread.

Operation	Effect	Diff
jthread *t*	Default constructor; creates a *nonjoinable* thread object	
jthread *t*{*f*,...}	Creates an object, representing a new thread that calls *f* (with additional args), or throws std::system_error	
jthread *t*{*rv*}	Move constructor; creates a new thread object, which gets the state of *rv*, and makes *rv nonjoinable*	
t.~***jthread***()	Destructor; calls request_stop() and join() if the object is *joinable*	Mod
t = *rv*	Move assignment; move assigns the state of *rv* to *t* (calls request_stop() and join() if *t* is *joinable*)	Mod
t.joinable()	Yields true if *t* has an associated thread (is *joinable*)	
t.join()	Waits for the associated thread to finish and makes the object *nonjoinable* (throws std::system_error if the thread is not *joinable*)	
t.detach()	Releases the association of *t* to its thread while the thread continues and makes the object *nonjoinable* (throws std::system_error if the thread is not *joinable*)	
t.**request_stop()**	Requests stop on the associated stop token	New
t.**get_stop_source()**	Yields an object to request a stop from	New
t.**get_stop_token()**	Yields an object to check for a requested stop	New
t.get_id()	Returns a unique thread ID of the member type id if *joinable* or a default constructed ID if not	
t.native_handle()	Returns a platform-specific member type native_handle_type for a non-portable handling of the thread	
t1.swap(*t2*)	Swaps the states of *t1* and *t2*	
swap(*t1*, *t2*)	Swaps the states of *t1* and *t2*	
hardware_concurrency()	Static function with hint about possible hardware threads	

Table 16.3. Operations of Objects of Class jthread

Note that both member types id and native_handle_type are the same for std::thread and std::jthread so hat it does not matter if you use decltype(mythread)::id or std::thread::id. That way, you can just replace std::thread in existing code by std::jthread without the need to replace anything else.

16.3.1 Using Stop Tokens with std::jthread

Besides the fact that the destructor joins, the major benefit of std::jthread is that it automatically establishes the mechanism to signal a stop. For that constructor starting a thread creates a stop source placed as

member of the tread object and passes the corresponding stop token to the called function ***if*** that function takes an additional `stop_token` as first parameter.

You can also get the stop source and stop token via the member function of the thread:

```
std::jthread t1{[] (std::stop_token st) {
                  ...
                }};
...
foo(t1.get_stop_token());   // pass stop token to foo()
...
std::stop_source ssrc{t1.get_stop_source()};
ssrc.request_stop();        // request stop on stop token of t1
```

Both, the stop source and the stop token are even usable if the thread was already detached and runs in the background.

It might also be necessary to request stop for multiple threads using the same stop token. That is easily possible. Just create the stop token yourself or take the stop token from a first started thread and pass it as first argument to the callable of the started threads:

```
// initialize a common stop token for all threads:
std::stop_source allStopSource;
std::stop_token allStopToken{allStopSource.get_token()};

for (int i = 0; i < 9; ++i) {
  threads.push_back(std::jthread{[] (std::stop_token st) {
                                   ...
                                   while (!st.stop_requested()) {
                                     ...
                                   }
                                 },
                                 allStopToken   // pass token to this thread
                                });
}
```

Remember that the that takes just all arguments passed. Only if there is a stop token parameter without passing a corresponding argument, the internal stop token of the started thread is used.

See lib/atomicref.cpp for a complete example.

Using Stop Tokens with Multiple `std::jthread`'s

If you start multiple `jthread`'s, each thread has its own stop token. Note that this might resolve in a situation that stopping all threads might take longer than expected. Consider the following code:

```
{
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    pool.push_back(std::jthread{[&] (std::stop_token st) {
                                  while (!st.stop_requested()) {
```

```
                                    ...
                                  }
                                }});
  }
  ...
} // destructor stops all threads
```

Note that at the end of the loop, the destructor stop all running threads similar to the following code:

```
for (auto& t : threads) {
  t.request_stop();
  t.join();
}
```

This means that we always wait for one thread to end before we signal stop to the next thread.

Code like this can be improved by requesting a stop for all threads before calling `join()` for all of them (via the destructor):

```
{
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    pool.push_back(std::jthread{[&] (std::stop_token st) {
                                  while (!st.stop_requested()) {
                                    ...
                                  }
                                }});
  }
  ...
  // BETTER: request stops before we start to join via destructor:
  for (auto& t : threads) {
    t.request_stop();
  }
} // destructor stops all threads
```

Now we first request a stop for all threads and then the destructor waits for their end.

16.4 Afternotes

The request that threads should join was first proposed by Herb Sutter in http://wg21.link/n3630 as a fix for `std::thread`. The finally accepted wording was formulated by Nicolai Josuttis, Lewis Baker, Billy O'Neal, Herb Sutter, and Anthony Williams in http://wg21.link/p0660r10.

This page is intentionally left blank

Chapter 17

Concurrency Features

After introducing `std::jthread` and stop tokens in the previous chapter, this chapter documents all other concurrency features C++20 introduced:

- Latches and barriers
- Counting and binary semaphores
- Various extensions for atomic types

17.1 Thread Synchronization with Latches and Barriers

Two new types provide new mechanisms to synchronize asynchronous computation/processing of multiple threads:

- **Latches** allow you to have a one-time synchronization where threads can wait for multiple tasks to finish.
- **Barriers** allow you to have a repeated synchronization of multiple threads when you have to react when they all have done their current/next processing.

17.1.1 Latches

A latch is a new synchronization mechanism for concurrent execution that support a single-use asynchronous countdown. Starting with an initial integral value various thread can atomically count this value down to zero. The moment the counter reaches zero, all threads waiting for this countdown continue.

Consider the following example:

lib/latch.cpp

```
#include <iostream>
#include <array>
#include <thread>
#include <latch>
using namespace std::literals;  // for duration literals
```

```
void loopOver(char c) {
  // loop over printing the char c:
  for (int j = 0; j < c/2; ++j) {
    std::cout.put(c).flush();
    std::this_thread::sleep_for(100ms);
  }
}

int main()
{
  std::array tags{'.', '?', '8', '+', '-'};  // tags we have to perform a task for

  // initialize latch to react when all tasks are done:
  std::latch allDone{tags.size()};    // initialize countdown with number of tasks

  // start two threads dealing with every second tag:
  std::jthread t1{[tags, &allDone] {
                    for (unsigned i = 0; i < tags.size(); i += 2) {  // even indexes
                      loopOver(tags[i]);
                      // signal that the task is done:
                      allDone.count_down();  // atomically decrement counter of latch
                    }
                    ...
                  }};
  std::jthread t2{[tags, &allDone] {
                    for (unsigned i = 1; i < tags.size(); i += 2) {  // odd indexes
                      loopOver(tags[i]);
                      // signal that the task is done:
                      allDone.count_down();  // atomically decrement counter of latch
                    }
                    ...
                  }};
  ...
  // wait until all tasks are done:
  std::cout << "\nwaiting until all tasks are done\n";
  allDone.wait();                         // wait until counter of latch is zero
  std::cout << "\nall tasks done\n";    // note: threads might still run
  ...
}
```

Here, we start two threads (using std::jthread) to perform a couple of tasks each processing a character of the array `tags`. Thus, the size of `tags` defines the number of tasks. The main thread blocks until all tasks are done. For this

- we initialize a latch with the number of tags/tasks:

```
std::latch allDone{tags.size()};
```

- let each task decrement the counter when it is done:

```
allDone.count_down();
```

- and let the main thread wait until all tasks are done (the counter is zero):

```
allDone.wait();
```

The output of the program might look as follows:

```
.?
waiting until all tasks are done
.??..??.?..?.??.?..??.?..?.??..?.??.?.?.?.?.8??88??88??8?8?8+8+88++8+8+8+88++
8+8+8+8+88++8+8+88++8+8-+--------------------
all tasks done
```

Note that the main thread may ***not*** assume that when the tasks are done all threads are done. The threads might still do other things and the system might still clean-up them. To wait until all threads are done, you have to call `join()` for both threads.

You can also use latches to synchronize them at a specific point and then continue. However, note that you can do this only once with each latch. One application of this is to ensure (as good as possible) that multiple threads start together with their actual work even when starting and initializing the thread might take some time.[1]

Consider the following example:

lib/latchready.cpp

```
#include <iostream>
#include <array>
#include <vector>
#include <thread>
#include <latch>
using namespace std::literals;  // for duration literals

int main()
{
  std::size_t numThreads = 10;

  // initialize latch to start the threads when all of them are initialized:
  std::latch allReady = 10;    // initialize countdown with number of threads

  // start numThreads threads:
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    threads.push_back(std::jthread{[i, &allReady]{
                                        // initialize each thread (simulate to take some time):
```

[1] Thanks to Anthony Williams for describing this scenario.

```
                                    std::this_thread::sleep_for(100ms * i);
                                    ...
                                    // synchronize threads so that all start together here:
                                    allReady.arrive_and_wait();

                                    // perform whatever the thread does
                                    // (loop printing its index):
                                    for (int j = 0; j < i + 5; ++j) {
                                      std::cout.put(static_cast<char>('0' + i)).flush();
                                      std::this_thread::sleep_for(50ms);
                                    }
                                  }});
  }
  ...
}
```

Here, we start `numThreads` threads (using std::jthread) which take time to start and being initialized and then use a latch to block until all started threads are started and initialized. For this

- we initialize a latch with the number of threads:

  ```
  std::latch allReady{numThreads};
  ```

- let each thread decrement the counter and wait when the initialization is done:

  ```
  allReady.arrive_and_wait();  // count_down() and wait()
  ```

The output of the program might look as follows:

```
867534210986753420199014253768867352419078635249107683524919425386794538769458
76957869786789899
```

You can see that all 10 threads (each printing its index) start more or less together.

Without the latch, the output might look as follows:

```
001010210213213241324352435243654635476357468547685479685796857965879687698769
98798798798989899
```

So, the early started threads run already while the latter were not started yet.

Latches in Detail

Class `std::latch` is declared in header file `<latch>`. Table *Operations of Objects of Class* `latch` lists the API of `std::latch`.

Note that you cannot copy or move (assign) a latch.

Also note that passing the size of a container (except `std::array`) as initial value of the counter is an error. The constructor takes a `std::ptrdiff_t`, which is signed, so that you get the following behavior:

```
std::latch l1{10};                  // OK
std::latch l2{10u};                 // warnings may occur
```

Operation	Effect
latch `l{`*`counter`*`}`	Creates a latch with *`counter`* as starting value for the countdown
`l.`**`count_down()`**	Atomically decrements the counter (if not `0` yet)
`l.count_down(`*`val`*`)`	Atomically decrements the counter by *`val`*
`l.`**`wait()`**	Blocks until the counter of the latch is `0`
`l.try_wait()`	Yields whether the counter of the latch is `0`
`l.`**`arrive_and_wait()`**	Calls `count_down()` and `wait()`
`l.arrive_and_wait(`*`val`*`)`	Calls `count_down(`*`val`*`)` and `wait()`
`max()`	Static function that yields the maximum possible value for *`counter`*

Table 17.1. Operations of Objects of Class `latch`

```
std::vector<int> coll{...};
...
std::latch l3{coll.size()};              // ERROR
std::latch l4 = coll.size();             // ERROR
std::latch l5(coll.size());              // OK (no narrowing checked)
std::latch l6{int(coll.size())};         // OK
std::latch l7{ssize(coll)};              // OK (see std::size())
```

17.1.2 Barriers

A barrier is a new synchronization mechanism for concurrent execution that allows you to synchronize multiple asynchronous tasks multiple times. Setting an initial counter multiple threads can count it down and wait until the counter reaches `0`. However, in contrast to latches then an optional call back is called and the counter resets to the initial counter again.

A barrier is useful when multiple threads repeatedly run compute/perform something together. Whenever all threads have done their task, an optional callback can process the result or new state and after that the asynchronous computation/processing can continue.

As an example, consider that we repeatedly use multiple threads to compute the square root of multiple values:

lib/barrier.cpp

```
#include <iostream>
#include <format>
#include <vector>
#include <thread>
#include <cmath>
#include <barrier>

int main()
{
  // initialize and print a collection of floating-point values:
```

```
  std::vector values{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

  // define a lambda function that prints all values
  // - NOTE: has to be noexcept to be used as barrier callback
  auto printValues = [&values] () noexcept{
                       for (auto val : values) {
                         std::cout << std::format(" {:<7.5}", val);
                       }
                       std::cout << '\n';
                     };
  // print initial values:
  printValues();

  // initialize a barrier that prints the values when all threads have done their computations:
  std::barrier allDone{int(values.size()),  // initial value of the counter
                       printValues};        // callback to call whenever the counter is 0

  // initialize a thread for each value to compute its square root in a loop:
  std::vector<std::jthread> threads;
  for (std::size_t idx = 0; idx < values.size(); ++idx) {
    threads.push_back(std::jthread{[idx, &values, &allDone] {
                                     // repeatedly:
                                     for (int i = 0; i < 5; ++i) {
                                       // compute square root:
                                       values[idx] = std::sqrt(values[idx]);
                                       // and synchronize with other threads to print values:
                                       allDone.arrive_and_wait();
                                     }
                                   }});
  }
  ...
}
```

After declaring an array of double value, we define a function to print them out:

```
  // initialize and print a collection of floating-point values:
  std::vector values{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

  // define a lambda function that prints all values
  // - NOTE: has to be noexcept to be used as barrier callback
  auto printValues = [&values] () noexcept{
                       for (auto val : values) {
                         std::cout << std::format(" {:<7.5}", val);
                       }
                       std::cout << '\n';
```

```
                    };
```

Note that the function has to be declared with `noexcept`. Internally, the function uses the new function `std::format()` for formatted output.

Now the goal is to use multiple threads so that each thread deals with one value. In this case we repeatedly compute the square roots of the values. For this

- we initialize a barrier to print all values whenever all threads have done their next computation: the number of tags/tasks:

  ```
  std::barrier allDone{int(values.size()),   // initial value of the counter
                       printValues};         // callback to call whenever the counter is 0
  ```

  Please note that that constructor should take a signed integral value. Otherwise the code might not compile.
- let each thread decrement the counter and wait so that the values are printed when all counters are done before the threads continue:

  ```
  allDone.arrive_and_wait();
  ```

The output of the program might look as follows:

```
1       2       3       4       5       6       7       8
1       1.4142  1.7321  2       2.2361  2.4495  2.6458  2.8284
1       1.1892  1.3161  1.4142  1.4953  1.5651  1.6266  1.6818
1       1.0905  1.1472  1.1892  1.2228  1.251   1.2754  1.2968
1       1.0443  1.0711  1.0905  1.1058  1.1185  1.1293  1.1388
1       1.0219  1.0349  1.0443  1.0516  1.0576  1.0627  1.0671
```

The API of barriers also provide a function to remove threads from this mechanism. This is something you need, for example, to avoid deadlocks, when the loop runs until a stop was signaled.

The code for the started threads might then look as follows:

```
// initialize a thread for each value to compute its square root in a loop:
std::vector<std::jthread> threads;
for (std::size_t idx = 0; idx < values.size(); ++idx) {
  threads.push_back(std::jthread{[idx, &values, &allDone] (std::stop_token st) {
                                   // repeatedly:
                                   while (!st.stop_requested()) {
                                     // compute square root:
                                     values[idx] = std::sqrt(values[idx]);
                                     // and synchronize with other threads to print values:
                                     allDone.arrive_and_wait();
                                   }
                                   // remove thread from counter so that other threads don't wait:
                                   allDone.arrive_and_drop();
                                 }});
}
```

The lambda takes now stop token to react if a stop was requested (explicitly or by calling the destructor for the thread). Now if the main threads signals the threads to stop the computing, for example as follows:

```
threads.clear();
```

it is important to call

```
allDone.arrive_and_drop();
```

This ensures that the counter for this threads is counted down ***and*** that the next starting value is decremented accordingly. Because in the next round (the other threads might still run) the barrier should no longer wait for this thread.

See *lib/barrierstop.cpp* for a complete example.

Barriers in Detail

Class `std::barrier` is declared in header file `<barrier>`. Table *Operations of Objects of Class* `barrier` lists the API of `std::barrier`.

Operation	Effect
barrier `b{`*num*`}`	Creates a barrier for *num* asynchronous tasks
barrier `b{`*num*`, cb}`	Creates a barrier for *num* asynchronous tasks and `cb` as callback
`b.arrive()`	Marks one task as done and yields an arrival token
`b.arrive(`*val*`)`	Marks *val* tasks as done and yields an arrival token
`b.wait(`*arrivalToken*`)`	Blocks until all tasks are done and the callback was called (if any)
`b.`**`arrive_and_wait()`**	Marks one task as done and blocks until all tasks are done and the callback was called (if any)
`b.`**`arrive_and_drop()`**	Marks one task as done and decrements the number of tasks to repeatedly perform
`max()`	Static function that yields the maximum possible value for *num*

Table 17.2. Operations of Objects of Class `barrier`

`std::barrier<>` is a class template with the type of the callback as template parameter. Usually the type is deduced by *class template argument deduction*:

```
void callback() noexcept;   // forward declaration
...

std::barrier b{6, callback};   // deduces std::barrier<decltype(callback)>
```

Note that the C++ standard requires that callbacks for barriers guarantees not to throw. Therefore, to be portable, you have to declare the function or lambda with `noexcept`. If no callback is passed, an implementation-specific type is used representing an operation that has no effects.

The call

```
l.arrive_and_wait();
```

is equivalent to

```
l.wait(l.arrive());
```

That means the `arrive()` function returns an *arrival token* of type `std::barrier::arrival_token`, which ensures that barrier knows to which of the treads waits. Otherwise it cannot handle `arrive_and_drop()` correctly.

Note that you cannot copy or move (assign) a barrier.

Also note that passing the size of a container (except `std::array`) as initial value of the counter is an error. The constructor takes a `std::ptrdiff_t`, which is signed, so that you get the following behavior:

```
std::barrier b1{10, cb};                    // OK
std::barrier b2{10u, cb};                   // warnings may occur

std::vector<int> coll{...};
...
std::barrier b3{coll.size(), cb};           // ERROR
std::barrier b4(coll.size(), cb);           // OK (no narrowing checked)
std::barrier b5{int(coll.size()), cb};      // OK
std::barrier b6{ssize(coll), cb};           // OK (see std::size())
```

17.2 Semaphores

C++20 provides new types to deal with semaphores. Semaphores are lightweight synchronization primitive, which allow you to synchronize or restrict access to one or a group of resources.

You can use them like mutexes with the benefit that the threads granting/releasing access to a resource do not necessarily have to be the same threads that requested/acquired a resource. You can also use them to limit the availability of resource such as enabling and disabling the use of threads in a thread pool.

There are two semaphore types provided by the C++ standard library:

- `std::counting_semaphore<>` to limit the use of multiple resources up to a maximum value.
- `std::binary_semaphore<>` to limit the use of a single resource.

17.2.1 Example of Using Counting Semaphores

The following program demonstrates how semaphores operate in general:

lib/semaphore.cpp

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <mutex>
#include <semaphore>
using namespace std::literals;   // for duration literals

int main()
{
  std::queue<char> values;   // queue of values
```

```
std::mutex valuesMx;        // mutex to protect access to the queue

// initialize a queue with multiple sequences from 'a' to 'z':
// - no mutex because no other thread is running yet
for (int i = 0; i < 1000; ++i) {
  values.push(static_cast<char>('a' + (i % ('z'-'a'))));
}

// create a pool of numThreads threads:
// - limit their availability with a semaphore (initially none available):
constexpr int numThreads = 10;
std::counting_semaphore<numThreads> enabled{0};

// create and start all threads of the pool:
std::vector<std::jthread> pool;
for (int idx = 0; idx < numThreads; ++idx) {
  pool.push_back(std::jthread{[&enabled, &values, &valuesMx, idx] (std::stop_token st) {
                              while (!st.stop_requested()) {
                                // request thread to become one of the enabled threads:
                                enabled.acquire();

                                // get next value from the queue:
                                char val;
                                {
                                  std::lock_guard lg{valuesMx};
                                  val = values.front();
                                  values.pop();
                                }

                                // print the value 10 times:
                                for (int i = 0; i < 10; ++i) {
                                  std::cout.put(val).flush();
                                  auto dur = 130ms * ((idx % 3) + 1);
                                  std::this_thread::sleep_for(dur);
                                }

                                // remove thread from the set of enabled threads:
                                enabled.release();
                              }
                            }});
}

std::cout << "== wait 2 seconds (no thread enabled)\n" << std::flush;
std::this_thread::sleep_for(2s);
```

```
  // enable 3 concurrent threads:
  std::cout << "== enable 3 parallel threads\n" << std::flush;
  enabled.release(3);
  std::this_thread::sleep_for(2s);

  // enable 2 more concurrent threads:
  std::cout << "\n== enable 2 more parallel threads\n" << std::flush;
  enabled.release(2);
  std::this_thread::sleep_for(2s);

  // Normally we would run forever, but let's end the program here.
  std::cout << "\n== stop processing\n" << std::flush;
  for (auto& t : pool) {
    t.request_stop();
  }
}
```

In this program, we start 10 threads, but limit how many of them are allowed to actually process data. For this we initialize a semaphore with the maximum possible number we might allow (10) and the initial number of resources allowed (zero):

```
constexpr int numThreads = 10;
std::counting_semaphore<numThreads> enabled{0};
```

You might wonder why we have to specify a maximum as template parameter. The reason is that with this compile-time value the library can decide to switch to the most efficient implementation possible (native support might be possible only up to a certain value or if the maximum is 1 we may be able to use simplifications).

In each thread we use the semaphore to ask for permission to run. We try to "acquire" one of the available resources to start the task and "release" the resource for other use when we are done:

```
std::jthread{[&, idx] (std::stop_token st) {
               while (!st.stop_requested()) {
                 // request thread to become one of the enabled threads:
                 enabled.acquire();
                 ...

                 // remove thread from the set of enabled threads:
                 enabled.release();
               }
             }}
```

Initially we are blocked, because the semaphore was initialized with zero so that no resources are available. However, later we use the semaphore to allow three threads to act concurrently:

```
// enable 3 concurrent threads:
enabled.release(3);
```

And even later we allow two more threads to run concurrently:

```
// enable 2 more concurrent threads:
enabled.release(2);
```

If you want to sleep or do something else if you cannot get a resource, you can use `try_acquire()`:

```
std::jthread{[&, idx] (std::stop_token st) {
               while (!st.stop_requested()) {
                 // request thread to become one of the enabled threads:
                 if (enabled.try_acquire()) {
                   ...

                   // remove thread from the set of enabled threads:
                   enabled.release();
                 }
                 else {
                   ...
                 }
               }
             }}
```

You can also try to acquire a resource for a limited time using `try_acquire_for()` or `try_acquire_until()`:

```
std::jthread{[&, idx] (std::stop_token st) {
               while (!st.stop_requested()) {
                 // request thread to become one of the enabled threads:
                 if (enabled.try_acquire_for(100ms)) {
                   ...

                   // remove thread from the set of enabled threads:
                   enabled.release();
                 }
               }
             }}
```

That way we would double check the status of the stop token from time to time.

Thread Scheduling is Not Fair

Note that thread schedulers do handle threads not fair. There is not guarantee that that are preferred that wait the longest time. Usually the contrary is true: if a thread scheduler already has a thread running calling `release()` and it immediately calls `acquire()` the scheduler keeps the thread running ("great, I need no context switch"). Therefore, we have no guarantee which one of multiple threads that wait with `acquire()` are woken-up. It might happen that always the same thread(s) is/are used.

As a consequence, you should ***not*** take the next value out of the queue before you ask for permission to run.

```
// BAD if order of processing matters:
{
```

```
    std::lock_guard lg{valuesMx};
    val = values.front();
    values.pop();
  }
  enabled.acquire();
  ...
```

It might happen that the value `val` is processed after several other values read later of even never processed. Ask for permission before you read the next value:

```
  enabled.acquire();
  {
    std::lock_guard lg{valuesMx};
    val = values.front();
    values.pop();
  }
  ...
```

For the same reason, you cannot easily reduce the number of enabled threads. Yes, you can try to call:

```
  // reduce the umber of enabled concurrent threads by one:
  enabled.acquire();
```

However, we do not know when this statement gets processed. As threads are not treated fair, the reaction to reduce the number of enabled threads may take very long or even take forever.

For a fair way to deal with queues and immediate reaction on resource limitations, you might want to use the new `wait()` and notify mechanism of atomics.

17.2.2 Example of Using Binary Semaphores

For semaphores there is a special type `std::binary_semaphore` defined, which is just a shortcut for `std::counting_semaphore<1>` so that it can only enable or disable the use of a single resource.

You could also use it as a mutex with the benefit that the thread releasing the resource does not have to be the same thread that formerly acquired the resource. However, a more typical application is a mechanism to signal/notify a thread from another. In contrast to condition variables you can to that multiple times.

Consider the following example:

lib/semaphorenotify.cpp

```
#include <iostream>
#include <chrono>
#include <thread>
#include <semaphore>
using namespace std::literals;  // for duration literals

int main()
{
  int sharedData;
  std::binary_semaphore dataReady{0};  // signal there is data to process
```

```
  std::binary_semaphore dataDone{0};    // signal processing is done

  // start threads to read and process values by value:
  std::jthread process{[&] (std::stop_token st) {
                         while(!st.stop_requested()) {
                           // wait until next value ready:
                           // - timeout after 1s to check stop_token
                           if (dataReady.try_acquire_for(1s)) {
                             int data = sharedData;

                             // process it:
                             std::cout << "[process] read " << data << std::endl;
                             std::this_thread::sleep_for(data * 0.5s);
                             std::cout << "[process]      done" << std::endl;

                             // signal processing done:
                             dataDone.release();
                           }
                           else {
                             std::cout << "[process] timeout" << std::endl;
                           }
                         }
                       }};

  // generate a couple of values:
  for (int i = 0; i < 10; ++i) {
    // store next value:
    std::cout << "[main] store " << i << std::endl;
    sharedData = i;

    // signal to start processing:
    dataReady.release();

    // wait until processing is done:
    dataDone.acquire();
    std::cout << "[main] processing done\n" << std::endl;
  }
  // end of loop signals stop
}
```

We use two binary semaphores to let one thread notify another thread:

- With `dataReady` the main thread notifies thread `process` that there is new data in `sharedData` to process.
- With `dataDone` the processing thread notifies the main thread that the data was processed.

Both semaphores are initialized by zero so that be default the acquiring thread blocks. The moment the notifying thread calls `release()`, the acquiring thread unblocks so that it can react.

The output of the program is something like the following:

```
[main] store 0
[process] read 0
[process]      done
[main] processing done

[main] store 1
[process] read 1
[process]      done
[main] processing done

[main] store 2
[process] read 2
[process]      done
[main] processing done

[main] store 3
[process] read 3
[process]      done
[main] processing done
...

[main] store 9
[process] read 9
[process]      done
[main] processing done

[process] timeout
```

Note that the processing thread only acquires for a limited time using `try_acquire_for()`. The return value yields whether it got notified (access to the resource). That way the thread can check from time to time whether a stop was signaled (as it is done after when main thread ends).

Semaphores in Detail

The class template `std::counting_semaphore<>` is declared in header file `<semaphore>` together with the shortcut `std::binary_semaphore` for `std::counting_semaphore<1>`:

```
namespace std {
  template<ptrdiff_t least_max_value = implementation-defined >
  class counting_semaphore;

  using binary_semaphore = counting_semaphore<1>;
}
```

Table *Operations of Objects of Class* `counting_semaphore<>` *and* `binary_semaphore` lists the API of semaphores.

Operation	Effect
semaphore `s{`*num*`}`	Creates a semaphore with the counter initialized with *num*
`s.`**`acquire()`**	Blocks until it could atomically decrement the counter (requesting for one more resource).
`s.try_acquire()`	Tries to immediately atomically decrement the counter (requesting for one more resource) and yields `true` whether this was successful
`s.try_acquire_for(`*dur*`)`	Tries for duration *dur* to atomically decrement the counter (requesting for one more resource) yields `true` whether this was successful
`s.try_acquire_until(`*tp*`)`	Tries until timepoint *tp* to atomically decrement the counter (requesting for one more resource) yields `true` whether this was successful
`s.`**`release()`**	Atomically increments the counter (enabling one more resource)
`s.release(`*num*`)`	Atomically adds *num* to the counter (enabling *num* more resource)
`max()`	Static function that yields the maximum possible value the counter

Table 17.3. Operations of Objects of Class `counting_semaphore<>` *and* `binary_semaphore`

Note that you cannot copy or move (assign) a semaphore.

Also note that passing the size of a container (except `std::array`) as initial value of the counter is an error. The constructor takes a `std::ptrdiff_t`, which is signed, so that you get the following behavior:

```
std::counting_semaphore s1{10};                  // OK
std::counting_semaphore s2{10u};                 // warnings may occur

std::vector<int> coll{...};
...
std::counting_semaphore s3{coll.size()};         // ERROR
std::counting_semaphore s4 = coll.size();        // ERROR
std::counting_semaphore s4(coll.size());         // OK (no narrowing checked)
std::counting_semaphore s6{int(coll.size())};    // OK
std::counting_semaphore s7{ssize(coll)};         // OK (see std::size())
```

17.3 Extensions for and New Atomic Types

C++20 introduces a couple of new atomics types (dealing with references and shared pointers) and new features for atomic types.

17.4 Atomic References with `std::atomic_ref<>`

Since C++11, the C++ standard library provides the class template `std::atomic<>` to provide types that wraps trivially copyable types with an atomic API.

C++20 introduces the class template `std::atomic_ref<>` to wrap trivially copyable ***reference*** types with an atomic API. That way you can provide a temporary atomic API to an existing object. One application would be to initialize the object without caring about concurrency and later use them with different threads. The reason to introduce a new type is that users should see that the object might provide non-atomic access and that there are weaker guarantees that for `std::atomic<>`.

In addition, we have a type `std::atomic<char8_t>` now.

An Example using Atomic References

The following program demonstrates how to use atomic references:

lib/atomicref.cpp

```
#include <iostream>
#include <array>
#include <algorithm>    // for std::fill_n()
#include <vector>
#include <format>
#include <random>
#include <thread>
#include <atomic>       // for std::atomic_ref<>
using namespace std::literals;  // for duration literals

int main()
{
  // create and initialize an array of integers with the value 100:
  std::array<int, 1000> values;
  std::fill_n(values.begin(), values.size(), 100);

  // initialize a common stop token for all threads:
  std::stop_source allStopSource;
  std::stop_token allStopToken{allStopSource.get_token()};

  // start multiple threads concurrently decrementing the value:
  std::vector<std::jthread> threads;
  for (int i = 0; i < 9; ++i) {
    threads.push_back(std::jthread{
      [&values] (std::stop_token st) {
        // initialize random engine to generate an index:
        std::mt19937 eng{std::random_device{}()};
        std::uniform_int_distribution distr{0, int(values.size()-1)};
```

```
        while (!st.stop_requested()) {
          // compute the next index:
          int idx = distr(eng);

          // enable atomic access to the value with the index:
          std::atomic_ref val{values[idx]};

          // and use it:
          --val;
          if (val <= 0) {
            std::cout << std::format("index {} is zero\n", idx);
          }
        }
      },
      allStopToken  // pass the common stop token
    });
  }

  // after a while/event request to stop all threads:
  std::this_thread::sleep_for(0.5s);
  std::cout << "\nSTOP\n";
  allStopSource.request_stop();
  ...
}
```

We first create and initialize an array of 1000 integral values without using atomics:

```
std::array<int, 1000> values;
std::fill_n(values.begin(), values.size(), 100);
```

However, later we start multiple threads that concurrently decrement these values. Now the point is that only in that context we use the values as atomic integers. For this, we initialization a `std::atomic_ref<>`:

```
std::atomic_ref val{values[idx]};
```

Note that due to class template argument deduction we do not have to specify the type of the object we refer to. Now when using the atomic reference the access to the value happens atomically:

- `--val` decrements the value atomically
- `val <= 0` loads the value atomically to compare it with `0`

You might consider to implement the latter as `val.load() <= 0` to document that an atomic interface is used.

Features of Atomic References

The header file is also `<atomic>` and as for `std::atomic<>` has a specialization for raw pointers and integral types:

```
namespace std {
```

```
    template<typename T> struct atomic_ref;        // primary template

    template<typename T> struct atomic_ref<T*>;    // partial specialization for pointers

    template<> struct atomic_ref<integralType>;    // full specializations for integral types
}
```

Atomic reference have the following restrictions compared to `std::atomic<>`:

- No `volatile` support.
- The objects to refer to might need an alignment that is greater than the usual alignment for the underlying type. The static member `std::atomic_ref<type>::required_alignment` provides this minimum alignment.

Atomic reference have the following extensions compared to `std::atomic<>`:

- A copy constructor to create another reference to the same underlying object (but an assignment operator is only provided to assign the underlying values).
- Thread synchronization support with `wait()`, `notify()_one()`, and `notify_all()`, as it is now provided for all atomic types.

Regarding all other aspects the same features as for `std::atomic<>` are provided:

- The type provides both a high-level API with memory barriers and a low-level API to disable them.
- The static member `is_always_lock_free` and the non-static member function `is_lock_free()` yield whether atomic support is lock-free. This might still depend on the used alignment.

Atomics References in Detail

******* This chapter/section is at work *******

17.4.1 Atomic Shared Pointers

C++11 introduced shared pointers with an optional atomic interface. With functions like `atomic_load()`, `atomic_store()`, and `atomic_exchange()` you could access the values to where shared pointers refer concurrently. However, the problem was you could still use these shared pointers with the non-atomic interface, which would then undermine all atomic use of the shared pointers.

C++20 now provides partial specializations for shared pointers and weak pointers:

- `std::atomic<std::shared_ptr<T>>`
- `std::atomic<std::weak_ptr<T>>`

The former atomic API for shared pointers is deprecated now.

Note that atomic shared/weak pointers do ***not*** provide the additional operations atomic raw pointers provide. They provide the same API as `std::atomic<>` provides in general for a trivially copyable type `T`.

Thread synchronization support with `wait()`, `notify()_one()`, and `notify_all()`, is provided. The low-level atomic interface with the option to use the parameters for memory order is also supported.

Example of Using Atomic Shared Pointers

The following example demonstrates the use of an atomic shared pointer that is use as head of a linked list of shared values.

lib/atomicshared.cpp

```
#include <iostream>
#include <thread>
#include <memory>                 // includes ¡atomic¿ now
using namespace std::literals;    // for duration literals


template<typename T>
class AtomicList {
 private:
  struct Node {
    T val;
    std::shared_ptr<Node> next;
  };
  std::atomic<std::shared_ptr<Node>> head;
public:
  AtomicList() = default;

  void insert(T v) {
    auto p = std::make_shared<Node>();
    p->val = v;
    p->next = head;
    while (!head.compare_exchange_weak(p->next, p)) {   // atomic update
    }
  }

  void print() const {
    std::cout << "HEAD";
    for (auto p = head.load(); p; p = p->next) {   // atomic read
      std::cout << "->" << p->val;
    }
    std::cout << std::endl;
  }
};


int main()
{
  AtomicList<std::string> alist;
```

```
  // populate list by elements from 10 threads:
  {
    std::vector<std::jthread> threads;
    for (int i = 0; i < 100; ++i) {
      threads.push_back(std::jthread{[&, i]{
                                        for (auto s : {"hi", "hey", "ho", "last"}) {
                                          alist.insert(std::to_string(i) + s);
                                          std::this_thread::sleep_for(5ns);
                                        }
                                      }});
    }
  } // wait for all treads to finish

  // print resulting list:
  alist.print();
}
```

The program may have the following output:

```
HEAD->94last->94ho->76last->68last->57last->57ho->60last->72last->... ->1hey->1hi
```

As usual with atomic operations, you could also write:

```
for (auto p = head.load(); p; p = p->next)
```

instead of:

```
for (auto p = head.load(); p.load(); p = p->next)
```

Example of Using Atomic Weak Pointers

The following example demonstrates the use of an atomic weak pointer:

lib/atomicweak.cpp

```
#include <iostream>
#include <thread>
#include <memory>                 // includes ¡atomic¿ now
using namespace std::literals;    // for duration literals


int main()
{
  std::atomic<std::weak_ptr<int>> pShared;   // pointer to current shared value (if any)

  // loop to set shared value for some time:
  std::atomic<bool> done{false};
  std::jthread updates{[&] {
                         for (int i = 0; i < 10; ++i) {
```

```
                        {
                          auto sp = std::make_shared<int>(i);
                          pShared.store(sp);    // atomic update
                          std::this_thread::sleep_for(0.1s);
                        }
                        std::this_thread::sleep_for(0.1s);
                      }
                      done.store(true);
                    }};

  // loop to print shared value (if any):
  while (!done.load()) {
    if (auto sp = pShared.load().lock()) {  // atomic read
      std::cout << "shared: " << *sp << '\n';
    }
    else {
      std::cout << "shared: <no data>\n";
    }
    std::this_thread::sleep_for(0.07s);
  }
}
```

Note that in this program we do not have to make the shared pointer atomic because it is used only by one thread. The only issue is that two threads concurrently update or use the weak pointer.

The program may have the following output:

```
shared: <no data>
shared: 0
shared: <no data>
shared: 1
shared: <no data>
shared: <no data>
shared: 2
shared: <no data>
shared: <no data>
shared: 3
shared: <no data>
shared: 4
shared: 4
shared: <no data>
shared: 5
shared: 5
shared: <no data>
shared: 6
shared: <no data>
```

```
shared: <no data>
shared: 7
shared: <no data>
shared: 8
shared: 8
shared: <no data>
shared: 9
shared: 9
shared: <no data>
```

As usual with atomic operations, you could also write:

```
pShared = sp;
```

instead of:

```
pShared.store(sp);
```

17.4.2 Atomic Floating-Point Types

Both `std::atomic<>` and `std::atomic_ref<>` now provide full specializations for types `float`, `double`, and `long double`.

In contrast to the primary template for arbitrary trivially copyable types they provide the additional atomic operations to add and subtract a value:[2]

- `fetch_add()`, `fetch_sub()`
- `operator+=`, `operator-=`

Thus, the following is possible now:

```
std::atomic<double> d{0};
...
d += 10;    // OK since C++20
```

17.4.3 Thread Synchronization with Atomic Types

All atomic types (`std::atomic<>`, `std::atomic_ref<>`, and `std::atomic_flag`) now provide a simple API to let threads block and wait for changes of their values caused by other threads.

Thus, for an atomic value:

```
std::atomic<int> aVal{100};
```

or an atomic reference:

```
int value = 100;
std::atomic_ref<int> aVal{value};
```

you can define that you want to wait until the referenced value has changed:

[2] In contrast to specializations to integral types, which also provide atomic support to increment/decrement values and perform bit-wise modifications.

```
int lastValue = aVal.load();
aVal.wait(lastValue);   // block unless/until value changed (and notified)
```

If the value of the referenced object does not fit the passed argument, it immediately returns. Otherwise, it returns if the value does not longer fit ***and*** `notify_one()` or `notify_all()` was called:

```
--aVal;                 // atomically modify the (referenced) value
aVal.notify_all();      // notify all threads waiting for a change
```

The underlying implementation even deal with spurious wake-ups (although in that case a `wait()` might return if the values changed without having a notification).

Note that it is not guaranteed to get all updates. Consider the following program:

lib/atomicwait.cpp

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std::literals;

int main()
{
  std::atomic<int> aVal{0};

  // reader:
  std::jthread tRead{[&] {
                       int lastX = aVal.load();
                       while (lastX >= 0) {
                         aVal.wait(lastX);
                         std::cout << "=> x changed to " << lastX << std::endl;
                         lastX = aVal.load();
                       }
                       std::cout << "READER DONE" << std::endl;
                     }};

  // writer:
  std::jthread tWrite{[&] {
                        for (int newVal : { 17, 34, 3, 42, -1}) {
                          std::this_thread::sleep_for(5ns);
                          aVal = newVal;
                          aVal.notify_all();
                        }
                      }};
  ...
}
```

The output might be:

```
=> x changed to 17
=> x changed to 34
=> x changed to 3
=> x changed to 42
=> x changed to -1
READER DONE
```

or:

```
=> x changed to 17
=> x changed to 3
=> x changed to -1
READER DONE
```

or just:

```
READER DONE
```

Note that the notification functions are `const` member functions.

Fair Ticketing with Atomic Notifications

One option to use atomic `wait()`'s and notifications is to use them like mutexes. This often pays off because using mutexes might be significant more expensive.

Here is a example where we use atomics to implement a fair processing of values in a queue (compare with the unfair version using semaphores). Although multiple threads might wait only a limited number of them might run. And by using a ticketing system we ensure that the elements in the queue are processed in order:[3]

lib/atomicticket.cpp

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <mutex>
#include <semaphore>
using namespace std::literals;  // for duration literals

int main()
{
  std::queue<char> values;  // queue of values
  std::mutex valuesMx;      // mutex to protect access to the queue

  // initialize a queue with some values
```

[3] The idea for this example is based on an example by Bryce Adelstein Lelbach in his talk *The C++20 Synchronization Library* at the CppCon 2029 (see http://youtu.be/Zcqwb3CWqs4?t=1810).

```
// - no mutex because no other thread is running yet:
for (int i = 0; i < 1000; ++i) {
  values.push(char(i % (128-32) + 32));
}

// limit the availability of threads with a ticket system:
std::atomic<int> maxTicket{0};   // maximum requested ticket no
std::atomic<int> actTicket{0};   // current allowed ticket no

// create and start a pool of numThreads threads:
constexpr int numThreads = 10;
std::vector<std::jthread> pool;
for (int idx = 0; idx < numThreads; ++idx) {
  pool.push_back(std::jthread{[&, idx] (std::stop_token st) {
                                while (!st.stop_requested()) {
                                  // get next value from the queue:
                                  char val;
                                  {
                                    std::lock_guard lg{valuesMx};
                                    val = values.front();
                                    values.pop();
                                  }

                                  // request a ticket and wait until it is called:
                                  int myTicket{++maxTicket};
                                  int act = actTicket.load();
                                  while (act < myTicket) {
                                    actTicket.wait(act);
                                    act = actTicket.load();
                                  }

                                  // print the value 10 times:
                                  for (int i = 0; i < 10; ++i) {
                                    std::cout.put(val).flush();
                                    auto dur = 20ms * ((idx % 3) + 1);
                                    std::this_thread::sleep_for(dur);
                                  }

                                  // next ticket please:
                                  ++actTicket;
                                  actTicket.notify_all();
                                }
                              }});
}
```

```
  // enable and disable threads in the thread pool:
  auto enable = [&, oldNum = 0] (int newNum) mutable {
                  actTicket += newNum - oldNum;             // enable/disable tickets
                  if (newNum > 0) actTicket.notify_all();   // wake up waiting threads
                  oldNum = newNum;
                };

  for (int num : {0, 3, 5, 2, 0, 1}) {
    std::cout << "\n\n====== enable " << num << " threads" << std::endl;
    enable(num);
    std::this_thread::sleep_for(2s);
  }

  std::quick_exit(0);
}
```

Any thread ready to perform its processing requests a ticket:

```
int myTicket{++maxTicket};
```

and waits until it is called:

```
int act = actTicket.load();
while (act < myTicket) {
  actTicket.wait(act);
  act = actTicket.load();
}
```

New tickets are enabled by increase the value of the `actTicket` and notifying all waiting threads. This happens either when a processing is done:

```
++actTicket;
actTicket.notify_all();
```

or when a new number of tickets is enabled:

```
actTicket += newNum - oldNum;             // enable/disable tickets
if (newNum > 0) actTicket.notify_all();   // wake up waiting threads
```

17.4.4 Extensions for `std::atomic_flag`

Before C++20, there was no way to check the value of a `std::atomic_flag` without setting it, so C++20 added global and member functions to check for the current value:

- `atomic_flag_test(const atomic_flag*) noexcept;`
- `atomic_flag_test_explicit(const atomic_flag*, memory_order) noexcept;`
- `atomic_flag::test()` const noexcept;
- `atomic_flag::test(memory_order)` const noexcept;

17.5 Synchronized Output Streams

C++20 provides a new mechanism to synchronize concurrent output to streams.

17.5.1 Motivation of Synchronized Output Streams

If multiple thread write concurrently to a stream, the output usually has to be synchronized:

- In general, concurrent output to a stream causes undefined behavior (it is a *data race*, the C++ term for a race condition with undefined behavior).
- Concurrent output to standard streams such as `std::cout` is supported, but the result is not be very useful, because character might by mixed in any arbitrary order.

For example, consider the following program:

lib/concstream.cpp

```
#include <iostream>
#include <cmath>
#include <thread>

void squareRoots(int num)
{
  for (int i = 0; i < num ; ++i) {
    std::cout << "squareroot of " << i << " is "
              << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
}
```

3 threads concurrently write to `std::cout`. This is valid because a standard output stream is used. However, the output may look like this:

```
squareroot of squareroot of 0 is 0 is 0
0squareroot of squareroot of
01squareroot of  is  is 101 is

1squareroot of squareroot of
12squareroot of  is  is 21 is 1.41421

1.41421squareroot of squareroot of
23squareroot of  is  is 31.41421 is 1.73205
```

```
1.73205squareroot of squareroot of
34squareroot of  is  is 41.73205 is 2

2squareroot of
4 is 2
```

17.5.2 Using of Synchronized Output Streams

By using synchronized output streams, we can now synchronize the concurrent output of multiple threads to the same stream. We only have to use a `std::osyncstream` initialized with the corresponding output stream. For example:

lib/syncstream.cpp

```
#include <iostream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(int num)
{
  for (int i = 0; i < num ; ++i) {
    std::osyncstream coutSync{std::cout};
    coutSync << "squareroot of " << i << " is "
             << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
}
```

Here, the synchronized output buffer synchronizes the output with other output to a synchronized output buffer so that it is only flushed, when the destructor of the synchronize output buffer is called.

As a result, the output looks like this:

```
squareroot of 0 is 0
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 2 is 1.41421
```

```
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 4 is 2
```

The three threads now write line-by-line to `std::cout`. However, still the exact order of the lines written is open.

You can get the same result by implementing the loop as follows:

```
for (int i = 0; i < num ; ++i) {
  std::osyncstream{std::cout} << "squareroot of " << i << " is "
                              << std::sqrt(i) << '\n';
}
```

Note that neither `'\n'` nor `std::endl` now `std::flush` writes the output. You really need the destructor. If we create the synchronized output stream outside the loop, the whole output of any thread is printed together when the destructor is reached.

However, there is a new manipulator to write the output before the destructor gets called: `std::flush_emit`. So, you can create and initialize the synchronized output stream and emit your output line-by-line also as follows:

```
std::osyncstream coutSync{std::cout};
for (int i = 0; i < num ; ++i) {
  coutSync << "squareroot of " << i << " is "
           << std::sqrt(i) << '\n' << std::flush_emit;
}
```

17.5.3 Using Synchronized Output Streams for Files

You can also use synchronized output stream for files. Consider the following example:

lib/syncfilestream.cpp

```
#include <fstream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(std::ostream& strm, int num)
{
  std::osyncstream syncStrm{strm};
  for (int i = 0; i < num ; ++i) {
    syncStrm << "squareroot of " << i << " is "
```

```
            << std::sqrt(i) << '\n' << std::flush_emit;

  }
}

int main()
{
  std::ofstream fs{"tmp.out"};
  std::jthread t1(squareRoots, std::ref(fs), 5);
  std::jthread t2(squareRoots, std::ref(fs), 5);
  std::jthread t3(squareRoots, std::ref(fs), 5);
}
```

This program uses three concurrent threads to write line-by-line to the same file opened at the beginning of the program.

Note that each thread uses its own synchronized output stream. However, they all have to use the same file stream. Thus, the program would not work if each thread opens the file itself.

17.5.4 Using Synchronized Output Streams as Output Streams

A synchronized output stream *is a* stream. Class `std::osyncstream` is derived from `std::ostream` (in fact and as usual for stream classes, class `std::basic_osyncstream<>` is derived from class `std::basic_ostream<>`). Therefore, you can also implement the program above as follows:

lib/syncfilestream2.cpp

```
#include <fstream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(std::ostream& strm, int num)
{
  for (int i = 0; i < num ; ++i) {
    strm << "squareroot of " << i << " is "
         << std::sqrt(i) << '\n' << std::flush_emit;

  }
}

int main()
{
  std::ofstream fs{"tmp.out"};
  std::osyncstream syncStrm1{fs};
  std::jthread t1(squareRoots, std::ref(syncStrm1), 5);
```

```
  std::osyncstream syncStrm2{fs};
  std::jthread t2(squareRoots, std::ref(syncStrm2), 5);
  std::osyncstream syncStrm3{fs};
  std::jthread t3(squareRoots, std::ref(syncStrm3), 5);
}
```

The manipulator `std::flush_emit` is defined for output streams in general and can be used here. For output streams that are not synchronized it has no effect.

Note that creating ***one*** synchronized output stream and passing it to all three threads would not work because then multiple threads would write to one stream:

```
// undefined behavior (concurrent writes to the same stream):
std::osyncstream syncStrm{fs};
std::jthread t1(squareRoots, std::ref(syncStrm), 5);
std::jthread t2(squareRoots, std::ref(syncStrm), 5);
std::jthread t3(squareRoots, std::ref(syncStrm), 5);
```

17.5.5 Synchronized Output Streams in Detail

******* This chapter/section is at work *******

17.6 Afternotes

Latches and barriers were first proposed by Alasdair Mackintosh in http://wg21.link/n3600. The finally accepted wording was formulated by Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen in http://wg21.link/p1135r6.

Semaphores were first proposed by Olivier Giroux in http://wg21.link/p0514r1. The finally accepted wording was formulated by Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen in http://wg21.link/p1135r6.

Atomic references were first proposed as `std::atomic_view<>` by H. Carter Edwards, Hans Boehm, Olivier Giroux, James Reus in http://wg21.link/p0019r0. The finally accepted wording was formulated by Daniel Sunderland, H. Carter Edwards, Hans Boehm, Olivier Giroux, Mark Hoemmen, D. Hollman, Bryce Adelstein Lelbach, and Jens Maurer in http://wg21.link/p0019r8. The API for `wait()` and notification was added as finally proposed by David Olsen in http://wg21.link/p1643r1.

Atomic shared pointers were first proposed as `atomic_shared_ptr<>` by Herb Sutter in http://wg21.link/n4058 and were adopted then for the concurrency TS (http://wg21.link/n4577). The finally accepted wording to integrate them to the C++ standard as partial specialization of `std::atomic<>` was formulated by Alisdair Meredith in http://wg21.link/p0718r2.

Atomic specializations for floating-point types were first proposed by H. Carter Edwards, Hans Boehm, Olivier Giroux, JF Bastien, James Reus in http://wg21.link/p0020r0. The finally accepted wording was formulated by H. Carter Edwards, Hans Boehm, Olivier Giroux, JF Bastien, James Reus in http://wg21.link/p0020r6.

Thread synchronization with atomic types was first proposed by Olivier Giroux in http://wg21.link/p0514r0. The finally accepted wording was formulated by Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen in http://wg21.link/p1135r6, http://wg21.link/p1643r1, and http://wg21.link/p1644r0.

Synchronized output streams were first proposed by Lawrence Crowl in http://wg21.link/n3750. The finally accepted wording was formulated by Lawrence Crowl, Peter Sommerlad, Nicolai Josuttis, and Pablo Halpern in http://wg21.link/p0053r7 and by Peter Sommerlad and Pablo Halpern in http://wg21.link/p0753r2.

This page is intentionally left blank

Chapter 18

Other C++ Standard Library Improvements

This chapter presents several other small features and extensions C++20 introduces in its standard library.

18.1 Updates for String Types

In C++20 some aspects of string types changed. These changes affect strings (type `std::basic_string<>` with its instantiations such as `std::string`), string views (`std::basic_string_view<>` with its instantiations such as `std::string_view`), or both.

In fact, C++20 introduces the following improvements for string types:

- All string types now support the spaceship operator `<=>`. For this, they now declare only `operator==` and `operator<=>` and no longer declare `operator!=`, `operator<`, `operator<=`, `operator>`, and `operator>=`.
- All string types now provide the new member functions `starts_with()` and `end_with()`.
- For strings, the member function `reverse()` can no longer be used to request to shrink the capacity (memory allocated for the value) of strings. For this reason, you can no longer pass no argument to `reserve()`.
- For UTF-8 characters, C++ provides now the string types `std::u8string` and `std::u8string_view`. They are defined as `std::basic_string<>` and `std::basic_string_view<>` for the new UTF-8 character type `char8_t`. For this reason, library functions returning a UTF-8 string now have the return type `std::u8string`. Note that this change might break existing code when switching to C++20.
- Strings (`std::string` and other instantiations of `std::basic_string<>`) are `constexpr` now so that you can use strings at compile time.

  Note that you cannot use a `std::string` at both compile time and runtime. However, there are ways to export a compile-time string to the runtime.
- String views are now marked as views and borrowed ranges.

- Standard hash functions were added for types `std::u8string` and `std::u8string_view` as well as for `std::pmr::string`, `std::pmr::u8string`, `std::pmr::u16string`, `std::pmr::u32string`, and `std::pmr::wstring`.

The sections explain the non-trivial improvements that are not introduced and explained in other chapters.

18.2 String Members `starts_with()` and `ends_with()`

Both strings and string views now have new member functions `starts_with()` and `ends_with()`. They provide an easy way to check the leading and trailing character of a string against a certain sequence of characters. You can compare against a single character, an array of characters, or a string or string view.

For example:

```
void foo(const std::string& s, std::string_view suffix)
{
  if (s.starts_with('.')) {
    ...
  }
  if (s.ends_with(".tmp")) {
    ...
  }
  if (s.ends_with(suffix)) {
    ...
  }
}
```

18.3 Restricted String Member `reserve()`

For strings, the member function `reverse()` can no longer be used to request to shrink the capacity (memory allocated for the value) of strings:

```
void modifyString(std::string& s)
{
  if (...) {
    s.clear();
    s.reserve(0);          // no longer may shrink memory (as before on some platforms)
    return;
  }
  ...
}
```

The reason is that releasing might take some time so that the performance of this call could vary significantly when porting code.

For this reason, passing no argument even is no longer supported:

```
    s.reserve();           // ERROR since C++20
```

Use `shrink_to_fit()` instead:

```
s.shrink_to_fit();   // still OK
```

even no longer compiles. The have the previous effect

18.4 New Utility Functions

18.4.1 `ssize()`

Pretty often we need the size of a collection or array or range as a signed value. For example, to avoid a warning here:

```
for (int i = 0; i < coll.size(); ++coll) {        // possible warning
  ...
}
```

The problem is that `size()` yields an unsigned value and comparisons between signed and unsigned value might fail with pretty high values.

While you can declare `i` as unsigned int or `std::size_t`, there might be a good reason to use it as `int`.

A helper function `std::ssize()` was introduced that allows the following use instead:

```
for (int i = 0; i < std::ssize(coll); ++coll) {   // usually no warning
  ...
}
```

Thanks to ADL, it is enough to write the following when a standard container or other standard types are used:

```
for (int i = 0; i < ssize(coll); ++coll) {        // OK for std types
  ...
}
```

Note that this sometimes is even necessary to avoid compile-time errors. An example is the initialization of latches, barriers, and semaphores.

Also note that the ranges library provides `ssize()` in namespace `std::ranges`.

18.5 `std::source_location`

Sometimes it is important to let the program deal with the location of the source code that is currently being processed. This is especially used for logging, testing, and checking of invariants. So far, programmers have had to use the C preprocessor macros `__FILE__`, `__LINE__`, and `__func__`. C++20 introduces a type safe feature for that so that an object can be initialized with the current source location and this information can be passed around just like any other object.

The use is simple:

```
#include <source_location>

void foo()
```

```
{
  auto sl = std::source_location::current();
  ...
  std::cout << "file:     " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}
```

The static `consteval` function `std::source_location::current()` yields an object for the current source location of type `std::source_location` with the following interface:

- `file_name()` yields the name of the file.
- `function_name()` yields the name of the function (empty if called outside any function).
- `line()` yields the line number (may be `0` when the line number is not known).
- `column()` yields the column in the line (may be `0` when the column number is not known).

Details such as the exact format of the function name and the exact position of the column might differ. For example, with the GCC's library the output might look as follows:

```
file:     sourceloc.cpp
function: void foo()
line/col: 8/42
```

while the output of Visual C++ might look as follows:

```
file:     sourceloc.cpp
function: foo
line/col: 8/35
```

Note that by using `std::source_location::current()` as a default argument in a parameter declaration, you get the location of the function call. For example:

```
void bar(std::source_location sl = std::source_location::current())
{
  ...
  std::cout << "file:     " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}

int main()
{
  ...
  bar();
  ...
}
```

The output might be something like:

```
file:     sourceloc.cpp
function: int main()
```

```
line/col: 34/6
```

or:

```
file:     sourceloc.cpp
function: main
line/col: 34/3
```

Because `std::source_location`'s are objects, you can store them in containers and pass them around:

```
std::source_location myfunc()
{
  auto sl = std::source_location::current();
  ...
  return sl;
}

int main()
{
  std::vector<std::source_location> locs;
  ...
  locs.push_back(myfunc());
  ...
  for (const auto& loc : locs) {
    std::cout << "called: " << loc.function_name() << '\n';
  }
}
```

The output might be:

```
called: 'std::source_location myfunc()'
```

or:

```
called: 'myfunc'
```

See *lib/sourceloc.cpp* for the complete example.

18.6 New Type Traits

Table *New Type Traits* lists the new type traits introduced with C++20.

In addition, the type trait `is_pod<>` was deprecated.

The following paragraphs discuss these traits in detail.

Trait	Effect
`is_bounded_array<T>`	Yields true if type *T* is an array type with known extent
`is_unbounded_array<T>`	Yields true if type *T* is an array type with unknown extent
`is_nothrow_convertible<T, T2>`	Yields true if *T* is convertible to type *T2* without throwing
`is_layout_compatible<T1, T2>`	Yields true if *T1* is layout compatible with type *T2* so that `reinterpret_cast<>` of pointers is defined
`is_layout_pointer_interconvertible...` `_base_of<BaseT, DerT>`	Yields true if a pointer to *DerT* can safely be converted to a pointer to its base type *BaseT*
`remove_cvref<T>`	Yields type *T* without being a reference, `const`, and `volatile`
`unwrap_reference<T>`	Yields the wrapped type of type `T` if it is a `std::reference_wrapper<>` or otherwise `T`
`unwrap_ref_decay<T>`	Yields the wrapped type of type `T` if it is a `std::reference_wrapper<>` or otherwise the decayed type of `T`
`common_reference<T...>`	Yields the common type (if any) of all types *T*... to which you can assign a value
`type_identity<T>`	Yields type *T* as it is
`is_clock<T>`	Yields true if *T* is a clock type

Table 18.1. New Type Traits

18.6.1 Type Traits `is_bounded_array<>` and `is_unbounded_array`

```
std::is_bounded_array<T>::value
std::is_unbounded_array<T>::value
```

yields whether type *T* is a bounded/unbounded array (extent known/unknown).

For example:

```
int a[5];
std::is_bounded_array_v<decltype(a)>      // true
std::is_unbounded_array_v<decltype(a)>    // false

extern int b[];
std::is_bounded_array_v<decltype(b)>      // false
std::is_unbounded_array_v<decltype(b)>    // true
```

18.6.2 Type Trait `is_nothrow_convertible<>`

`std::is_nothrow_convertible<`*`From`*`, `*`To`*`>::value`

yields whether type *`From`* is convertible to type `To` with the guarantee not to throw any exception.

For example:

```
// char* to std::string:
std::is_convertible_v<char*, std::string>                  // true
std::is_nothrow_convertible_v<char*, std::string>          // false

// std::string to std::string_view:
std::is_convertible_v<std::string, std::string_view>       // true
std::is_nothrow_convertible_v<std::string, std::string_view> // true
```

18.6.3 Type Trait `is_layout_compatible<>`

`std::is_layout_compatible<`*`T1`*`, `*`T2`*`>::value`

yields whether types *`T1`* and *`T2`* are *layout compatible* so that you can safely convert pointers to them with `reinterpret_cast`.

18.6.4 Type Trait `is_layout_pointer_interconvertible_base_of<>`

`std::is_layout_pointer_interconvertible_base_of<`*`Base`*`, `*`Der`*`>::value`

yields true if a pointer to type *`Der`* can safely be converted to a pointer to its base type *`Base`* with `reinterpret_cast`.

If both are the same types it always yields `true`.

18.6.5 Type Trait `remove_cvref<>`

`std::remove_cvref<`*`T`*`>::type`

Yields type `T` without being a reference, `const`, and `volatile`.

The expression

`std::remove_cvref_t<T>`

is equivalent to:

`std::remove_cv_t<remove_reference_t<T>>`.

18.6.6 Type Traits `unwrap_reference<>` and `unwrap_ref_decay`

`std::unwrap_reference<`*`T`*`>::type`

Yields the wrapped type of `T` if it is a `std::reference_wrapper<>` (created with `std::ref()` or `std::cref()`) or otherwise `T`.

`std::unwrap_ref_decay<`*`T`*`>::type`

Yields the wrapped type of T if it is a `std::reference_wrapper<>` (created with `std::ref()` or `std::cref()`) or otherwise the decayed type of T.

For example:

```
std::unwrap_reference_t<decltype(std::ref(s))>    // std::string&
std::unwrap_reference_t<decltype(std::cref(s))>   // const std::string&
std::unwrap_reference_t<decltype(s)>              // std::string
std::unwrap_reference_t<decltype(s)&>             // std::string&
std::unwrap_reference_t<int[4]>                   // int[4]

std::unwrap_ref_decay_t<decltype(std::ref(s))>    // std::string&
std::unwrap_ref_decay_t<decltype(std::cref(s))>   // const std::string&
std::unwrap_ref_decay_t<decltype(s)>              // std::string
std::unwrap_ref_decay_t<decltype(s)&>             // std::string
std::unwrap_ref_decay_t<int[4]>                   // int*
```

18.6.7 Type Trait `common_reference<>`

`std::common_reference<`*T*`...>::type`

Yields the common type (if any) of all types *T*... to which you can assign a value. So, given types T1, T2, and T3, the trait yields the type where you can assign values of all three types. Ideally, it is a reference type. However, if a type conversion is involved that creates a temporary object, it is a value type.

For example:

```
std::common_reference_t<int&, int>                            // int
std::common_reference_t<int&, int&>                           // int&
std::common_reference_t<int&, int&&>                          // const int&
std::common_reference_t<int&&, int&&>                         // int&&
std::common_reference_t<int&, double>                         // double
std::common_reference_t<int&, double&&>                       // double
std::common_reference_t<char*, std::string, std::string_view> // std::string_view
std::common_reference_t<char, std::string>                    // ERROR
```

18.6.8 Type Trait `type_identity<>`

`std::type_identity<`*T*`>::type`

yields just type T.

This type trait has a surprising number of use cases:

- You can disable that a parameter is used to deduce a template parameter. For example:

```
template<typename T>
void insert(std::vector<T>& coll, const std::type_identity_t<T>& value)
{
   coll.push_back(value);
}
```

```
std::vector<double> coll;
...
insert(coll, 42);      // OK: type of 42 not used to deduce type T
```

If the parameter `value` would be declared just with `const T&`, the compiler would raise an error because it would deduce two different types for type `T`.

- You can use it as building block to define type traits that yield types. For example, you could define a type trait that removes constness simply as follows:[1]

```
template<typename T>
struct remove_const : std::type_identity<T> {
};

template<typename T>
struct remove_const<const T> : std::type_identity<T> {
};
```

18.6.9 `is_pointer_interconvertible_with_class<>()` and `is_corresponding_member<>()`

To check the relationship between class members, C++20 introduces two new functions:

```
template<typename S, typename M>
constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;

template<typename S1, typename S2, typename M1, typename M2>
constexpr bool is_corresponding_member(M1 S1::*m1, M2 S2::*m2) noexcept;
```

As the C++20 standard notes: The type of a pointer-to-member expression `&C::b` is not always a pointer to member of C, leading to potentially surprising results when using these functions in conjunction with inheritance.

```
struct A { int a; };                  // a standard-layout class
struct B { int b; };                  // a standard-layout class
struct C: public A, public B { };     // not a standard-layout class

std::is_pointer_interconvertible_with_class(&C::b)       // true
  // true because, despite its appearance, &C::b has type "pointer to member of B of type int"

std::is_pointer_interconvertible_with_class<C>(&C::b)    // false
  // false because it forces the use of class C and fails

std::is_corresponding_member(&C::a, &C::b)               // true
```

[1] See the talk *Modern Template Metaprogramming: A Compendium* by Walter E. Brown at CppCon 2014 (http://www.youtube.com/watch?v=Am2is2QCvxY) for a source of this example.

```
  // true because, despite its appearance, &C::a and &C::b have types "pointer to member of A of type int" and
  // "pointer to member of B of type int", respectively

std::is_corresponding_member<C, C>(&C::a, &C::b)          // false
  // false because it forces the use of class C and fails
```

******* This chapter/section is at work *******

18.7 Mathematical Constants

C++20 introduces constants for the most important mathematical floating-point constants. Table *Math Constants* lists them.

Constant	Template
`std::numbers::e`	`std::numbers::e_v<>`
`std::numbers::pi`	`std::numbers::pi_v<>`
`std::numbers::inv_pi`	`std::numbers::inv_pi_v<>`
`std::numbers::inv_sqrtpi`	`std::numbers::inv_sqrtpi_v<>`
`std::numbers::sqrt2`	`std::numbers::sqrt2_v<>`
`std::numbers::sqrt3`	`std::numbers::sqrt3_v<>`
`std::numbers::inv_sqrt3`	`std::numbers::inv_sqrt3_v<>`
`std::numbers::log2e`	`std::numbers::log2e_v<>`
`std::numbers::log10e`	`std::numbers::log10e_v<>`
`std::numbers::ln2`	`std::numbers::ln2_v<>`
`std::numbers::ln10`	`std::numbers::ln10_v<>`
`std::numbers::egamma`	`std::numbers::egamma_v<>`
`std::numbers::phi`	`std::numbers::phi_v<>`

Table 18.2. Math Constants

The constants are provided in header **<numbers>** in namespace `std::numbers`.

The constants are specializations for type `double` of corresponding variable templates that have the suffix `_v`. The values are the nearest representable values of the corresponding type. For example:

```
namespace std::number {
  template<std::floating_point T> inline constexpr T pi_v<T> = ... ;
  inline constexpr double pi = pi_v<double>;
}
```

As you can see, the definitions use the i`std::floating_point` concept (which was introduced for that reason).

Therefore, you can use them as follows:

```
#include <numbers>
...
```

```
double area1 = rad * rad * std::numbers::pi;

long double area2 = rad * rad * std::numbers::pi_v<long double>;
```

***** **This chapter/section is at work** *****

18.8 Utilities to Deal with Bits

C++20 provides better and cleaner support for dealing with bits:

- Missing low-level bit operations
- Bit casts
- Checks for the endianess of a platform

All of these utilities are defined in header `<bit>`.

18.8.1 Bit Operations

Hardware usually has special support for bit operations such as "rotate left" or "rotate right." However, before C++20, C++ programmers did not have direct access to these instructions. The newly introduced bit operations provide a direct API to the bit instructions of the underlying CPU.

Table *Bit Operations* lists all standardized *bit operations* C++20 introduced. They are provided in header file `<bit>` as free-standing functions in namespace `std`.

Operation	Meaning
`rotl(`*val*`,` *n*`)`	Yields *val* with *n* bits rotated to the left
`rotr(`*val*`,` *n*`)`	Yields *val* with *n* bits rotated to the right
`countl_zero(`*val*`)`	Yields number of leading (most significant) `0` bits
`countl_one(`*val*`)`	Yields number of leading (most significant) `1` bits
`countr_zero(`*val*`)`	Yields number of trailing (least significant) `0` bits
`countr_one(`*val*`)`	Yields number of trailing (least significant) `1` bits
`popcount(`*val*`)`	Yields number of `1` bits in the value
`has_single_bit(`*val*`)`	Yields whether *val* is a power of `2` (one bit set)
`bit_floor(`*val*`)`	Yields previous power-of-two value
`bit_ceil(`*val*`)`	Yields next power-of-two value
`bit_width(`*val*`)`	Yields number of bits necessary to store the value

Table 18.3. Bit Operations

Consider, for example, the following program:

lib/bitops8.cpp

```
#include <iostream>
```

```
#include <format>
#include <bitset>
#include <bit>

int main()
{
  std::uint8_t i8 = 0b0000'1101;
  std::cout
    << std::format("{0:08b} {0:3}\n", i8)                  // 00001101
    << std::format("{0:08b} {0:3}\n", std::rotl(i8, 2))    // 00110100
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, 1))    // 10000110
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, -1))   // 00011010
    << std::format("{}\n", std::countl_zero(i8))           // 4 leading zeros
    << std::format("{}\n", std::countr_one(i8))            // 1 trailing one
    << std::format("{}\n", std::popcount(i8))              // 3 ones
    << std::format("{}\n", std::has_single_bit(i8))        // false
    << std::format("{0:08b} {0:3}\n", std::bit_floor(i8)) // 00001101
    << std::format("{0:08b} {0:3}\n", std::bit_ceil(i8))  // 00001101
    << std::format("{}\n", std::bit_width(i8));            // 4
}
```

The program has the following output:

```
00001101  13
00110100  52
10000110 134
00011010  26
4
1
3
false
00001000   8
00010000  16
4
```

Note the following:

- All bit operations except the `std::bit_cast<>`
- Note that the rotate functions also take a negative *n*, which means that the rotation changes its direction.
- All functions returning a count have th return type `int`. The only exception is `bit_width()`, which returns a value of the passed type, which is an inconsistency (I would assume a bug) in the standard. So when using it as an `int` or printing it directly, you might have to use a static cast.

If you run a corresponding program for a `std::uint16_t` (see *lib/bitops16.cpp*) you get the following output:

```
0000000000001101     13
0000000000110100     52
```

```
1000000000000110 32774
0000000000011010    26
12
1
3
false
0000000000001000     8
0000000000010000    16
4
```

Note also that these functions are only defined for *unsigned integral types*. That means:[2]

- You cannot use the bit operations for signed integral types:

```
int b1 =  ... ;
auto b2 = std::rotl(b1, 2);    // ERROR
```

- You cannot use the bit operations for type `char`:

```
char b1 =  ... ;
auto b2 = std::rotl(b1, 2);    // ERROR
```

  Type `unsigned char` works fine.

- You cannot use the bit operations for type `std::byte`:

```
std::byte b1{... };
auto b2 = std::rotl(b1, 2);    // ERROR
```

Table *Hardware Support for Bit Operations*, taken from `http://wg21.link/p0553r4` list the possible mapping of some of the new bit operation to existing hardware.

Operation	Intel/AMD	ARM	PowerPC
`rotl()`	`ROL`	–	`rldicl`
`rotr()`	`ROR`	`ROR`, `EXTR`	–
`popcount()`	`POPCNT`	–	`popcntb`
`countl_zero()`	`BSR`, `LZCNT`	`CLZ`	`cntlzd`
`countl_one()`	–	`CLS`	–
`countr_zero()`	`BSF`, `TZCNT`	–	–
`countr_one()`	–	–	–

Table 18.4. Hardware Support for Bit Operations

[2] Thanks to JeanHeyd Meneide for pointing that out.

18.8.2 std::bit_cast<>()

C++20 provides a new cast operations to change the type of a sequence of bits. Unlike when using reinterpret_cast<> or union's, the new operator std::bit_cast<> ensures that the number of bits fits, a standard layout is used, and no pointer type is used.

For example:

```
std::uint8_t b8 = 0b0000'1101;

auto bc = std::bit_cast<char>(b8);              // OK
auto by = std::bit_cast<std::byte>(b8);         // OK
auto bi = std::bit_cast<int>(b8);               // ERROR: wrong number of bits
```

18.8.3 std::endian

C++20 introduces a new utility enumeration type std::endian which can be used to check the endianess of the execution environment. It introduces three enumeration values:

- std::endian::big a value that stands for "big-endian" (scalar types are stored which the most significant byte placed first and the rest in descending order).
- std::endian::little a value that stands for "little-endian" (scalar types are stored which the least significant byte placed first and the rest in ascending order).
- std::endian::native a value that specifies the endianess of the execution environment.

If all scalar types are big-endian, std::endian::native is equal to std::endian::big. If all scalar types are little-endian, std::endian::native is equal to std::endian::little. Otherwise, std::endian::native has a value that is neither std::endian::big nor std::endian::little.

If all scalar types have size 1, then all of endian::little, endian::big, and endian::native have the same value.

The type is defined in header <bit>

As enumeration values these values can be used at compile-time. For example:

```
#include <bit>
...

if constexpr (std::endian::native == std::endian::big) {
   ...      // handle big-endian
}
else if constexpr (std::endian::native == std::endian::little) {
   ...      // handle little-endian
}
else {
   ...      // handle mixed endian
}
```

18.9 <version>

C++20 introduces a new header file **<version>**. This header file provides no active functionality. Instead, it shall provide all implementation-specific general information about the C++ standard library being used.

For example, <version> might contain:

- The version and release date of the C++ standard library
- Copyright information

In addition, <version> defines the feature test macros of the C++ standard library (they are also defined in the header files they apply to).

Because this header file is short and fast to load, tools can include this header file to get all necessary information to make decisions based on the provided feature set or find all information they need to deal with general information of the library used.

Adding the unseq Execution Policy for Algorithms

C++17 introduced various execution policies for the newly introduced parallel algorithms. You could enable parallel computing and allow threads to operate on multiple data items in parallel (called *vectorization* or *SIMD processing*).

However, you could not allow algorithms to operate on multiple data items but restrict the algorithms to use only one thread. For this, C++20 now provides the execution policy `std::execution::unseq`. As usual, the new execution policy is a `constexpr` object of a corresponding unique class `unsequenced_policy` in namespace `std::execution`. Table *Execution Policies* lists all standardized *execution policies* now supported.

Policy	Meaning
`std::execution::seq`	Sequential execution of single data with one thread
`std::execution::par`	Parallel execution of single data with multiple threads
`std::execution::unseq`	Parallel execution of multiple data with one thread (since C++20)
`std::execution::par_unseq`	Parallel execution of multiple data with multiple threads

Table 18.5. Execution Policies

The unsequenced execution policy allows the execution of the operations passed to access the elements to be interleaved on a single thread of execution. You should not use this policy with blocking synchronization (such as using mutexes) because that might result in deadlocks.

Here is an example using it:

lib/unseq.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <execution>
```

```
int main (int argc, char** argv)
{
  // init number of argument from command line (default: 1000):
  int numElems = 1000;
  if (argc > 1) {
    numElems = std::atoi(argv[1]);
  }

  // init vector of different double values:
  std::vector<double> coll;
  coll.reserve(numElems);
  for (int i=0; i<numElems; ++i) {
    coll.push_back(i * 4.37);
  }

  // process square roots:
  // - allow SIMD processing but only one thread
  std::for_each(std::execution::unseq,      // since C++20
                coll.begin(), coll.end(),
                [](auto& val) {
                  val = std::sqrt(val);
                });

  for (double value : coll) {
    std::cout << value << '\n';
  }
}
```

As usual for execution policies, you have no way to impact when and how the policy is used. With this policy you ***enable*** vectorization or SIMD computing; however, you do not mandate it. On hardware not supporting this policy or if the implementation decides at runtime not to use it (e.g., because the CPU load is too high) the `unseq` policy results in sequential execution.

18.10 Afternotes

The string member functions `starts_with()` and `ends_with()` were introduced as finally formulated by Mikhail Maltsev in http://wg21.link/p0457r2.

The restriction of the `reserve()` function of strings was first proposed by Andrew Luo in http://wg21.link/lwg2968. The finally accepted wording was formulated by Mark Zeren and Andrew Luo in http://wg21.link/p0966r1.

The `ssize()` functions were discussed for a long time as a fix to the problem of comparing signed indexes with unsigned sizes. We would like to let all `size()` functions yield a signed value; however, there is the backward compatibility problem. As a reaction to a `std::span` proposal to use a signed size type,

`ssize()` was first proposed by Robert Douglas, Nevin Liber, and Marshall Clow in http://wg21.link/p1089r0. The finally accepted wording was formulated by Jorg Brown in http://wg21.link/p1227r2. `std::ranges::ssize()` was later added by Hannes Hauswedell, Jorg Brown, and Casey Carter in http://wg21.link/p1970r2.

Access to the source location was first proposed by Robert Douglas in http://wg21.link/n3972. The finally accepted wording was formulated by Robert Douglas, Corentin Jabot, Daniel Krügler, and Peter Sommerlad in http://wg21.link/p1208r6.

The type traits `is_bounded_array<>` and `is_unbounded_array<>` were accepted as proposed by Walter E. Brown and Glen J. Fernandes in http://wg21.link/p1357r1.

The type trait `is_nothrow_convertible<>` was accepted as proposed by Daniel Krügler in http://wg21.link/p0758r1.

The type trait `common _reference<>` was accepted as part of the ranges library by Eric Niebler, Casey Carter, and Christopher Di Bella in http://wg21.link/p0896r4.

The type traits `unwrap _reference<>` and `unwrap_ref_decay<>` were accepted as proposed by Vicente J. Botet Escriba in http://wg21.link/p0318r1.

The type trait `remove_cvref<>` was accepted as proposed by Walter E. Brown in http://wg21.link/p0550r2.

The type trait `type_identity<>` was accepted as proposed by Timur Doumler in http://wg21.link/p0887r1.

The type traits `is_layout_compatible<>` `is_pointer_interconvertible_base_of<>` and the function templates `is_pointer_interconvertible_with_class<>()` `is_corresponding_member<>()` were accepted as proposed by Lisa Lippincott in http://wg21.link/p0466r5.

The mathematical constants were first proposed by Lev Minkovsky in http://wg21.link/p0631r0. The finally accepted wording was formulated by Lev Minkovsky and John McFarlane in http://wg21.link/p0631r8.

Bit operations were first proposed by Matthew Fioravante in http://wg21.link/n3864. The finally accepted wording was formulated by Jens Maurer in http://wg21.link/p0553r4 and http://wg21.link/p0556r3. The names were later modified as proposed by Vincent Reverdy in http://wg21.link/p1956r1

`std::bit_cast<>()` was introduced as finally proposed by JF Bastien in http://wg21.link/p0476r2.

`std::endian` was first proposed by Howard Hinnant in http://wg21.link/p0463r0. The finally accepted wording was formulated by Howard Hinnant in http://wg21.link/p0463r1. The header file was later corrected as proposed by Walter E. Brown and Arthur O'Dwyer in http://wg21.link/p1612r1.

Header `version>` was first proposed by Alan Talbot in http://wg21.link/p0754r0. The finally accepted wording was formulated by Alan Talbot in http://wg21.link/p0754r2.

The `unseq` execution policy was first proposed by Arch D. Robison, Pablo Halpern, Robert Geva, and Clark Nelson, in http://wg21.link/p0076r0. The finally accepted wording was formulated by Alisdair Meredith and Pablo Halpern in http://wg21.link/p1001r2.

This page is intentionally left blank

Chapter 19

Modules

This chapter presents the new C++20 feature *modules*. Modules provide a way to combine code from multiple files into one logical entity (module, component). As for classes, data encapsulation helps to provide a clear definition of the API of the module. As a side effect, it is possible to ensure that module code does not have to be compiled multiple times even if it is placed in "header files."

This chapter was written with tremendous help and support by Daniela Engert and Hendrik Niemeyer, who also gave great introductions about this topic at the Meeting C++ 2020 and the ACCU 2021 conference, respectively.

19.1 Motivation of Modules by a First Example

Modules allow programs to define an API for code in the large. The code might consist of multiple classes, multiple files, several functions, and various auxiliary utilities including templates. By using the keyword `export` you can exactly specify at one specific location what is exported as the API of a module wrapping all code providing a certain functionality. Thus, we can define a clean API of a component implemented in various files.

Let us look at a few simple examples that declares a module in one file and uses this module then in another file.

19.1.1 Implementing and Exporting a Module

The specification of the API of a module is defined in its ***primary module interface unit***, which each module has exactly once:

lang/mod0.cppm

```
export module Square;  // declare module Square

int square(int i);

export class Square {
```

```
 private:
   int value;
 public:
   Square(int i)
    : value{square(i)} {
   }
   int getValue() const {
     return value;
   }
};

export template<typename T>
Square toSquare(const T& x) {
  return Square{x};
}

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

First, you might recognize that the file uses a new file extension: **`.cppm`**. The file extension of module files is currently not clear yet. We will discuss the handling of module files by compilers later.

The key entry for the primary module interface is the line declaring and exporting the module using the name `Square`:

```
export module Square;   // declare module Square
```

Note that the name is only used as an identifier to import the module. It does ***not*** introduce a new scope or namespace. Any name exported by the module is still in the scope it has when being exported.

The name of a module may contain dots, but there is no special meaning for the dot:

```
export module Math.Square;   // declare module "Module.Square"
```

This has no other effect than naming the module "`ModuleDotSquare`." Dots can be used to signal some logical relationships between modules; however, using them has no syntactic or formal consequences.

The public API of a module is defined by everything that is explicitly exported using the keyword `export`. In this case we export the class `Square` and the function template `toSquare<>()`:

```
export class Square {
  ...
};

export template<typename T>
Square toSquare(const T& x) {
   ...
}
```

Everything else is ***not*** exported and cannot be used directly by code importing the module (we will discuss later how non-exported module stuff may be reachable but not visible). Therefore, the function `square()` declared without `export` is not usable by code importing this module.

The file looks like a header file with the following differences:

- We have the line with the module declaration.
- We have symbols, types, functions (even templates) that are exported with `export`
- We do not need `inline` to define functions
- We do not need preprocessor guards

Each module must have exactly one *primary module interface unit* file with its specified name. As you can see, the name of the module does ***not*** conflict with any symbol *in* the module. Therefore, a module can have the name of its (major) namespace, class, or function (note that a module does *not* automatically introduce a new namespace; you have to implement that explicitly).

19.1.2 Importing and Using a Module

To use the module (or its code) in a program, you have to import the module under its name. Here is a simple example of a program just using the module `Square` define above:

lang/mod0main.cpp

```
#include <iostream>

import Square;  // import module "Square"

int main()
{
  Square x = toSquare(42);
  std::cout << v.getValue() << '\n';
}
```

With

```
import Square;  // import module "Square"
```

we import all exported stuff from the module `Square`. That means we can then use the exported class `Square` and the function template `toSquare<>()`.

Using anything from the module that is not exported results in a compile-time error:

```
import Square;  // import module "Square"

square(42)      // ERROR: square() not exported
```

Again, note that a module does *not* automatically introduce a new namespace. We use the exported symbols of the module in the scope they were when being exported. If you want to have all stuff from the module in its own namespace, you have to implement that in the module.

19.1.3 Reachable versus Visible

When using modules, a new distinction comes into play: *reachability* versus *visibility*.[1] When exporting data, we might not be able to see and directly use a name or symbol of the module; although we might be able to use it indirectly.

Symbols that are reachable but not visible can occur, when an exported API provides access to a type that is not exported. Consider the following example:

```
export module ModReach;    // declare module ModReach

struct Data {              // declare a type not exported
  int value;
};

export struct Customer {   // declare an exported type
 private:
  Data data;
 public:
  Customer(int i)
   : data{i} {
  }
  Data getData() const {   // yield not exported type
    return data;
  }
};
```

When importing this module, type `Data` is ***not visible*** and therefore not directly usable:

```
import ModReach;

Data d{11};                      // ERROR: type Data not exported
Customer c{42};
const Data& dr = c.getData();    // ERROR: type Data not exported
```

However, type `Data` is ***reachable*** and that way indirectly usable:

```
import ModReach;

Customer c{42};
const auto& dr = c.getData();    // OK: type Data is used
auto d = c.getData();            // OK: d has type Data
std::cout << d.value << '\n';    // OK: type Data is used
```

You can even declare an object of type `Data` as follows:

```
decltype(std::declval<Customer>().getData()) d;   // d has non-exported type Data
```

[1] Thanks to Daniela Engert for pointing this out.

The code uses `std::declval<>()` and `decltype` to declare `d` with the return type of `getData()` called for an assumed object of type `Customer`.

Private module fragments can be use to restrict the reachability of classes and functions.

19.1.4 Modules and Namespaces

As already mentioned, symbols of a module are imported in the same scope they are exported. In contrast to some other programming languages, C++ modules do ***not*** automatically introduce a namespace for a module.

You might therefore use the convention export everything in a module in a namespace with its name. You can do that in two ways:

- Specify what you want to export with `export` inside the namespace:

```
export module Square;  // declare module "Square"

namespace Square {
  int square(int i);

  export class Square {
    ...
  };

  export template<typename T>
  Square toSquare(const T& x) {
    ...
  }

  int square(int i) {  // not exported
    ...
  }
}
```

- Specify everything you want to export in a namespace declared with `export`:

```
export module Square;  // declare module "Square"

int square(int i);

export namespace Square {
  class Square {
    ...
  };

  template<typename T>
  Square toSquare(const T& x) {
    ...
```

```
  }
}

int square(int i) {  // not exported
  ...
}
```

You might still declare `square()` inside a namespace `Square` not declared with `export`.

In both cases, the module exports class `Square::Square` and `Square::toSquare<>()` (thus the namespace of the symbols is exported, even if not marked with `export`).

Using the module would now look as follows:

```
#include <iostream>

import Square;  // import module "Square"

int main()
{
  Square::Square x = Square::toSquare(42);
  std::cout << v.getValue() << '\n';
}
```

19.1.5 Modules and Filenames

In contrast to other programming languages, C++ also does not require that a module has a special file name or is in a special directory. Any C++ file can define a module (but only one) and the name of the module may have nothing to do with the name or location of the file.

Of course, it makes sense to keep filenames and module names somehow in sync; however, that decision is only a decision of the configuration management of your source code.

As written already, there are also no specific file extensions required for module files. This is also a question of the used conventions and tools. Nevertheless, as you saw I give advice to use special file extensions for module interfaces, which we discuss later in detail.

19.2 Modules with Multiple Files

The purpose of modules is to deal with code of significant size distributed over multiple files. In the future, modules will be used to wrap code of small, medium-size, and very large components consisting of 2, 10, or even 100 files. These files might be even provided and maintained by multiple programmers and teams.

To demonstrate the scalability of this approach and its benefits, let us now look how multiple files can be used to define a module that can be used/imported by other code. The code size of the example is still small so that you normally would not place the code in multiple files; however, we want to demonstrate the features with a very simple example.

19.2.1 Module Units

In general modules consist out of multiple ***module units***. Module units are translation units that belong to a module.

All module units have to be compiled in some way. Even if they contain only declarations, which would have been placed in header files in traditional code, some kind of pre-compilation is necessary. The files are transferred into an internal binary format, which can be used to avoid that you have to compile the same code again and again.

However, we have different ways to split a module into multiple files:

- ***Module implementation units*** allow programmers to provide just code that is compiled in the traditional way into object files.
- ***Internal partitions*** allow programmers to provide declarations and definitions that are only visible inside a module. However, even when only declared, they are pre-compiled to avoid multiple compilations.
- ***Interface partitions*** allow programmers to split the exported module API even in multiple files.

The next section introduces these different module units by example.

19.2.2 A Module with Multiple Implementation Units

The first example of a module that is provided by multiple files only uses multiple files to define/implement types and functions.

A Primary Module Interface with a Global Module Fragment

First, again we have ***the*** primary module interface unit that defines what we export:

lang/mod1/mod1.cppm

```
module;                 // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod1;   // module declaration

struct Order {
  int count;
  std::string name;
  double price;

  Order(int c, std::string n, double p)
   : count{c}, name{n}, price{p} {
  }
};

export class Customer {
```

```
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n)
   : name{n} {
  }
  void buy(std::string ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, std::string ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

This time the module starts just with `module;` to signal that we start with some preprocessor stuff for a module interface unit:

```
module;                 // start module unit with global module fragment

#include <iostream>
#include <string>
#include <vector>

export module Mod1;   // module declaration
...
```

Here we introduce a ***global module fragment*** where we can place preprocessor commands like `#define` and `#include`. Everything from this area is ***not*** exported (no macros, no declarations, no definitions).

Nothing else can be done before we really start the module with

```
export module mod1;   // module declaration
```

The things defined in the module are:

- An internal data structure `Order`:

  ```
  struct Order {
    ...
  };
  ```

  This data structure is for order entries. Each entry holds how many items were ordered, their name, and their price. The constructor ensures that we initialize all members.
- A class `customer`, which we export:

  ```
  export class Customer {
    ...
  ```

```
};
```

As you can see, we need the header files and the internal data structure `Order` to define class `Customer`. However, by not exporting them, they cannot be used directly by code importing this module.

For class `Customer`, the member functions `averagePrice()`, `sumPrice()`, and `print()` are only declared. We implement them in implementation files associated with the module.

Module Implementation Units

To provide source code that is not exported with a module, you can use ***module implementation units***. They serve like ordinary CPP translations units with a leading statement that they apply to a specific module.

We may have as many implementation units as we like. In our example, I provide two, one to implement numeric operations and one to implement I/O operations.

The module implementation unit for numeric operations looks as follows:

lang/mod1/mod1price.cpp

```
module Mod1;          // implementation unit of module Mod1

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  if (orders.empty()) {
    return 0.0;
  }
  return sumPrice() / orders.size();
}
```

The file is a module unit because it starts with a declaration introducing that this is a file of module `Mod1`:

```
module Mod1;
```

This declaration implicitly imports the primary module interface (but nothing else). Thus, the declarations of types `Order` and `Customer` are known and we can directly provide implementations of their member functions.

Note that a module implementation unit does ***not*** `export` anything. `export` is only allowed in a module interface, which is declared with `export module` (and remember that there is only one primary module interface allowed per module).

Again, a module implementation unit may start with a global module fragment, which we can see in the module implementation unit for I/O:

lang/mod1/mod1io.cpp

```
module;                 // start module unit with global module fragment

#include <iostream>
#include <format>

module Mod1;            // implementation unit of module Mod1

void Customer::print() const
{
  // print name:
  std::cout << name << ":\n";
  // print order entries:
  for (const auto& od : orders) {
    std::cout << std::format("{:3} {:14} {:6.2f} {:6.2f}\n",
                             od.count, od.name, od.price, od.count * od.price);
  }
  // print sum:
  std::cout << std::format("{:25} ------\n", ' ');
  std::cout << std::format("{:25} {:6.2f}\n", "    Sum:", sumPrice());
}
```

Here, we introduce the module with `module;` to have a global module fragment for the header files that we use in the implementation unit. `<format>` is the header file of the new formatting library.

As you can see, module implementation units use the file extension of traditional C++ translation units (most of the time `.cpp`). Compilers deal with them just like any other C++ code.

Using the Module

The code using the module looks as follows:

lang/mod1/testmod1.cpp

```
#include <iostream>

import Mod1;

int main()
{
  Customer c1{"Kim"};

  c1.buy("table", 59.90);
  c1.buy(4, "chair", 9.20);
```

```
  c1.print();
  std::cout << "  Average: " << c1.averagePrice() << '\n';
}
```

Here we use the exported class `Customer` from the primary module interface to create a customer, place some orders, print the customer with all orders, and print the value of the average order.

The program has the following output:

```
Kim:
  1 table           59.90  59.90
  4 chair            9.20  36.80
                          ------
    Sum:                  96.70
  Average: 48.35
```

Note that any trial to use type `Order` in the code importing the module results in a compile-time error.

19.2.3 Internal Module Partitions

In the example above, we have introduced a data structure `Order` that is used inside the module only. It looks like we have to declare it in the primary module interface to make it available to all implementation units, which of course does not really scale in large projects.

With ***internal partitions*** you can define such internal stuff of a module in separate files. Note that *partitions* can also be used to define the exported interface in a separate file, which we will discuss later.

Note that *internal partitions* are sometimes called *partition implementation units* which is based on the fact that in the C++20 standard they are officially called "*module implementation units* that are *module partitions*" and that sounds like they provide the implementations of interface partitions. They do not. They just serve like internal header files for a module, that may provide declarations and definitions.

Defining an Internal Partition

Using an internal partition, we can define the local type `Order` in its own module unit as follows:

lang/mod2/mod2order.cppp

```
module;                 // start module unit with global module fragment

#include <string>

module Mod2:Order;      // internal partition declaration

struct Order {
  int count;
  std::string name;
  double price;
```

```
  Order(int c, std::string n, double p)
   : count{c}, name{n}, price{p} {
  }
};
```

As you can see a partition has the name of the module, then a colon, and then its name:

```
module Mod3:Order;
```

You might again recognize that the file uses another new file extension: **.cppp**, which we discuss later after looking at its contents.

Now the primary module interface simply imports this partition:

lang/mod2/mod2.cppm

```
module;                 // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod2;     // module declaration

import :Order;          // import internal partition Order

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n)
   : name{n} {
  }
  void buy(std::string ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, std::string ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
};
```

The primary module interface has to import the internal partition because it uses type `Order`. With that it is available in all implementation units of the module. It the primary module interface does not need type `Order` and does not import the internal partition, all module units that need type `Order` would have to import the internal partition directly.

Note that partitions are only an internal implementation aspect of a module. Whether or not code is in a partition has no impact on code that imports a module.

19.2.4 Interface Partitions

You can also split the interface of a module in multiple files. In that case you declare ***interface partitions***, which themselves export whatever should be exported.

Interface partitions are especially useful if modules provide multiple interfaces maintained by different programmers and/or teams. However, for simplicity let us just use the current example to demonstrate how to use this feature by only defining the `Customer` interface in a separate file.

To define only the `Customer` interface we can provide the following file:

lang/mod3/mod3customer.cppm

```
module;                          // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod3:Customer;     // interface partition declaration

import :Order;                   // import internal partition to use Order

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n)
   : name{n} {
  }
  void buy(std::string ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, std::string ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

The partition is more or less the former primary module interface with only one difference:

- As a partition we declare its name after the module name and a colon: `Mod3:Customer`

Like the primary module interface:

- We `export` this module partition:

```
export module Mod3:Customer;
```

- We use the new file extension **`.cppm`**, which we again discuss later

However, still the primary module interface is the only place to specify what a module exports. So, the primary module interface has to specify that the exported stuff from the partition `Customer` is exported by the module as a whole.

lang/mod3/mod3.cppm

```
export module Mod3;          // module declaration

export import :Customer;     // import and export interface partition Customer
...                          // import and export other interface partitions
```

By importing the partition interface and exporting it at the same time (you have to write both keywords), the primary module exports the interface of the partition `Customer` as its own interface:

```
export import :Customer;   // import and export partition Customer
```

Importing it without exporting it is not allowed.

Again, note that partitions are only an internal implementation aspect of a module. It does not matter whether interfaces and implementations are provided in partitions. Partitions do not create a new scope.

Therefore, for the implementation of the member functions of `Customer` moving the declaration of the class to a partition does not matter. You implement a member function of class `Customer` being part of module `Mod3`:

lang/mod3/mod3io.cpp

```
module;                 // start module unit with global module fragment

#include <iostream>
#include <vector>
#include <format>

module Mod3;            // implementation unit of module Mod3

import :Order;          // import internal partition to use Order

void Customer::print() const
{
  // print name:
  std::cout << name << ":\n";
  // print order entries:
  for (const Order& od : orders) {
    std::cout << std::format("{:3} {:14} {:6.2f} {:6.2f}\n",
                             od.count, od.name, od.price, od.count * od.price);
  }
```

```
  // print sum:
  std::cout << std::format("{:25} ------\n", ' ');
  std::cout << std::format("{:25} {:6.2f}\n", "    Sum:", sumPrice());
}
```

However, there is once difference in this implementation unit: because the primary module interface no longer imports the internal partition `:Order`, this module has to do so, because it uses type `Order`.

For code importing the module the way the code is distributed internally also does not matter. We still export class `Customer` in the global scope:

lang/mod3/testmod3.cpp

```
#include <iostream>

import Mod3;

int main()
{
  Customer c1{"Kim"};

  c1.buy("table", 59.90);
  c1.buy(4, "chair", 9.20);

  c1.print();
  std::cout << "  Average: " << c1.averagePrice() << '\n';
}
```

This example, demonstrates how you can define huge modules having multiple (sub-)interfaces, internal declarations and definitions, and using many files to implement it.

- Particular exported interfaces can be specified in interface partitions.
- Internal declarations can be specified in internal partitions.
- Definitions and implementations can be performed in module implementation units.
- You provide the primary module interface only to bring all together and specify what gets exported.
- It imports and exports all interface partitions

19.2.5 How Modules Replace Traditional Code

******* This chapter/section is at work *******

19.2.6 Module Declaration/Export in Detail

******* This chapter/section is at work *******

19.2.7 Module Import in Detail

******* This chapter/section is at work *******

19.2.8 Dealing with Module Files in Compilers

In C++, extensions are not standardized. In practice, different file extensions are used (usually `.cpp` and `.hpp`, but also `.cc`, `.cxx`, `.C`, `,hh`, `.hxx`, `.H`, and `.h` are used).

We also have no standard extension for modules. Even worse, we do not even agree on whether new extensions are necessary (yet). Therefore, different file extensions are used for module files in practice: `.cppm`, `.ixx`, and `.cpp`.

There are multiple reasons why different file extensions seem to be necessary:

- Compilers require different command-line options for dealing with different types of module files
- Similar to header files, you have to provide ***some*** of the module files to customers and third-party code.
- Different module files create different artifacts, which you might have to deal with (e.g., when removing the generated artifacts).

I personally have no ***final*** decision and recommendation for file extensions of module files yet. However, according to the current situation, I recommend the following:

- For **interface files** (both primary module interfaces and interface partitions), use the file extension **`.cppm`**. The reason is:
  - Visual C++ requires special treatment so that you cannot use `.cpp`.
  - It is the best self-explanatory file extension (far better than `.ixx` recommended by Visual C++ here currently).
  - clang requires it currently.
  - For gcc it is possible to use.
- For **internal partition files** (partition implementation files) use the file extension **`.cppp`**. The reason is:
  - Visual C++ requires special treatment so that you cannot use `.cpp`.
  - For gcc it is possible to use.
  - clang does not support these files currently (September 2021), at all.
- For **module implementation files** (but not ***not*** partition implementation files) use the usual file extension, which usually **`.cpp`**. The reason is:
  - No special artifacts are generated.
  - No special command-line options are required

As a consequence, you have to (pre-) compile module files as follows (only the special cases are mentioned):

- **Visual C++:**

Visual C++ requires specific command-line extensions and prefers a different file extension `.ixx` than the one I propose. For this reason:

- Compile an interface file `file.cppm` as follows:

  ```
  cl /c /interface /Tp file.cppm
  ```

  The option `/Tp` specifies that the following file is C++ code. The option `/interface` specifies that ***all*** following files are interface files (having both interface and non-interface files on one command-line might not work).

  If you use the file extension **`.ixx`** instead the compiler recognizes the file automatically as interface file.

- Compile an internal partition file `file.cppp` as follows:

  ```
  cl /c /internalPartition /Tp file.cppp
  ```

  The option `/Tp` specifies that the following file is C++ code. The option `/internalPartition` specifies that ***all*** following files are internal partitions.

- **gcc/g++:**

  GCC does not require any special file extension or command-line option at all. So, by using special file extensions, we have to specify that the files contain C++ code using the command-line option `-xc++`:

  - Compile an interface file `file.cppm` as follows:

    ```
    g++ -c -xc++ file.cppm
    ```

  - Compile an internal partition file `file.cppp` as follows:

    ```
    g++ -c -xc++ file.cppp
    ```

- **Clang:**

  Clang currently only supports interface files. As the proposed extension **`.cppm`** for them is required anyway, using it should just work.

  However, you cannot use internal partition files.

******* This chapter/section is at work *******

19.3 Exporting

******* This chapter/section is at work *******

19.4 Importing

******* This chapter/section is at work *******

19.5 Private Module Fragments

If you declare a module in a primary module interface unit, you might sometimes need a *private module fragment*. It allows programmers to have declaration and definitions inside the module interface unit that are neither visible nor reachable by any other module or translation unit. One way to use it is to disable exporting the definition of a class or function although its declaration is exported.

Consider, for example, the following module interface unit:

```
export module MyMod;

export class C;                    // class C is exported
export void print(const C& c);     // print() is exported

class C {                          // provides details of the exported class
 private:
  int value;
 public:
  void print() const;
};

void print(const C& c) {           // provides details of the exported function
  c.print();
}
```

Here we first forward declare class `C` and function `print()` with `export`:

```
export module MyMod;

export class C;                    // class C is exported
export void print(const C& c);     // print() is exported
```

`export` may be specified only once at the point where a name is introduced in its namespace. The details are also exported later. Therefore, any translation unit can import this module and use objects of type `C`:

```
import MyMod;
...

C c;          // OK, definition of class C was exported
print(c);     // OK (compiler can replace the function call by its body)
```

However, if you want to encapsulate the definition inside the module so that importing code only sees the declaration and you still want to have the definition in the module interface unit, you have to place the definitions in the *private module fragment*:

```
export module MyMod;

export class C;                    // declaration is exported
export void print(const C& c);     // declaration is exported

module :private;   // stuff from here is not (implicitly) exported
```

```
class C {                           // complete class not exported
 private:
  int value;
 public:
  void print() const;
};

void print(const C& c) {            // definition not exported
  c.print();
}
```

The *private module fragment* is declared with

```
module :private;   // stuff from here is not (implicitly) exported
```

It can only occur in primary module interface units and occur only once. With its declaration, the rest of the file is no longer (implicitly) exported. Using `export` afterwards to export anything is an error.

By moving the definition into the *private module fragment*, importing code code can no longer use any definitions there. It can only use the forward declaration of class `C` (class `C` is an *incomplete type*) and `print()`.

For example, you cannot create objects of type `C`:

```
import MyMod;
...

C c;          // ERROR (C only declared, not defined)
print(c);     // OK (compiler can replace the function call by its body)
```

However, the declarations are good enough to use references and pointers of type `C`:

```
import MyMod;
...

void foo(const C& c) {  // OK
  print(c);             // OK
}
```

19.6 Dealing with Header Files

******* This chapter/section is at work *******

19.7 Status of Modules in Practice

******* This chapter/section is at work *******

Providing files that can be both header and module are not allowed (would need preprocessor statements before `#define` and `#ifdef`).

19.8 Afternotes

Glossary

This glossary provides a short definition of the most important non-trivial technical terms used in this book.

A

argument-dependent lookup (ADL)
A feature that allows the programmer to skip namespace qualification of a function when one of the parameters is in its namespace. That is, a function will be looked up also in all namespaces of the arguments passed.

C

class template argument deduction (CTAD)
The process that implicitly determines template arguments from the context in which class templates are used. It was introduced with C++17 and allows you to skip specifying template arguments of an object when the template parameters can be deduced from the constructor.

F

forwarding reference
The term the C++ standard uses for *universal references*.

full specialization
An alternative definition for a (*primary*) template that no longer depends on any template parameter.

function object (functor)
An object that can be used as a function. For this, in the class `operator()` is defined. All lambdas are function objects.

G

glvalue
A *value category* of expressions that produce a location for a stored value (generalized localizable value). A glvalue can be an *lvalue* or an *xvalue*.

I

incomplete type
A class, struct, or unscoped enumeration type that is declared but not defined, an array of unknown size, `void` (optionally with `const` and/or `volatile`), or an array of an incomplete element type.

L

lvalue
A *value category* of expressions that that produce a location for a stored value that is not assumed to be movable (i.e., *glvalues* that are no *xvalues*). Examples are:
– Named objects (variables)
– String literals
– Returned lvalue references
– Functions and all references to functions
– Data members (unless values member of *rvalues*)

P

partial specialization
An alternative definition for a (*primary*) template that still depends on one or more template parameters.

predicate
A callable (function, function object, or lambda) that checks whether a certain criterion applies to one or more arguments. It returns a Boolean value, is read-only, and *stateless*.

prvalue
A *value category* of expressions that perform initializations. Prvalues can be assumed to designate pure mathematical value such as `1` or `true` and temporaries objects without names. Examples are:
– All literals except string literals (`42`, `true`, `nullptr`, etc.)
– Returned values (values not returned by reference)
– Results of constructor calls
– Lambdas
– `this`
What was called a *rvalue* before C++11 is called a *prvalue* in C++11.

R

resource acquisition is initialization (RAII)
a programming pattern to delegate clean-ups necessary for the end of using a resource to a destructor, so that they happen automatically when the object representing the resource leaves its scope or ends its lifetime.

regular type
A type that matches built-in type (such as `int`) semantics. Based on http://stepanovpapers.com/DeSt98.pdf it offers basic operations
– Default constructor (`T x;`)
– Copying (`T y = x;`) and in C++ moving
– Assignment (`x = y;`) and in C++ move assignment
– Equality and inequality (`x == y` and `x != y`)
– Ordering (`x < y` etc.)
with the "usual" naive rules:
– If an object is a copy of the other the objects are equal.
– A copy of an object is equal to an object created with the default constructor and where the source value was assigned to.
– Objects have value semantics. If two objects are equal and we modify one of them they are no longer equal.
If only the comparisons operators are missing the type is called *semiregular*.

rvalue
A *value category* of expressions that are not *lvalues*. An rvalue can be a *prvalue* (such as a temporary object without name) or an *xvalue* (e.g., an *lvalue* marked with `std::move()`). What was called an *rvalue* before C++11 is called a *prvalue* since C++11.

S

semiregular type
A type that is *regular* but does not provide comparison operators:
– If an object is a copy of the other the objects have the same value
– A copy of an object has the same value as an object created with the default constructor and where the source value was assigned to
– Objects have value semantics. If two objects have the same value and we modify one of them they no longer have the same value.

substitution failure is not an error (SFINAE)
A mechanism that silently discards templates instead of triggering compilation errors when arguments to template parameters make their declarations ill-formed. You can use the mechanism to disable templates for certain arguments by forcing ill-formed declarations then.

small/short string optimization (SSO)
An approach to save allocating memory for short strings by always reserving memory for a certain number of characters. A typical value in standard library implementations is to always reserve 16 or 24 bytes of memory so that the string can have 15 or 23 characters (plus 1 byte for the null terminator) without allocating memory. This makes all string objects larger but usually saves a lot of running time because in practice, strings are often shorter than 16 or 24 characters and allocating memory on the heap is quite an expensive operation.

stateless
A function or operations is *stateless* if it does not change its state due to a call. That is it should not change its behavior over time and always yield the same result for the same arguments.

standard template library (STL)
The STL is the part of the C++ standard library that deals with containers (data structures), algorithms, and as glue the iterators. This approach was adopted for the first C++ standard with the goal that programmers can benefit from various standard data structures and algorithms without the need to implement them. As generic code you still get compile-time error messages when you try to combine things that are not supported.

U

universal reference
A reference that can universally refer to any object while not making it `const`. It can also extend the lifetime of return values. It is declared as rvalue reference of a template parameter (`T&&`) or with `auto&&`.
Universal references are useful to perfectly forward arguments with `std::forward<>()` (for that reason the C++ standard names them *forwarding references*), but also to refer to passed arguments or extend the lifetime of temporary return values, when making them `const` is a problem (which is the case for some views).

V

value category
A classification of expressions. The traditional value categories *lvalues* and *rvalues* were inherited from the programming language C. C++11 introduced alternative categories: *glvalues* (generalized lvalues), whose evaluation identifies stored objects, and *prvalues* (pure rvalues), whose evaluation initialize objects. Additional categories subdivide *glvalues* into *lvalues* (localizable values) and *xvalues* (eXpiring values). In addition, *rvalues* (readable values) serve as a composed category for both *xvalues* and *prvalues* (before C++11, *rvalues* were what *prvalues* are since C++11).

variable template
A templified variable. It allows us to define variables or static members by substituting the template parameters with specific types or values.

variadic template
A template with a template parameter that represents an arbitrary number of types or values.

X

xvalue
A *value category* of expressions that that produce a location for a stored object that can be assumed to be no longer needed. Examples are:
– Values marked with `std::move()`
– Returned rvalue references
– Casts of objects (not of functions) to an rvalue reference,
– Value data members of rvalues

This page is intentionally left blank

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z