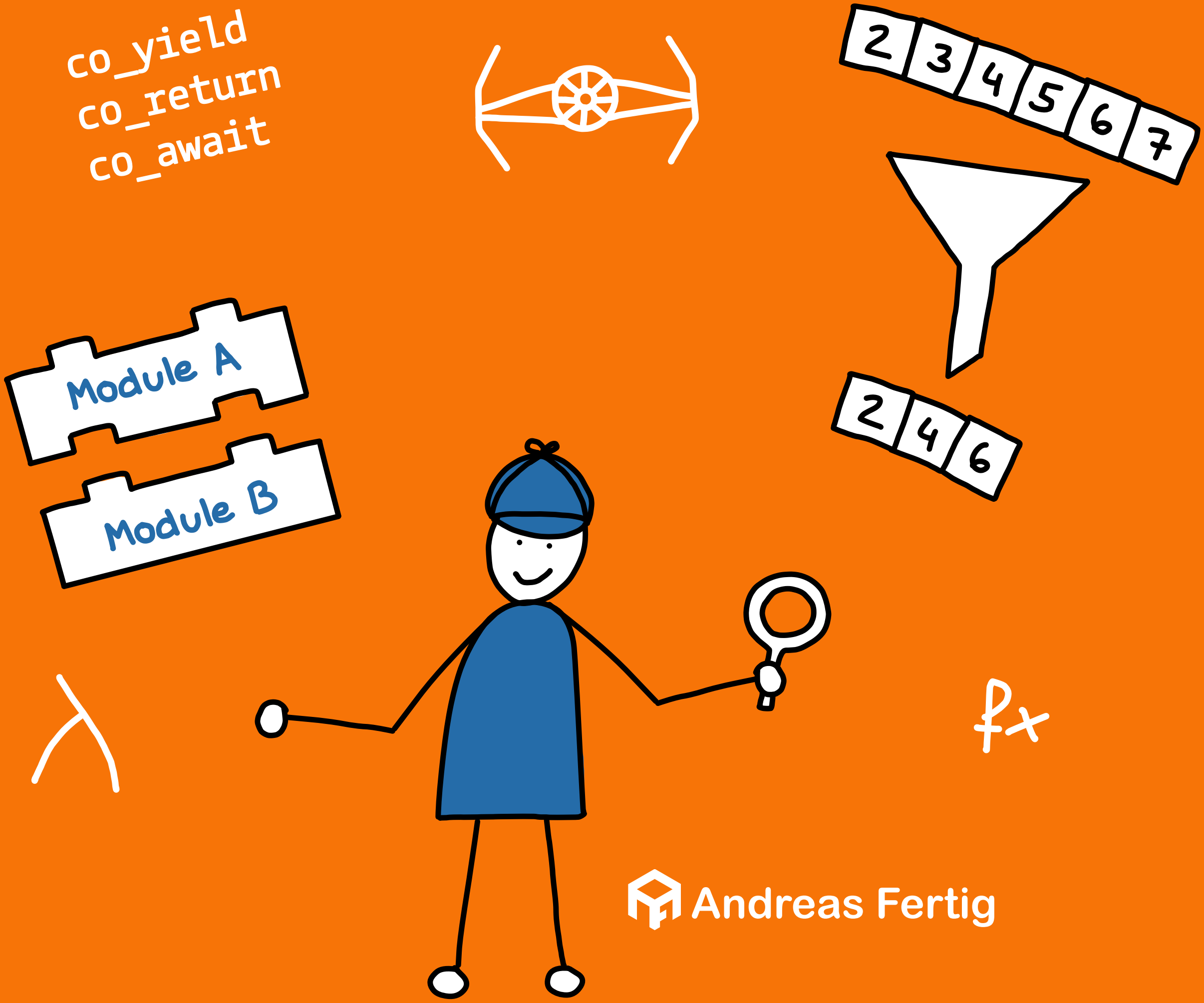
Programming with C++20

Concepts, Coroutines, Ranges, and more



Andreas Fertig

Andreas Fertig

Programming with C++20

Concepts, Coroutines, Ranges, and more

1. Edition

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

Planning and text:
Andreas Fertig

Cover art and illustrations:
Franziska Panter
https://franziskapanter.com

Published by:
Fertig Publications
https://andreasfertig.info

ISBN: 978-3-949323-01-0

This book is available as ebook at https://leanpub.com/programming-with-cpp20

Using Code Examples

This book exists to assist you during your daily job life or hobbies. All examples in this book are released under the MIT license.

The main reason for the MIT license was to avoid uncertainty. It is a well established open source license and comes without many restrictions. That should make it easy to use it even in closed-source projects. In case you need a dedicated license or have some questions around it, feel free to contact me.

Code download

The source code for this book's examples is available at
`https://github.com/andreasfertig/programming-with-cpp20`.

Used Compilers

For those of you who try to test the code and like to know the compiler and revision I used here you go:

- g++ (GCC) 10.1.0
- clang version 10.0.0 (https://github.com/llvm/llvm-project.git d32170dbd5b0d54436537b6b75beaf44324e0c28)

About the Author

Andreas Fertig is an independent trainer and consultant for C++ specializing in embedded systems. Since his computer science studies in Karlsruhe, he has dealt with embedded systems and the associated requirements and peculiarities. He worked for about 10 years for Philips Medizin Systeme GmbH as a C++ software developer and architect with a focus on embedded systems.

Andreas is involved in the C++ standardization committee. He is a regular speaker at conferences internationally. Textbooks and articles by Andreas are available in German and English.

He has a passion for teaching people how C++ works, which is why he created C++ Insights (cppinsights.io).

You can find him online at andreasfertig.info and his blog at andreasfertig.blog

About the Book

This book teaches programmers with C++ experience the new features of C++20 and how to apply them. It does so by assuming C++11 knowledge. Elements of the Standards between C++11 and C++20 will be briefly introduced, if necessary. However, the focus is on teaching the elements of C++20.

`main` vs. `Main`

You will notice that in some examples, you see a function `void Main()`. The reason for it is that I have unit-tests in the background. Now the unit-test framework also wants to provide the usual `main` to set itself up. To have the same code compile with and without unit-tests, I move in some cases `main` to `Main` and let the preprocessor take care of whether to include the unit-test frameworks `main` or not. The code published on GitHub does use only `main`. All unit-test code is stripped away.

Feedback

This book is published on Leanpub as a digital version. The printed version will follow. It helps if you indicate the book type you're referring too.

In any case, I appreciate your feedback. Whether it is a typo, a grammatical issue, naming of variables and functions, or logical things, please report it to me. You can send it to books@andreasfertig.info.

Thank you

I like to say thank you to everyone who reviewed drafts for this book and gave me valuable feedback. Thank you! All this feedback helped to improve the book. Here is a list of people who provided feedback: Vladimir Krivopalov, Hristiyan Nevelinov, John Plaice, Peter Sommerlad, Salim Pamukcu, and others.

A special thanks goes to Fran Buontempo. She provided extensive feedback from the beginning and was never tired to point out some grammar issues along with poking to the base of my code examples, helping me to make them better.

Revision History

2021-01-30: First release (Leanpub)

2021-04-22: Various spelling and grammar updates due to feedback. Added chapter 9. (Leanpub)

Table of Contents

Chapter 1

Concepts: Predicates for strongly typed generic code

Templates have been with C++ since the early beginnings. Recent Standard updates added new facilities to them like variadic templates. Templates enable Generic Programming (GP), the idea of abstracting concrete algorithms to get generic algorithms. They then can be combined and used with different types to produce a wide variety of software without the need to provide a dedicated algorithm for each type. GP or Template Meta-Programming (TMP) are powerful tools, for example, the Standard Template Library (STL) heavily uses them.

However, template code was always a bit, well, clumsy. When we write a generic function we only need write it once; then it can be applied to various different types. Sadly, when using the template with an unsupported type, finding any error required a good understanding of the compilers errors message. All we needed to face such an compiler error message was a missing `operator<` that wasn't defined for the type. The issue was, that we had not way of specifying the requirements to prevent the missuse and at the same time give a clear error message.

Concepts vs concepts vs `concepts`

This chapter comes with an additional challenge. The language feature we will discuss in this chapter is called Concepts. We can also define a concept ourself and there is a `concept` keyword. When I refer to the feature itself it is spelled with a capital C, Concepts. The lower case version is used when I refer to a single concept definition and the code-font version `concept` refers to the keyword.

1.1 Programming before Concepts

Let's consider a simple generic `add` function. This function should be able to add an arbitrary number of values passed to it and returns the result. Much like this:

```cpp
const int a = add(2,3,4,5);
const int b = add(2,3);
```

To do that `add` needs to be a variadic template. With just these requirements an implementation using C++17s fold expressions looks like this:

```cpp
template<typename... Args>
auto add(Args&&... args)
{
  return (... + args);
}
```

Listing 1.1

1.1 C++17: Fold expressions

Before C++17 whenever we had a parameter pack, we need to recursively call the function which received the pack and split up the first parameter. That way we could traverse a parameter pack. C++17 allows us to apply an operation to all elements in the pack. For example, `int result = (... + args);` applies the $+$ operation to all elements in the pack. If we assume that the pack consists of three objects, this example would produce `int result = arg0 + arg1 + arg2;`. This is much shorter to write than the recursive version. That one needs to be terminated at some point. With fold expressions, this is done automatically by the compiler. We can use other operations instead of $+$, like $-$, $/$, $*$ and so on.
The important thing to realize about fold expressions is, that it is only a fold expression if the pack expansion has parentheses around it and an operator like $+$.

So far this is a trivial exercise, but what if the function should ensure that all passed values are of the same type? Such that a user cannot mix `int` and `float`, for example. One way to go is with a `static_assert` inside of `add` that checks all types, but against what? The answer is against the first type. For an implementation we can use C++17 type-trait `std::conjunction` together with `std::is_same` and we are done:

```cpp
template<typename T, typename... Ts>
inline constexpr bool are_same_v =
  std::conjunction_v<std::is_same<T, Ts>...>;

template<typename... Args>
auto add(Args&&... args)
{
  static_assert(are_same_v<Args...>);

  return (... + args);
}
```

Listing 1.2

What `std::conjunction` does is to perform a logical conjunction of all the `std::is_same` values from the pack we pass to `are_same_v`. The result is equivaltent to a boolean expression in the form of arg_1 && arg_2 && arg_3 ....

1.1.1 The `enable_if` requirement

There is one issue, well at least one. We just put a requirement to `add` but it is buried inside the implementation of `add`. Yes, we get a compile-time error, when passing values of different types, just that is probably late. Had we known about this upfront, we would not have used that function, for example. The other approach is to state the requirement somewhere in a comment for the function. Comments are great, but they are not checked by the compiler and can be outdated or just wishes of what the implementation should do. Having self-documenting code which is checked by the compiler is even better. Leave text-comments for explaining things. Back to the issue, one way to get the requirement out of the body of the implementation is to do the checking with an `enable_if`. In cases like this, I usually put the `enable_if` on the return type. For `enable_if` to work, we need a single type that `enable_if` returns.

The easiest way to get this type is by adding an additional template parameter `T` as the first one and use it as the first function parameter as well.

```
template<typename T, typename... Args>
std::enable_if_t<are_same_v<T, Args...>, T> add(T&& arg, Args↩
    &&... args)
{
  return (arg + ... + args);
}
```

Listing 1.3

All right, this works and does the job. We can also argue that we put the requirement into the function's signature. Do you think this form is readable? If your answer is yes, you probably have written and read a lot of code like this and are used to it. In this case take a step back. If your answer is no, I can relate. I can read that code as many of you can, but teaching such code to people and stating that it is a great code is different. I would not call such code good code, but it was what we had in our toolbox for a long time.

1.1.2 Long error messages from template errors

There are alternatives, like the `static_assert` and there are other variations, but all remain a hack at the end of the day. The error message we get from our attempts is still hard to interpret. Say we change the `add` to be called with two int's and a double like this:

```
printf("%d\n", add(2, 3, 4.0));
```

Listing 1.4

The error message a compiler gives us is this:

```
error: no matching function for call to 'add'
  printf("%d\n", add(2, 3, 4.0));
                 ^~~
note: candidate template ignored: requirement 'are_same_v<int, int, ↩
    double>' was not satisfied [with T = int, Args = <int, double>]
std::enable_if_t<are_same_v<T, Args...>, T> add(T&& arg, Args&&... ↩
    args)
                                            ^
```

"No matching function for call to 'add'" isn't what at least I see as a good error message. When we are reading more of it and we are more familiar with templates and there usual error messages we can see, that the compiler tells us that it is unhappy with `Args = <int, double>`. The `enable_if` approach has its limits when it comes to precise error messages.

1.2 What are Concepts

What we are looking at is the concept of an `add` function, which requires that all values are of the same type. C++20 gives us the ability to express Concepts by using the new keyword `concept`. We can specify the requirements of a function or class and any other template with a concept. Aside from many powerful aspects of Concepts, they allow us to not only specify the requirements but also to name the concept. Naming is a hard but useful exercise. Naming a concept allows us to reuse this concept. We can see Concepts as compile-time predicates. All the concept checks are evaluated at compile-time. Hence, all the code we write to express a concept is compile-time only and does not add to the size of the resulting binary.

1.2.1 Thinking in concepts

Thinking in concepts instead of types is the C++20 era. We are currently talking about generic code, one example where it always became interesting was, when we have a generic function that was supposed to work with floating point numbers as parameters. Why? Because there are three different types of floating point numbers in C++: `float`, `double`, and `long double`. In the `cmath` header we find the `abs` function. We can find three functions doing the same thing, one for each floating point type.

```
float       abs(float arg);
double      abs(double arg);
long double abs(long double arg);
```

To be fair, these functions are from C. In C++ with templates there would probably be one function template combining all three. It's just, how would this function template look? Probably with a not so much readable `enable_if` which restricts the parameter to be a floating point type. That same restriction is then later required for `sqrt`. However, reusing an `enable_if` was not that common. Looking at this without the focus on `enable_if`, we can say that `abs` as well as `sqrt` have the requirement to take only floating point numbers. Wouldn't such a version be much more readable and meaningful, if we could write code like the following:

```
floating_point abs(floating_point arg);
```

Here one can clearly see that `abs` takes a `floating_point` and returns one. The difference is, that in this version we no longer think in concrete types, we think in Concepts.

1.2.2 Concepts and types

When we think about what a type means, we can say that a type specifies which set of operations we can apply to an object of that type. Some of them are explicitly visible to us, others are implicitly defined private or friend functions. One thing that comes up, which is often important as well, is the type of an object specifies its memory layout. We usually either think in values or in types. The latter one more often when doing TMP. There, like in the `add` or `abs` function examples above, we do not care about the values. We only care about types and their operations. For `add` to work the type passed to the function must provide a plus operation, yet the value behind the type is not important.

With Concepts there is a third view. All the above applies to a concept as well, except that a concept specifies nothing about the layout of an object.

In C++ a concept is a compile-time predicate which yields a boolean value. The STL comes with a new header `<concepts>` that provides 31 Concepts. Before defining your own concept, have a look at what the STL offers and prefer to use existing Concepts. For cases where you need a modified concept, we will see that we can use Concepts to build other Concepts.

1.3 The anatomy of a concept

We define a concept the following way:

```
template-head
concept NAME_OF_THE_CONCEPT = CONSTRAINT_EXPRESSION;
```

Concepts always have a template-head which follows the same rules as other templates. For example, variadic arguments are possible in a concept template-head. A concept is then introduced by the `concept` keyword followed by the name we give that concepts. A concept is then initialized with a constraint-expression which must yield a boolean value. Due to the compile-time nature of a concept it can be seen as `constexpr`.

Writing a concept looks a bit similar to writing a variable template.

```cpp
template<typename T, typename U>
constexpr bool IsSame = std::is_same_v<T, U>;
```

Listing 1.5

This defines a new variable `IsSame` which is of type `bool` and initialized with `true` if `T` and `U` are of the same type. Because `IsSame` is a variable template, we can have `IsSame` for multiple types. Here is the concept version:

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

Listing 1.6

We can clearly see the similarities. The difference is more or less only in the variable's type or the concept. While the variable template needs to specify its type, a concept is always of type `bool` and also always `constexpr`. Hence both are implicit only. You can see the new keyword `concept` as a substitute for `constexpr bool`.

1.2 C++14: Variable templates

Variable templates were introduced with C++14. They allow us to define a variable, which is a template. This feature allows us to have generic constants like π:

```cpp
template<typename T>
constexpr T pi(3.14);
```

Listing 1.7

One other use-case is to make TMP more readable. Whenever we have type-trait which has a value we want to access, before C++14 we needed to do this: `std::is_same<T, int>::value`. Admittingly, the `::value` part is not very appealing. Variable templates allow to store the value of `::value` in a variable:

```cpp
template<typename T, typename U>
constexpr bool is_same_v = std::is_same<T, U>::value;
```

Listing 1.8

With that the shorter and more readable version is `is_same_v<T, int>`. Whenever you see a `_v` together with a type-trait, you're looking at a variable template.

A concept which returns always `true` or `false` is probably not the most useful, so let's look a different example. Suppose we like to create a concept `ByteLikeType` which should ensure that a type the concept is called with is

- either of type `char`, `unsigned char` or `byte`, and
- a pointer

We can then use this concept to create generic wrappers around Portable Operating System Interface (POSIX) functions like read or write and ensure that only byte-like types are passed to it.

For that, we can use the type-trait `std::is_same_v`. We check for each of these types and their `const` version.

```
template<typename T>
concept ByteLikeType = Ⓐ Define a new concept ByteLikeType
  std::is_same_v<char*, T> || Ⓑ Test for char*
  std::is_same_v<unsigned char*,
               T> || Ⓒ Test for unsigned char*
  std::is_same_v<const char*, T> || Ⓓ Test for const char*
  std::is_same_v<const unsigned char*,
               T>; Ⓔ Test for const unsigned char*
```

Listing 1.9

This time we used traditional type-traits. As you can see, if they provide a boolean result we can use disjunction and even conjunction to create the final result. Negation is possible as well. Concepts are not limited to working with type-traits only. We can use concepts in concepts. Rather than using `std_is_same_v`, we can use the new concept `same_as` inside our `ByteLikeType` concept as follows:

```
template<typename T>
concept ByteLikeType =
  std::same_as<void*, T> || std::same_as<char*, T> ||
  std::same_as<unsigned char*, T> || std::same_as<const void*, T>↩
      ||
```

Listing 1.10

```
  std::same_as<const char*, T> ||
  std::same_as<const unsigned char*, T>;
```

Listing 1.10

Notice that because `same_as` is a concept there is no need for a `_v`. In contrast to type-traits a concept is not a `struct` and hence there is no value member in it.

This still looks a bit like what we could do with `enable_if` but is already more powerful as the error messages we get from this version are short and tell us that a requirement wasn't satisfied. Before we look at more ways to define requirements let's first have a brief look at existing concepts.

1.4 Existing Concepts

Before we dive deeper into how we can create Concepts ourselves I like to point out, that there is no need to invent all concepts yourself. The STL comes with 31 pre-defined common concepts. They are included in the `<concepts>` header. Most of them are concept definitions for existing type-traits, `same_as` uses `std::is_same_v` as we well see in this chapter. These pre-defined concepts consider subsumption rules, something we will discuss later in this chapter, and have the requires helpers to avoid different results due to parameter swapping. Table 1.1 lists the concepts defined in the Standard and available with the STL.

Now that we have seen what concepts the STL already provides and how we can build our own, more elaborated, concept using type-traits, it is time to look at the much bigger power Concepts gives us by introducing the ability to formulate requirements in the form of a requires-clause.

1.5 The requires-clause: The runway for Concepts

A concept always comes with a set of requirements. To formulate these requirements, a new keyword `requires` was added to C++20. We can use a requires-clause without making it a concept. A requires-clause can be seen as the foundation of a Concept. We can use multiple requires-clauses to build a Concept. A requires-clause produces a boolean value. In the case that all requirements in a requires-clause are met, the

Table 1.1: Existing concepts in C++20

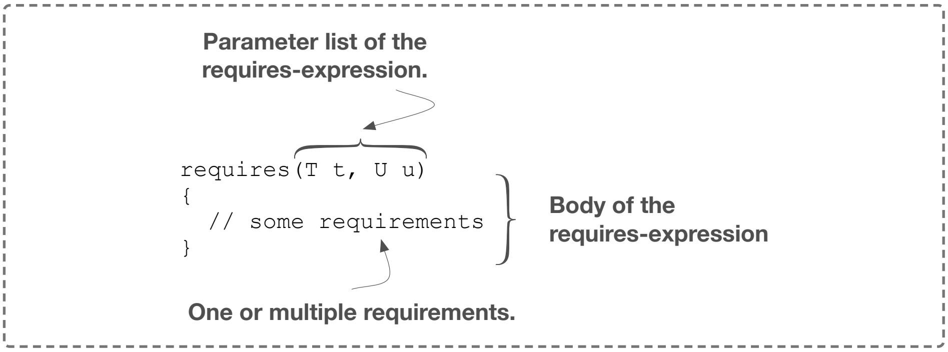
Arithmetic concepts	Type concepts	Construction concepts
integral	same_as	assignable_from
signed_integral	derived_from	swappable_with
unsigned_integral	convertible_to	destructible
floating_point	common_reference_with	constructible_from
	common_with	default_initializable
		move_constructible
		copy_constructible

Object concepts	Callable concepts	Comparison concepts
moveable	invocable	equality_comparable
copyable	regular_invocable	equality_comparable_with
semiregular	predicate	totally_ordered
regular	relation	strict_weak_order
	equivalence_relation	

boolean value of a requires-clause yields `true`, otherwise `false`. This can be seen a bit like `std::true_type`. In the simplest case we just prefix a type-trait which yields a value with `requires` and we have a requirement. Let's do it and apply `are_same_v` to the former add function:

```
template<typename... Args>
requires are_same_v<Args...> A Requires-clause using are_same_v
    to ensure all Args are of the same type.
 auto add(Args&&... args)
{
 return (... + args);
}
```

Listing 1.11


Parameter list of the requires-expression.
Body of the requires-expression
One or multiple requirements.

```
requires(T t, U u)
{
  // some requirements
}
```

Figure 1.1: Parts of a requires-expression.

This is nice, clean and readable. We did so far not define a concept; we put a visible requirement at the function definition. In terms of documentation, this is great. This form is also easy to teach, compare to alternatives like `enable_if`.

1.6 The requires-expression

It wouldn't be C++, if there wasn't an edge to `requires`. This edge is, that `requires` comes in two flavours, as a requires-clause as we have already seen but also as a requires-expression. Such a requires-expression stands out with a requirement body enclosed in curly braces as usual for a body. The requires-expression form can have an optional parameter list.

Despite the fact, that a requires-expression looks like code, it is still compile-time only. In fact, a requires-expression works much like an `enable_if`, using something close to substitution failure is not an error (SFINAE). The compiler checks whether the requirement listed inside the requires-expression is instantiable. A failed substitution of a requirement leads to a failed requirement-expression. In this case the requires-expression yields `false`. Depending on how and where the requires-expression is used, for example together with a concept, the bool value can be used or the result is used like SFINAE.

Inside a requires-expression there are four different kinds of possible requires-expressions:

- Simple requirement
- Type requirement
- Compound requirement
- Nested requirement

We will explore all of them and see their differences.

1.6.1 Simple requirement

Simple things first, the simple requirement. A simple requirement asserts the validity of an expression. Such an expression has to be ended by a semi-colon, very much like any other expression on C++. In the simplest case, we can check whether a given type provides operations like $+$, $-$, $*$, and $/$. Before Concepts and with that the requires-expression, the compiler would have failed if there were no such operations and we tried to use them. With `requires` we get a compile-time `true` or `false` back which we can use in a Concept or a constant evaluation like in a `constexpr if`. Using our former `add` function again, an object's type passed to the function needs to have a $+$ operation, let's check for that. The parameter pack provides a small obstacle. To assert that $+$ is possible we can execute the fold expression `(... + args)` as a simple requirement such that we can check, whether this is a valid expression.

```
requires(Args... args)
{
  (... + args); A Check that args provides +
}
```

Listing 1.12

As you can see, a simple requirement is a reason for having a requires-expression with parameters. We need variables to perform a simple requirement assertion. Should any of the types in `args` not support an add operation the simple requirement yields `false` and with that the whole requires-clause evaluates to `false`.

Is that all? No! `add` has more requirements. Before we already established that `add` should work only with values of the same type. So let's add that requirement as well. Then, a zero-sized parameter pack makes no sense. While we are on it, let's add that requirement.

1.6.2 Nested requirement

Whenever we need to assert that an expression evaluates to true, we need to request that this is a requirement. We do that by prefixing the expression in question with `requires`. This is then called a nested requirement. A nested requirement can nest all the other requirement types. You can see it as a way to start a new requires-clause inside a requires-expression.

```
requires(Args... args)
{
  (... + args); A Check that args provides +
  requires are_same_v<Args...>; B Check all types are the same
  requires sizeof...(Args) > 1; C Check that the pack contains
       at least two elements
}
```

Listing 1.13

One important remark, all parameters we define in the optional parameter list of the requires-expression shall appear as unevaluated operands only. That means that these optional local parameters can be used in a `sizeof` or `decltype` expression but we cannot pass them for example to a `constexpr` function because even in this case they would be evaluated.

1.6.3 Compound requirement

We don't have to stop there. Until now, we assumed that `add` is only called with numbers like `int` or `double`. The requires-clause does not state that requirement so far. So why not lift this requirement and say, all the type needs is a plus operation. That requirement we already have. For illustration, say we have a type `Rational` . A rational number has a plus operation, so has our `Rational` implementation an `operator+`. This class also comes with a constructor for `int` and stores this initial value. Then there are getters for the numerator and denominator.

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1)
  : mNumerator(numerator)
  , mDenominator(denominator)
```

Listing 1.14

```
  {}

  Rational& operator+(const Rational& rhs) noexcept;

  // other operators

  int Numerator() const { return mNumerator; }
  int Denominator() const { return mDenominator; }

private:
  int mNumerator;
  int mDenominator;
};
```

Listing 1.14

Looking at the implementation of `Rational` we can see that `operator+` is `noexcept`. This is probably a good thing and would be beneficial for `add` as well. With a compound requirement, we can check the properties of an expression. In the simple requirement before we checked, that `arg + arg` is a valid operation for a type, with a compound requirement we can enhance this check. A compound requirement allows us to assert that an expression is `noexcept`. Further, such a requirement allows to assert on the return-type of an expression. Related to `add` we can check whether the `operator+` is `noexcept` and that this operation returns an object of its own type. For example, `opertor+` in `Rational` could also return pointer. Both checks can be done with a single compound expression.

```
requires(Args... args)
{
  (... + args); Ⓐ Check that arg provides +
  requires are_same_v<Args...>; Ⓑ Check all types are the same
  requires sizeof...(Args) > 1; Ⓒ Check that the pack contains
      at least two elements

  Ⓓ Assert that ...+arg is noexcept and the return type is the same as
      the first argument type

  { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
```

Listing 1.15

```
}
```

Listing 1.15

A closer look at D raises the question if `same_as` works with just a single argument. It clearly needs something to compare `T` to. The answer is that the compiler injects the first parameter, which is the type of the result of the expression `arg + arg`. This internally leads to `same_as<decltype(arg + arg),T>`.

1.6.4 Type requirement

The last variant of requirement we can have in a requires-expression is the type requirement. This type of requirement asserts, that a certain type is valid. Based on the former example of `Rational` as objects passed to `add` and this could be, that `Rational` is a template which exposes its underlying type via a `using` alias with the name `type`.

```
template<typename T>
class Rational {
public:
  using type = T;

  // same as before
```

Listing 1.16

All that is left to do, is to add a type requirement to the existing requirements in the requires-expressions. A type requirement is always preceded by `typename`. This is how you can tell a type requirement apart from a nested requirement or a simple requirement which has absolutely nothing in front.

```
requires(T arg)
{
  arg + arg; A Check that arg provides +
  requires are_same_v<T, Args...>; B Check all types are the same
  requires sizeof...(Args) > 0; C Check that the pack contains
      at least one parameter
  {
    arg + arg
  }
```

Listing 1.17

```
  noexcept->same_as<T>; D Assert the properties of arg+arg
  typename T::type; E Assert a type requirement, T has "type"
}
```

Listing 1.17

If we now add a type requirement to the requirement-expression plain types like `int` or `double` will no longer work with `add` as they have no internal `type` defined. This is a reminder to think hard about what the requirements really are that a requires-expression or a concept models. Should we like to keep built-in types working with `add` but require class-types to have a `type` defined, we can formulate it, because we can use conjunction and disjunction to apply binary logical operations to constraints. In pseudo-code this can be describe as:

```
typename T::type || not std::is_class_v<T>
```

The question is now, which requirement kinds do we need to translate the above stated into a requires-expression to improve our existing implementation. Let's recap the requirements for the requirement. The type `T` is required to either not be a class type, or to have a `type` definition. Using type-traits again, we can check for class-types with `std::is_class_v`. As the requirement is not to be a class type, we need to negate the result. To make the requires-expression evaluate `std::is_class_v` as boolean, we need to prefix such a statement, as shown below in A, with `requires`. At this point we have a nested requirement. The second part of the or-condition is the same as before, we assert that in `T` a `type` exists. However, this time, we need to wrap that assertion in a compound requirement. We can apply binary logical operations only on expressions that return a boolean value. What we had before was a type requirement, which ends instantly, if the type is not present. So you can say that by putting a type requirement inside a compound requirement we can delay the termination decision. In out example, the result depends on whether `T` is a class or not.

```
requires(T arg)
{
  arg + arg;
  requires are_same_v<T, Args...>;
  requires sizeof...(Args) > 0;
```

Listing 1.18

```
  {
    arg + arg
  }
  noexcept->same_as<T>;
  requires not std::is_class_v<T> (A) Nested requirement to check for
      class types
    || (B) Logical or to the former expression
    requires
  {
    typename T::type;
  }; (C) Require a type only if T is a class
}
```

Listing 1.18

This is now the final requires-expression.

1.6.5 Complete constrained version of add

The final version of add with all the constraints we established looks like this

```
template<typename T, typename... Args>
requires requires(T arg)
{
  arg + arg;
  requires are_same_v<T, Args...>;
  requires sizeof...(Args) > 0;
  {
    arg + arg
  }
  noexcept->same_as<T>;
  requires not std::is_class_v<T> || requires { typename T::type;↩
       };
}
auto add(T&& arg, Args&&... args)
{
  return (arg + ... + args);
}
```

Listing 1.19

Here we can see, that if we use our requires-expression after the template-head we have to introduce it with another `requires`. This is due to fact that the first `requires` starts a requires-clause and the second `requires` starts a requires-expression. We go more in detail about this in Section 1.7.

We can use `add` either with a built-in like an `int` or with the rational type we created before, which meets the requirement that a `::type` exists in the class. Using the `Rational` class from before calling `add` with three `Rational` objects looks like this:

```cpp
const auto sum = add(3, 4, 5);

using RationalInt = Rational<int>;
const auto rationalSum =
  add(RationalInt{3, 4}, RationalInt{4, 4}, RationalInt{5, 4});
```

Listing 1.20

As you can see, from a users perspective of the `add` function nothing changed. Table 1.2 captures all four kinds of requires-expressions in a compact manner.

1.7 Adding Concepts to make requirements reusable

When looking at the definition of the constraints we created so far, one can say that is unreadable and looks like expert only. It is the same as with every code, there is a chance, that after a period of time even we as author do no longer easily understand what a code fragment does. Meaningful function and variable names and short functions which do a single task are usually a way to improve such code and make it speak even to colleagues who have not written the code in the first place. The requirements we've created so far are the same. They work and they are understandable on their own, but looking at the requirements of `add` as a user and being able to quickly tell what the requirements are is likely still a tough call.

What we have created for `add` by now is called an ad hoc constraint. We make these constraints up as we go and the whole requires-expression is not reusable. The constrains are attached to the single `add` function. This is why you should think about using ad hoc constraints. They are potentially good for short requirements, which are not used anywhere else. For what we've create so far, the ad hoc constraint is

Table 1.2: The four different kinds of requires-expressions

Code	Requires-expression kind	Description
`a + b;`	Simple requirement	Asserts that the operation $a+b$ is possible.
`requires are_same_v<Args...>;`	Nested requirement	Asserts that all types of the pack `Args` are of the same type.
`{ a + b } noexcept;`	Compound requirement	Asserts that the plus operation is `noexcept`.
`{ a + b } -> same_as<U>;`	Compound requirement	Asserts that the return type of the plus operation as the same as U.
`{ a + b } noexcept -> same_as<U>;`	Compound requirement	Combination of the former two compound requirements. Asserts that the return type of the plus operation as the same as U and the operation is `noexcept`.
`typename T::some_type;`	Type requirement	Asserts that a type “`some_type`“ exists in T.

likely the wrong thing. We can fix that. Instead of burying all the requirements, unnamed, in an ad hoc constraint we can split them and create a concept for each of them. This has two advantages, first we can give that concept a meaningful name, and second a named concept is, like a variable or function, reusable.

Ad hoc constraints

The form `requires requires`, where a requires-clause is followed by a requires-expression used on template arguments, is called an ad hoc constraint. Such a constraint can be used on template parameters, for example. This is can be a sign of a code smell. Before writing it, consider whether this is the right thing or a reusable concept definition is what is really needed.

```cpp
template<typename T, typename... Args>
concept Addable = requires(T arg)
{
  arg + arg;
};

template<typename... Args>
concept PackHasElements = sizeof...(Args) > 0;

template<typename T, typename... Args>
concept AddReturnsSameType = requires(T arg)
{
  {
    arg + arg
  }
  noexcept->same_as<T>;
};

template<typename T>
concept ClassWithTypeOrNotAClass = not std::is_class_v<T> || requires
{
  typename T::type;
};
```

Listing 1.21

As good as this looks, at least to me, keep in mind that on the other hand not every simple requirement needs to be a concept. Reusing concepts easily is only possible if one knows all the existing concepts and what they are doing and remembers them at the right time. As a rough estimate, over 50 concepts in a code base are hard to remember and to get right. With that in mind, we can apply the newly created concepts to our constrained add by replacing each requirement with the corresponding concept. As a result, the requirements are much more readable; we created an abstraction for users of the function.

```cpp
template<typename T, typename... Args>
requires requires(T arg)
{
  requires Addable<T, Args...>;
  requires are_same_v<T, Args...>;
  requires PackHasElements<Args...>;
  requires AddReturnsSameType<T>;
  requires ClassWithTypeOrNotAClass<T>;
}
auto add(T&& arg, Args&&... args)
{
  return (arg + ... + args);
}
```

Listing 1.22

We are free to completely drop the requires-expression and just use a requires-clause which uses all the previously defined concepts. At this point, it is a matter of taste.

```cpp
template<typename T, typename... Args>
requires Addable<T, Args...> && are_same_v<T, Args...>
        && PackHasElements<Args...> && AddReturnsSameType<T>
        && ClassWithType<T>
auto add(T&& arg, Args&&... args)
// ...
```

Listing 1.23

We now have an add function with a requires-clause consisting of meaningful and reusable concepts. At this point we should start talking about something that is essential for good software development and concepts are part of software development; testing.

1.8 Testing requirements

Now that we learned about the different requirement kinds and have written our first requirement, how to we test that what we have written is valid? Sure, we can apply invalid value or types to an requirement or concept and watch the compiler yell error messages at us. The good thing is, compared to former template errors, these are rather small and readable. One might say a pleasure to read. In my experience expected failing compilations are still a bad test. You never know whether the error was the expected one. Plus, we have to write a dedicated program and compile this program for each test.

Why not use concepts to test the constraints of functions or objects? We can use a concept in a `static_assert` and check its boolean value during compile-time without triggering a compilation failure, if the concept is not fulfilled. Because `add` can have multiple arguments, we define a concept `Test` with a variadic number of template arguments. Inside `Test` in the requires-expression we call `add` using `std::declval` to create a number of objects equal to the numbers in the parameter pack of the concept. These objects are directly passed to `add`. That way we can simulate zero to whatever number of arguments with which `add` is called.

```
template<typename... Args>
concept Test = requires Ⓐ Define a variadic concept as helper
{
  add(std::declval<Args>()...); Ⓑ Use declval to call add
      with the given types
};
```

Listing 1.24

Next we need a stub, as more or less always when creating unit tests. I'm a simple man and I like to write code only once. Plus, we are talking about GP here, so let's create a stub which is a class template. This takes four non-type template parameters (NTTPs):

- `NOEXCEPT`: Set `operator+ noexcept`.
- `hasOperatorPlus`: Enable `operator+` in the stub.
- `hasType`: The stub object has a `type`.

- `validReturnType`: The return type of the stub is of its type.

```
A Helper to either have a type in the class
namespace details {
  struct WithType {
    using type = int;
  };

  struct WithoutType {};
} // namespace details

B Class template stub to create the different needed properties
template<bool NOEXCEPT,
         bool hasOperatorPlus,
         bool hasType,
         bool validReturnType>
class ObjectStub
: public std::conditional_t<
    hasType,
    details::WithType,
    details::WithoutType> C Based on hasType
         select a
         base class
{
public:
  ObjectStub() = default;

  D Operator plus with controlled noexcept can be enabled
  ObjectStub& operator+(const ObjectStub& rhs) noexcept(NOEXCEPT)↩
         requires(
    hasOperatorPlus&& validReturnType)
  {
    return *this;
  }

  E Operator plus with invalid return type
```

Listing 1.25

```
  int operator+(const ObjectStub& rhs) noexcept(NOEXCEPT) ↩
      requires(
    hasOperatorPlus && not validReturnType)
  {
    return 3;
  }
};

Ⓕ Create the different stubs from the class template
using NoAdd = ObjectStub<true, false, true, true>;
using ValidClass = ObjectStub<true, true, true, true>;
using NotNoexcept = ObjectStub<false, true, true, true>;
using WithoutType = ObjectStub<true, true, false, true>;
using DifferentReturnType = ObjectStub<true, true, false, false↩
    >;
```

Listing 1.25

After that we call Test with the different stubs and use or negate the result based on the expected outcome. We wrap all these classes inside a static_assert.

```
Ⓐ Assert that type has operator+
static_assert(Test<int, int>);
static_assert(not Test<NoAdd, NoAdd>);

Ⓑ Assert, that no mixed types are allowed
static_assert(not Test<int, double>);

Ⓒ Assert that pack has at least one parameter
static_assert(not Test<int>);

Ⓓ Assert that operator+ is noexcept
static_assert(not Test<NotNoexcept, NotNoexcept>);

Ⓔ Assert that operator+ returns the same type
static_assert(not Test<DifferentReturnType, DifferentReturnType↩
    >);

Ⓕ Assert that a valid class works
```

Listing 1.26

```cpp
static_assert(Test<ValidClass, ValidClass>);

G Assert that there is a type present in a class type
static_assert(not Test<WithoutType, WithoutType>);
```

Listing 1.26

That's it. We have just written and used a concept to verify the correctness of the requirements of `add`. Of course, all at compile-time with no additional software or libraries. The only not so pleasant thing is that should one of the test fail we get only pointed to the failed `static_assert` and the call to `add` in the `Test` concept.

1.9 Using a Concept

Now that we have seen how to define and test requirements and concepts, it is time to use them. Let's think about the `ByteLikeType` concept from before, whose purpose is to limit the types. Suppose we intend to implement a network stack, like Transmission Control Protocol / Internet Protocol (TCP/IP). This hypothetical protocol has two different types of packets: DATA and ACK for acknowledgment. The latter one has a fixed size of 16 bytes. A data packet has a max size of 1.024 bytes. As other network stacks, we like to offer buffer pools of different sizes. Let's use two different sized buffer pools, one for ACK's and the other one for a maximum data packet, to keep the example simple. To keep the latency low on the network, we like to queue up to 10 ACK packets. Because of their small size, they would always add overhead if we were to send them one at a time. This is where two different `Send` functions come into play. The one for DATA sends the passed buffer out immediately, while the `Send` function for ACK uses a more complicated mechanism. Up to 10 packets are queued, and a timer ensures that the queue is flushed at a fixed interval. This function and the one for sending a data packet is named `Send`. To make this requirement and with that the following code work, we need generic code to implement the two `Send` functions. `ackData` calls the first `Send` function, while `payloadData` calls the second.

```cpp
std::array<char, 16> ackData{};
std::array<char, 100> payloadData{};
```

Listing 1.27

```

Send(ackData);
Send(payloadData);
```

Listing 1.27

From before we already have our `ByteLikeType` concept. We can use this concept to check whether an array like type has a underlying data type that is byte like. In addition we know that an ACK packet has 16 bytes. We can use this size to constrain the container further to have distinguished types for ACK and data. `std::array` comes with a typedef `value_type` which stores the type of the underlying data type. A `std::array` also has a `size` method which is `constexpr` and returns the number of elements that container provides. Both these conditions are true for the custom implementation. We use them to build the first part of our constraint.

```
Ⓐ Make value_type easily accessible
template<typename T>
using value_type_t = typename std::remove_reference_t<T>::↩
    value_type;

Ⓑ A constexpr function to obtain the size
template<typename T>
constexpr auto ExtractSize(T t = {}) Ⓒ Default parameter

{
  return t.size(); Ⓓ Call the size function of the container
}
```

Listing 1.28

We use a `constexpr` wrapper function to wrap the actual call to the containers `size` method. This function is a template to work with any container and has a default parameter of the container's type. We discuss the reason for that decision shortly.

Let's formulate the concept `SmallBuffer`. We use `ExtractSize` and instantiate it explicitly with the current concept type and compare the result to be less than or equal to 16. Of course, the concept also checks whether the `value_type` of the container fulfils the `ByteLikeType` concept. Our second concept `LargeBuffer` is fulfilled if the type does not satisfy `SmallBuffer` but still is a `ByteLikeType`.

```
template<typename T>
concept SmallBuffer =
  ExtractSize<std::remove_reference_t<T>>() <= 16 && Ⓐ Constrain
      the size
  ByteLikeType<value_type_t<T>>; Ⓑ Ensure the value_type is byte-like

template<typename T>
concept LargeBuffer = not SmallBuffer<T> && Ⓒ Not a SmallBuffer
                      ByteLikeType<value_type_t<T>>; Ⓓ But still a
                           ByteLikeType
```

Listing 1.29

The code above doesn't use `std::array`. The reason is to also allow `std::array` -like types. Suppose one customer uses `std::array` while another one uses its own implementation, which shares the same Application Programming Interface (API) as `std::array` and by that also provides `::value_type`. With two different types which share the same API we cannot simply write a concept that assumes `std::array`. The concept as shown above, is able to handle that case as well.

1.9.1 Using a `constexpr` function in a concept

As you can see, we can call `constexpr` functions as a nested requirement in a concept or requires-expression. The question is, why the additional `ExtractSize` wrapper instead of calling `.size()` directly. We learned the answer in Subsection 1.6.2, even if we define a parameter list in a requires-expression, we are not allowed to use these parameters in anything than an unevaluated context. Calling a `constexpr` functions and using the result is a evaluated context, and with that we will get a compiler error. The solution is to introduce a wrapper function with a default parameter, in this case `ExtractSize` and let it do the job. This trick only works, if the type `T` has a default constructor which also sets the values of an object to the one we need. In case of `std::array` which has a `.size()` method this is true. This method returns the value of the NTTP for the array's size. The reason why the requires-expression does not allow us to use the parameter is exactly this. By declaring a requires-expression parameter we only have a type, but not a workable fully instantiated object which is required in a lot of cases.

To summarize, we can call and use the result of `constexpr` functions in concepts or a nested requirement, but we cannot pass parameters from a requires-expression to a `constexpr` function.

1.9.2 Applying a concept with a requires-clause

At this point we have all the required pieces to implement the two generic `Send` functions. All we have to do is to create two function templates `Send` taking a single template parameter `typename T` and apply the concept as a requires-clause.

```
template<typename T>
requires SmallBuffer<T> void Send(T&& buffer)
{
  if(gAckQueue.size() > 10) {
    FlushAckQueue();

  } else {
    PushToAckQueue(buffer);
  }
}

template<typename T>
requires LargeBuffer<T> void Send(T&& buffer)
{
  send(buffer.data(), buffer.size());
}
```

Listing 1.30

Have I said that it is C++ we are talking about? Wouldn't you be a little disappointed if they would be only this one way to use a concept? Right. No reason to be disappointed. This is not the only way.

1.9.3 Applying a concept with a trailing requires-clause

C++20 allows a concept to be used as a type-constraint. We cannot only use a concept in a requires-clause, we can also apply a concept as a so called trailing requires-clause.

```cpp
template<typename T>
void Send(T&& buffer) requires SmallBuffer<T>
{
  // same as before
}

template<typename T>
void Send(T&& buffer) requires LargeBuffer<T>
{
  // same as before
}
```

Listing 1.31

In Subsection 1.12.2 we will see an example, where this trailing requires-clause is most helpful.

1.9.4 Applying a concept as a type-constraint

There is more. A probably much better syntax than the trailing requires-clause is using a concept as a type constraint because a concept increases the expressiveness. In case we use a concept, `typename` is replaced by the concept in question. In our case, `typename T` becomes `SmallBuffer T`.

```cpp
template<SmallBuffer T>
void Send(T&& buffer)
{
  // same as before
}

template<LargeBuffer T>
void Send(T&& buffer)
{
  // same as before
}
```

Listing 1.32

As you can see in the example above, the compiler injects the first parameter automatically to the concept. Hence, compared to the former version, we do not need

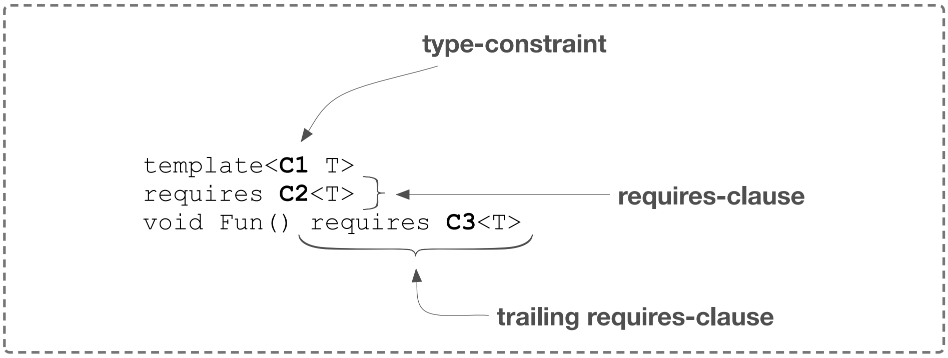


Figure 1.2: The different places where we can constrain a template or template argument.

to call the concept with template arguments. This is done implicitly by the compiler. The missing argument is filled by the compiler from left to right.

The type-constraint way is probably the most expressive, so far. The meaningless `typename` is replaced by a meaningful and constraining concept. Should we have a template with multiple template parameters then we can use some of them or all with type-constraints. Like in pre C++20 we can use already declared template parameters and pass them as arguments to a following concept. As you probably expect, a concept in a type-constraint is applicable for a parameter pack as well.

Figure 1.2 lists all the places in a template declaration where we can apply Concepts. Some of them need to be introduced by a requires-clause. Table 1.3 gives a guidance when to use which constraint form.

Table 1.3: When to use which constraint form

Type	When to use
type constraint	Use this when you already know that a template type parameter has a certain constraint. For example, not all types are allowed. In Figure 1.2 the type is limited to a floating-point type.
requires-clause	Use this when you need to add constraints for multiple template type or non-type template parameters.
trailing requires-clause	Use this for a method in a class template to constrain it based in the class template parameters.

1.10 Abbreviated function template with `auto` as generic parameter

We have seen the three places where we can use a concept in a function template to constrain a template parameter. C++20 opened the door for GP that looks more like regular programming. We can use a concept in the so called abbreviated function template syntax. This syntax comes without a template-head, making the function terse. Instead we declare what looks like a regular function, using the concept as a parameter type together with `auto`. The former definition of `Send` can be rewritten to this:

```cpp
void Send(SmallBuffer auto&& buffer);
void Send(LargeBuffer auto&& buffer);
```

Listing 1.33

1.10.1 What does such a construct do?

In the background, the compiler creates a function template for us. Each concept parameter becomes an individual template parameter to which the associated concept is applied as a constraint. This makes this syntax a shorthand for writing function templates. The abbreviated function template syntax makes function templates less scary and looks more like regular programming. The `auto` in such a functions signature is a sign, that we are looking at a template.

The abbreviated function template syntax can be a little more terse. The constraining concept here is optional. We can indeed declare a function, like `Send` above, with only `auto` parameters. This way, C++20 allows us to create a function template in a very comprehensive way.

1.10.2 Exemplary use case: Requiring a parameter type to be an invocable

The abbreviated syntax together with Concepts enables us to write less, but more precise code. Assume a system in which certain operations need to acquire a lock before performing an operation. An example is a file system operation with multiple processes try to write data onto the file system. As only one can write at a time because of control structures which have to be maintained such a write operation is often locked by a mutex or a spinlock. Thanks to C++11's lambdas we can write a `DoLocked` function template which takes a lambda as argument. In its body `DoLocked` first acquires a global mutex `globalOsMutex` using a `std::lock_guard` to release it after we leave the scope. Then in the next line the lambda itself is executed. Safely locked without each user needing to know which mutex to use. Plus the scope is limited and thanks to `std::lock_guard` the mutex is automatically released. Dead-locks should no longer happen.

```cpp
template<typename T>
void DoLocked(T&& f)
{
  std::lock_guard lck{globalOsMutex};

  f();
}
```

Listing 1.34

I used this pattern in different variations in a lot of places, but there is one thing which I always disliked. Do you guess what? Correct `typename T`. It is not obvious to a user that `DoLocked` requires some kind of callable, a lambda, a function object, or a function. Plus, for some reason, in this particular case the template-head added some boiler-plate code I did not like.

With a combination of the new C++20 features Concepts and abbreviated function templates, we can get rid of the entire template-head. As the parameter of this function we use the abbreviated syntax together with the concept `std::invocable`. The functions requirements are clearly visible now.

```cpp
void DoLocked(std::invocable auto&& f)
{
  std::lock_guard{globalOsMutex};

  f();
}
```

Listing 1.35

This is just one example, showing how abbreviated templates reduce the code to the necessary part. Thanks to Concepts, the type is limited as necessary. As often, I claim that especially for starters, this is much more understandable than the previous version. Thinking of a bigger picture with a more complex example, this syntax is useful for experts as well, clarity is key here.

Even more terse

In earlier proposals of the Concepts feature the abbreviated function template syntax was even more terse, without `auto` using just the concept as type. However, some people claim that new features have to be expressive and type intensive before users later complain about all the overhead they have to type. Maybe, in a future Standard, we can write the terse syntax without `auto`.

1.11 Concepts and constrained `auto` types

For some years now at least some of us struggle with placeholder variables `auto x = something;`. One argument I often hear against using `auto` instead of a concrete type is that this syntax is hard to know the variable's type without a proper Integrated Development Environment (IDE). I agree with that. Some people at this point tell me and others to use a proper IDE, problem solved. However, in my experience this is not entirely solving the problem. Think about code review, for example. They often take place in either a browser showing the differences or in another diff tool, tending not to provide context. All that said, there is also a good argument for using `auto` variables. They help us getting the type right, preventing implicit conversion or loss of precision. Herb Sutter showed years ago [1], that in a lot of cases we can put the type at the right and with that leave glues. Let's look at an example where `auto` makes our code correct.

1.11.1 Constrained `auto` variables

Suppose that we have a `std::vector v` which is filled with a couple of elements. At some point we need to know how many elements are in this vector. Getting this information is easy. We call `size()` on `v`. Often the resulting code looks like this.

```cpp
auto v = std::vector<int>{3, 4, 5};
const int size = v.size(); Ⓐ int is the wrong type
```

Listing 1.36

This example uses `int` to store the size. The code compiles and in a lot of cases works. Despite that, the code is incorrect. A `std::vector` does not return `int` in for its `size`-function. The `size`-function usually returns `size_t` but this is not guaranteed. Because of that, there is a `size_type` definition in the vector that tells us the correct type. Especially if your code runs on different targets using different compilers and standard libraries these little things matter. The correct version of the code is using the `size_type` instead of `int`. To access it, we need to spell out the vector, including its arguments, making the statement long and probably beginner unfriendly.

```cpp
auto v = std::vector<int>{3, 4, 5};
const std::vector<int>::size_type size = v.size(); // A Using the
     correct type
```

Listing 1.37

The alternative so far was to use `auto`, as shown below, and by that let the compiler deduce the type and keep the code to write and read short. Now the code is more readable in terms of what it does, but even harder to know what the type is. Most likely the deduced type is not a floating point type but only with a knowledge of the STL you can say that.

```cpp
auto v = std::vector<int>{3, 4, 5};
const auto size = v.size(); // A Let the compiler deduce the type
```

Listing 1.38

This is where Concepts can help us use the best of the two worlds. Because, think what we like to know in theses places usually. It is not necessarily the precise type, but the interface that type gives us. We expect and the following code probably requires the type to be some sort of integral type. Do you remember the abbreviated function template syntax? There we could prefix `auto` with a concept to constrain the parameter's type. We can do the exact same thing for `auto` variables in C++20.

```cpp
auto v = std::vector<int>{3, 4, 5};
const std::integral auto size = v.size(); // A Limit the types
     properties
```

Listing 1.39

This allows us to limit the type and its properties, without the need to specify an exact type.

1.11.2 Constrained `auto` return-type

We can do more than just constrain placeholder variables with Concepts. This syntax applies to return-types as well. Annotating an `auto` return-type has the same benefit as for `auto` variables or instead of `typename`, a user can see or lookup the interface definition. For example, the `add` function we constrained earlier has multiple constraints for the types we are passing to `add`, but we have to lookup the implementation of the function to see what requirements the return-type fulfills. With `add` we have a very short function, looking the return statements up is likely easy. Well, what do we actually see there? That all values are added together. What does this say about the return-type? Little.

One helpful constraint is `Addable` which tells us, with no need for looking into the implementation, that the type we get back can be used for another pass to `add`.

```cpp
template<typename T, typename... Args>
requires Addable<T, Args...>&& are_same_v<T, Args...>&&
  PackHasElements<Args...>&& AddReturnsSameType<T>&& ↩
      ClassWithType<T>

    Addable auto add(T&& arg, Args&&... args)
  // ...
```

Listing 1.40

Other constraints are possible as well. Whether the returned type is a pointer or a reference is often hard to tell without peaking into the implementation. A constrained return-type brings clarity without requiring a single dedicated type.

1.12 The power of Concepts: `requires` instead of `enable_if`

Concepts are more than just a replacement for SFINAE and a nicer syntax for `enable_if`. While these two elements allowed us for years to write good generic code Concepts enlarge the application areas. We can use Concepts in more places than `enable_if`.

1.12.1 Call method based in requires

One thing that gets pretty easy with Concepts is checking whether an object has a certain function. In combination with `constexpr if` this can be used to conditionally call this function, if it exists. An example is a function template which sends data via the network. Some objects may have a validation function to do a consistency check. Other objects, probably simpler types, do not need such a function and hence to not provide it. Before Concepts they would have probably provided a dummy implementation. In terms of efficiency of run-time and binary size this did not matter, thanks for state-of-the-art optimizers. However, for us developers it means to write and read this nonsense function. We have maintain these functions over years, just to do... nothing. There are solutions using C++11, `decltype` in the trailing-return type and the comma operator to test for the existence of a method. Thing is, writing this was a lot boiler-plate code and needed a deeper understanding of all these elements and their combination. With C++20 we can define a concept which has a requires-expression containing a simple requirement, with the name `SupportsValidation` and we are done.

```
template<typename T>
concept SupportsValidation = requires(T t)
{
  t.validate();
};
```

Listing 1.41

1.3 C++17: `constexpr if`

This kind of `if` statement is evaluated at compile-time. Only one of the branches remains, the other is discarded at compile-time dependent on the condition. The condition needs to be a compile-time constant, for example, from a type-trait or a `constexpr` function.

Instead of applying the concept to a type as a requires-clause, a trailing requires, or a type constraint, we can use `SupportsValidation` inside the function template together with `constexpr if` and call `validate` on `T` only, if the method exists.

```
template<typename T>
void Send(const T& data)
{
```

Listing 1.42

```cpp
  if constexpr(SupportsValidation<T>) { data.validate(); }

  // actual code sending the data
}

class ComplexType {
public:
  void validate() const;
};

class SimpleType {};
```

Listing 1.42

This allows us to provide types free of dummy functions and code. They are much cleaner and portable this way.

1.12.2 Conditional copy operations

When we create any wrapper, much like `std::optional` from C++17, that wrapper should behave like the objected wrapped in the `std::optional`. A `wrapper<T>` should behave like `T`. The Standard defines `std:optional` shall have a copy constructor, if `T` is copy constructible and that it should have a trivial destructor, if `T` is trivially destructible. This makes sense, if `T` is not copyable, how can a wrapper like `optional` copy its contents? Even if you find a way of doing it, the question is why should such a wrapper behave differently. Let's take only the first requirement and try to implement this using C++17, `optional` has a copy construct if and only if `T` is copy constructible. This task is fairly easy. There is even a type-trait `std::is_copy_constructible` and `std::is_default_destructible` to do the job.

We create a class template with a single template parameter `T` for the type the optional wraps. One way of storing the value is using a placement new in a aligned buffer. As this should be no complete implementation of `optional` let's ignore storing the value of `T`. An `optional` is default constructible regardless of the properties of its wrapped type, otherwise an optional would not be optional as it would always need a value. For the constrained copy constructor we need to apply a `enable_if` and in that check whether `T` is copy constructible and whether the parameter passed to the copy constructor is of type `optional`. This is an additional check we have to do, because of the templated version of this method. The resulting code is less in code than it took in text to explain.

```cpp
template<typename T>
class optional {
public:
  optional() = default;

  template<typename U,
           typename = std::enable_if_t<std::is_same_v<U, optional> ↩
               &&
                                      std::is_copy_constructible_v<T>>>
  optional(const U&);

private:
  storage_t<T> value;
};
```

Listing 1.43

After that we can try out our shiny, admittingly reduced, implementation. We create a struct called `NotCopyable`. In that struct we set the copy constructor as well as the copy assignment operator as deleted. So far we looked only at the copy constructor, but that is fine. The copy assignment operator behaves the same. With `NotCopyable` we can test our implementation. A quick test is to create to object of `optional<NotCopyable>` and try to copy construct the second passing the first as argument.

```cpp
Ⓐ A struct with delete copy operations
struct NotCopyable {
  NotCopyable(const NotCopyable&) = delete;
  NotCopyable& operator=(const NotCopyable&) = delete;
};

optional<NotCopyable> a{};
optional<NotCopyable> b = a; Ⓑ This should fail
```

Listing 1.44

That is great, the code compiles! Oh wait, that is not expected, isn't it? Did we make a mistake? Yes, one which is sadly easy to make. The standard says specifically what a copy constructor is and how it looks. A copy constructor is never a template. It follows exactly the syntax `T(const T&)`, that's it. The question is now what did we do, or more specifically what did we create? We created a conversion constructor. Looking at the code from a different angle, that supposed to be copy constructor takes and `U`. The compiler cannot know that instantiation of this constructor fails for every type except `optional<T>`. The correct way to implement this in C++17 and before was to derive `optional` from either a class with a deleted copy constructor and copy assignment operator or derived from one with both defaulted. We can use `std::conditional` to achieve this. That way, the copy operations of `optional` are deleted by the compiler, if a base class has them deleted, otherwise they are defaulted.

```
struct copyable {};

struct notCopyable {
  notCopyable(const notCopyable&) = delete;
  notCopyable operator=(const notCopyable&) = delete;
};

template<typename T>
class optional : public std::conditional_t<std::↩
    is_copy_constructible_v<T>,
                                         copyable,
                                         notCopyable> {
public:
  optional() = default;
};
```

Listing 1.45

Teach that to some person who is new to C++. We again have a lot of code for a simple task. How does this look in C++20? Much better. This is one case, where the trailing requires-clause shows its powers. In C++20 we can write just a copy constructor as we always do. No template required, the class itself is already a template. But we can apply the trailing requires to even non-templated methods. This helps us, because a trailing requires-clause doesn't make the copy constructor anything else. This method stays a copy constructor. It is even better, we can directly put our requirement, in form of the type-trait `std::is_copy_constructible_v<T>`, in the trailing requires-clause. Absolutely beautiful and so much more readable than any other approach before. As another plus, this requires zero additional code, which often looks unrelated, can be used by colleagues and needs maintenance.

```cpp
template<typename T>
class optional {
public:
  optional() = default;

  optional(const optional&) requires std::is_copy_constructible_v↩
      <T>;
};
```

Listing 1.46

1.4 C++17: `std::optional`

With `std::optional` we look at a type which has two states, it can contain a value or not. This is a big win in situations where we cannot reserve a dedicated value in a function return which expresses that the value could not be computed. Various access functions like `value_or` or `has_value` make its use clean and easy. Whenever you have a function which may not always be able to return a valid result, for example, the peripheral device is not available, `std::optional` is the choice.

1.12.3 Conditional destructor

The second requirement of an `optional` is the destructor. To be as efficient as possible, a destructor should only be present, if the type is non-trivially destructible, otherwise the destructor should be defaulted, keeping this `optional` instance trivially destructible. This is just another flavor of replicating the behavior of the wrapped type. In general, a conditional destructor is much like the conditional copy operations we discussed before. There is one important difference, the compiler does not allow us to create a templated destructor in the first place. This saves us from making a mistake like before, where we in fact failed to disable the copy constructor by making it a template. Anyway, the C++17 version looks a lot like the conditional copy operations with a lot of additional code.

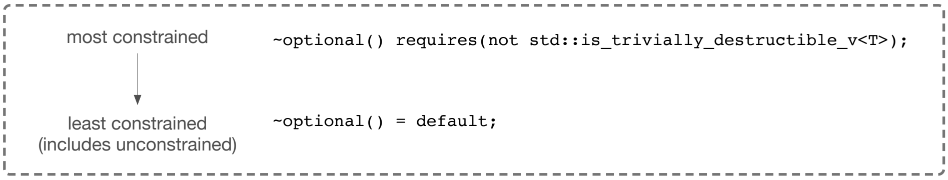
The good news is, we can put the trailing requires-clause on a destructor as well. As good as this news is, a conditional destructor as in the case of `optional` is a bit different. For the copy operations, it was enough to enable or disable them. The case is different, for the destructor. Here, if the wrapped type is trivially destructible, the destructor should be defaulted, but in case T is not trivially destructible we need a destructor which calls the destructor of T for the `optional` internal storage.

```cpp
template<typename T>
class optional {
public:
  optional() = default;
  // The real constructor is omitted here because it doesn't matter

  ~optional() requires(not std::is_trivially_destructible_v<T>)
  {
    if(has_value) { value.as()->~T(); }
  }

  ~optional() = default;

  optional(const optional&) requires std::is_copy_constructible_v<T> =
    default;

private:
  storage_t<T> value;
  bool has_value;
};
```

Listing 1.47

1.12.4 Conditional methods

What we have seen so far for the copy operations and the destructor works for all methods including special members like the default constructor as long as we have a class template.



```cpp
~optional() requires(not std::is_trivially_destructible_v<T>);

~optional() = default;
```

Figure 1.3: The compiler evaluates possible constrained overloaded functions from the most to the least constrained.

1.13 Concepts ordering

From time to time we run into cases where we need to know which function or method the compiler selects , and the compiler needs to have a way of reaching that decision. We already have seen a case in the last listing. In the final solution only one of the two destructors is constrained, the one which is active for a non-trivially destructible type. The defaulted destructor is unconstrained. So how does the compiler decide which one to pick? The rule is that the lookup in the overload-set for a match starts with the most constrained function and walks its way down to the least constrained one. A function with no constraint counts as least constrained as well. In the `optional` example, we could also have added a restriction to the defaulted destructor requiring it to be active only for trivially destructible types. My advice is to abstain from that. Don't do inverted requires-clauses, apply the least constraint principle.

In the current example of `optional` least and most constrained destructor is very easy to determine, for us and for the compiler. What if things get more complicated? For example, sometimes we have types that do not release their data in the destructor, as they are shared. One example are COM like objects in Microsoft Windows. A rough sketch of such a class could be this:

```cpp
struct COMLike {
  ~COMLike() {} Ⓐ Make it not default destructible
  void Release(); Ⓑ Release all data

  // Some data fields
};
```

Listing 1.48

Now, we assume that `optional` should act differently on a type with a `Release` method. In the destructor `optional` should always call `Release`, if the type has such a method. For types without a `Release` method the behavior as before applies, if the type in the `optional` is not trivially destructible the destructor of the type is called in the destructor of `optional`, otherwise the defaulted default destructor is provided.

The concept `HasRelease` to detect whether a type has a `Release` method is, with our current knowldge, quickly written:

```cpp
template<typename T>
concept HasRelease = requires(T t)
{
  t.Release();
};
```

Listing 1.49

Next, we add a new destructor to `optional` which in its requires-clause uses the new `HasRelease` concept plus the check that restricts the destructor to non trivially destructible types. In this destructor we call `Release`, if the optional contains a value when getting destructed.

```cpp
A Only if not trivially destructible
~optional() requires(not std::is_trivially_destructible_v<T>)
{
  if(has_value) { value.as()->~T(); }
}

B If not trivially destructible and has Release method
~optional() requires(not std::is_trivially_destructible_v<T> &&
                     HasRelease<T>)
{
  if(has_value) {
    value.as()->Release();
    value.as()->~T();
  }
}

~optional() = default;
```

Listing 1.50

Sadly, this doesn't compile. The compiler ends up finding two destructors Ⓐ and Ⓑ. The reason is that both requires-clauses yield to true and both destructors are constraint. Before we had the case of a constrained and an unconstrained one. One way to handle this is to write mutually exclusive requires-clauses. We add to destructor Ⓐ in the requires-clause a `not HasRelease<T>`. For this example this may be a way to go as the number of constraints is still low, for more complex examples this for sure blows up.

But there is this most, and least constrained thing, so let's look into this. For the constraint evaluation one constraint can subsume another constraint. Consider concepts *a* and *b*, and least and most constrained combinations of these. Using boolean algebra we can formulate the following:

$$\begin{aligned} a \wedge (a \vee b) &= a \vee b (1) \\ a \vee (a \wedge b) &= a (2) \end{aligned}$$

In (1) $a \vee b$ subsumes *a*, while in (2) *a* subsumes $a \wedge b$.

Constraint subsumption works only with concepts, this is a strong case to define named concepts instead of applying requires-clauses. To solve the situation with the desturctors, we need to define a second concept `NotDefaultDestructible` and replace the type-trait in the destructors requires-clauses with that. Now we have concepts in the requires-clause which enables the compiler to do constraint subsumption. If the compiler now approaches the two destructors (Ⓐ and Ⓑ below) it hits the following constraints

$$\begin{aligned} &(1) NotTriviallyDestructible < T > \\ &(2) NotTriviallyDestructible < T > \quad \wedge \quad HasRelease < T > \end{aligned}$$

Now the most constraint rule applies, because both destructor has only concepts as constraints. Now the compiler sees that both clauses contain `NotTriviallyDestructible` but (2) has `HasRelase` in addition. This results in (2) subsuming (1) and we have the final destructor for this type. The one left. The defaulted default destructor is unconstrained and with that the least constrained.

```
Ⓐ Only if not trivially destructible
~optional() requires NotTriviallyDestructible<T>;

Ⓑ If not trivially destructible and has Release method
~optional() requires NotTriviallyDestructible<T>&& HasRelease<T↩
    >;

~optional() = default;
```

Listing 1.51

There is more. This solution now is easily extendable by another destructor. By now `optional` handles classes with `Release` but which are trivial wrong. Such class results in calling the defaulted destructor, but then `Release` is never called. Think about the long requires-clause we would have to write with the type-trait version from the beginning. We would need to say that this fourth destructor is only for types which are trivially destructible and have a `Release` method. Certainly doable, but this approach starts blowing up. With the concept version we have so far we can simply add a new destructor which has the `HasRelease<T>` constraint and we are done.

```
Ⓐ Only if not trivially destructible
~optional() requires NotTriviallyDestructible<T>;

Ⓑ If not trivially destructible and has Release method
~optional() requires NotTriviallyDestructible<T>&& HasRelease<T↩
    >;

Ⓒ Trivial and has Release method
~optional() requires HasRelease<T>
{
  if(has_value) { value.as()->Release(); }
}

~optional() = default;
```

Listing 1.52

To summarize, prefer named concepts over ad hoc requirements or type-traits. This specifically is true for more complex concept based overloading of methods, if you like to profit from concept subsumption.

1.13.1 Subsumption rules details

With the rules so far, we are good to go. Nevertheless, from time to time we need to know more details. Subsumption rules of concepts can get very complex. Before we concluded to prefer named concepts over, well, everything else in a requires-clause. Now, let's dig into more details. Consider the `AreSame` concept we developed before for the variadic `add` example. This time we simplify the concept to `IsSame` comparing only two types.

For the purpose of illustration we create another concept `AlwaysTrue`, which has its value set to true. The only purpose of this concept is to have a second, different, concept to `IsSame`. We re-use the `add` function from before and limit it to two template arguments. We use this function and create two methods, both called `add` with identical arguments and template-head, only one is more constrained than the other due to the `AlwaysTrue` concept, while before have the `IsSame` concept as well.

```cpp
template<typename V, typename W>
concept IsSame = std::is_same_v<V, W>;

template<typename T>
concept AlwaysTrue = true;

template<typename T, typename U>
requires IsSame<T, U> auto add(const T& t, const U& u)
{
  return t + u;
}

template<typename T, typename U>
requires IsSame<T, U>&& AlwaysTrue<T> auto add(const T& t, const↩
     U& u)
{
  return t + u;
```

Listing 1.53

```cpp
}
```

Listing 1.53

This code is then invoked with two variables a, and b, both of type `int`. The values itself do not matter at this point, the focus is on subsumption, which is about types and not values.

```cpp
int a = 1;
int b = 2;

const auto res = add(a, b);
```

Listing 1.54

The code as show compiles and produces the correct result so far. The second `add` function is selected, because this one is the more constraint one. So far this is the same as we saw with the multiple destructors of `optional` before. How about swapping `T` and `U` for the first `add` functions `IsSame` constraint? `IsSame` still yields `true`, as all arguments are of type `int`. Both functions use only concepts, so what can possibly go wrong?

```cpp
template<typename T, typename U>
requires IsSame<U, T> (A) The arguments are swapped T, U vs U, T
  auto add(const T& t, const U& u)
{
  return t + u;
}

template<typename T, typename U>
requires IsSame<T, U> (B) The arguments remain unchanged
  && AlwaysTrue<T> auto add(const T& t, const U& u)
{
  return t + u;
}
```

Listing 1.55

Well, if you try this and your code looks like the above, where the first `add` has swapped arguments for `IsSame` (A) and the second remains untouched (B) our friend the compiler tells us the following:

```
error: call to 'add' is ambiguous
  const auto res = add(a, b);
```

```
                    ^~~
note: candidate function [with T = int, U = int]
  auto add(const T& t, const U& u)
       ^
note: candidate function [with T = int, U = int]
  && AlwaysTrue<T> auto add(const T& t, const U& u)
                        ^
1 error generated.
```

Interesting, now the call to `add` is ambiguous. That implies that after this little argument swap the compiler can now longer detect which of the two `add` functions is more constrained. That is probably a bit unexpected. Let's get an understanding of how the compiler approaches this.

The compiler sees the template arguments in this case as the textual values we use, not the types after or during instantiation. With that we have `IsSame<U, T >` vs. `IsSame<T, U>`. To evaluate whether both concepts are the same, the compiler looks into their definition, which is `is_same_v<U, T>` and `is_same_v<T, U>`. Once again the textual names are kept, no types are involved. So far, `IsSame` is seen as different by the compiler. The deep look into the concept definition reveals the type-trait `is_same_v`. This is once again treated with the textual names and by that leads to two different `is_same_v` components in the eyes of the compiler. They are not the same, so they are not the same concept, and with that the compiler cannot treat `IsSame` from both `add` functions as the same and use `AlwaysTrue` for finding the most constraint function. This little textual change, let's the compiler think the two concepts are different. The compiler looks at an ambiguity here. Figure 1.4 shows a more visual explanation.

To make multi-parameter concepts work regardless of the argument order, we need to add a concept which does that swapping. In this case `IsSameHelper` which does call `std::is_same_v` as before. The difference is that `IsSame` makes two calls to `IsSameHelper`, one with the argument order `<V, W>` and another with `<W, V>`.

```cpp
template<typename V, typename W>
concept IsSameHelper = std::is_same_v<V, W>;

template<typename V, typename W>
concept IsSame = IsSameHelper<V, W>&& IsSameHelper<W, V>;
```

Listing 1.56

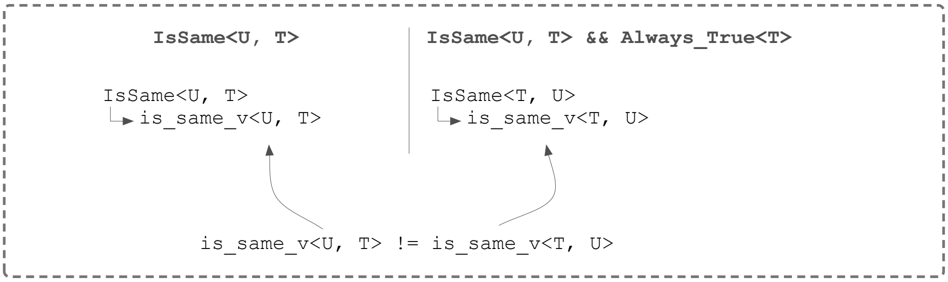


Figure 1.4: Two `is_same_v` differ in their argument order and with that considered different by the compiler.

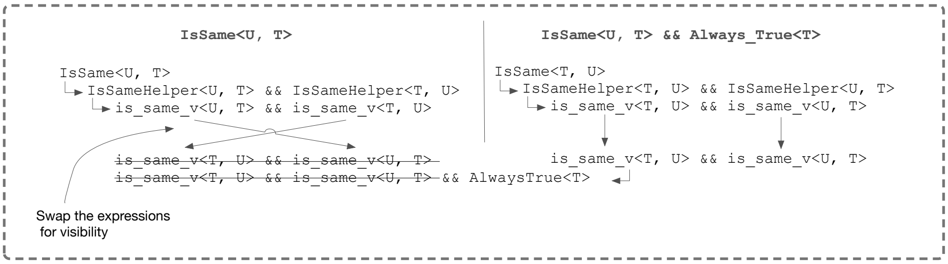

IsSame<U, T>
IsSame<U, T> && Always_True<T>
IsSame<U, T>
IsSameHelper<U, T> && IsSameHelper<T, U>
is_same_v<U, T> && is_same_v<T, U>
IsSame<T, U>
IsSameHelper<T, U> && IsSameHelper<U, T>
is_same_v<T, U> && is_same_v<U, T>
~~is_same_v<T, U> && is_same_v<U, T>~~
~~is_same_v<T, U> && is_same_v<U, T>~~ && AlwaysTrue<T>
is_same_v<T, U> && is_same_v<U, T>
Swap the expressions for visibility

Figure 1.5: `IsSame` concept with helper to make swapped arguments work.

By introducing this helper, we add the argument swapping in `IsSame` with the help of `IsSameHelper`. The latter one is a concept, remember that only concepts can take part in subsumption. We also call `IsSameHelper` in `IsSame` with both forms. With that the compiler can eliminate `IsSame` as before. This elimination leaves the second `add` function with `AlwaysTrue`. This function is now the most constraint one and subsumes the `add` definition without `AlwaysTrue`. No ambiguities any more.

We can conclude here, for multi-parameter concepts, use an indirection helper concept to make swapped arguments yield to the same result.

1.13.2 One more thing, never say not

In the last part we already learned that the compiler treats concepts during subsumption evaluation a bit different from usual. Subsumption evaluation is more textual driven. There is another part to consider, this time when applying concepts.

Let's assume we'd like an add function which has the requirement opposite of what we had before. The two arguments must not be of the same type. Modifying the last example is easy, we change IsSame<T,U> into not IsSame<T,U>. Doing this is indeed simple, but only to create a compile error again. The message is more or less same as before, except that this time we use int and double as arguments to add. What I described is shown in code below.

```cpp
template<typename T, typename U>
requires(not IsSame<T, U>) Ⓐ Inverting IsSame with not
 auto add(const T& t, const U& u)
{
  return t + u;
}

template<typename T, typename U>
  requires(not IsSame<T, U>) Ⓑ Inverting IsSame with not
  && AlwaysTrue<T> auto add(const T& t, const U& u)
{
  return t + u;
}
```

Listing 1.57

For completeness, here is the add and the arguments it is invoked with:

```cpp
int a = 1;
double b = 2;

Ⓒ Call to add is again ambiguous
// const auto res = add(a, b);
```

Listing 1.58

With the code as shown above and the arguments applied to it, we get the following error message from the compiler:

```
error: call to 'add' is ambiguous
  const auto res = add(a, b);
                   ^~~
note: candidate function [with T = int, U = double]
  auto add(const T& t, const U& u)
       ^
note: candidate function [with T = int, U = double]
  && AlwaysTrue<T> auto add(const T& t, const U& u)
```

```
                        ^
note: similar constraint expressions not considered equivalent;
    constraint expressions cannot be considered equivalent unless
    they originate from the same concept
requires (not IsSame<T,
          ^~~~~~~~~~~~~
note: similar constraint expression here
requires (not IsSame<T, U>)
          ^~~~~~~~~~~~~~~~
1 error generated.
```

What did we do now? We used the helper concept as before; we did not even swap the arguments so the helper is not necessary. We use concepts so subsumption should apply. As you can see in the last part, yes the arguments are `int` and `double` which are two different types so that should compile and as before the second add function is the one that is the most constraint. At least from my point of view. However, the compiler disagrees. The reason the compiler disagrees is the `(not IsSame<T, U>)`. The moment we apply `not` or `!` to a concept, the operands become part of the expression. In this context expression can be seen as source location. Two concepts are only equal, if they originate from the same written source (location). By adding the `not` and making that entire part an expression, the two negated `IsSame` are different concepts in the view of the compiler.

A guiding rule here, stay positive with your concepts and try to avoid negating them. For those who wondered, this is the reason why I chose `NotTriviallyDestructible` in the `optional` example instead of `not TriviallyDestructible`.

1.14 Improved error message

So far we have seen how to write and apply concepts. By that we also saw the improvements they give us when writing generic code. I often pointed out that Concepts also affects using function templates or class templates. With named concepts instead of `typename` the interface is much more distinguished. One other at least implicit goal of concepts is improving the error messages you get from your beloved compiler in case you supplied an invalid argument for a template. For illustration, let's think about a function template `PrintSorted`. This function does, hopefully, what the name implies. `PrintSorted` takes a container as an argument and sorts the values in the container. The function does so by using the STL algorithm `std::sort`. Afterwards `PrintSorted` prints each element of the container. In general a pretty straightforward function, `PrintSorted` is a template because the function should work with any container.

In addition, we have another function `sortedVector` which declares a `std::vector<int>` with several values. This function then calls the former function `PrintSorted`. All in all a few lines of code.

```cpp
template<typename T>
void PrintSorted(T c) (A) By copy to be able to sort it
{
  std::sort(c.begin(), c.end());

  for(const auto& e : c) { std::cout << e << ' '; }

  std::cout << '\n';
}

void sortedVector()
{
  std::vector<int> v{30, 4, 22, 5};

  PrintSorted(v);
}
```

Listing 1.59

Above you see an example implementation. For some of you this probably seems very easy and you might think what does he want now? Bare with me for a moment, please. There are potential others who had to think a bit longer. We all have in common that at some point we have written code like this. Generic code using a container and applying a STL algorithm to it. Those who found the example above trivial try to think about one of the first times you did implement it. Those of you who are newer may remember that moment easily. Now that we are all on the same page, at one point someone uses the great `PrintSorted` and passes a different STL container as argument. For example, `std::list`. For consistency we assume that there is another function `sortedList` which declares a `std::list` and initializes this list with multiple unsorted values. After that `sortedList` passes the list to `PrintSorted`. I know, fairly easy. In code we have this:

Listing 1.60

```cpp
void sortedList()
{
  std::list<int> l{36, 2, 5};

  PrintSorted(l);
}
```

Now be honest to yourself, do or did you think that this code compiles? Don't bother looking for a missing semicolon, the code is semantically correct. However, this code does not compile. Why? Well, because `std::sort` requires a container which fulfils the `random_access_iterator` concept. `std::vector` does, sadly `std::list` doesn't. Simply something we have to learn whether we like it or not. `std::list` come with a class method `sort` for sorting its data. If you try to compile the example with `std::list` calling `PrintSorted` you get a lot of error output. On my machine 520 lines! The last line says `8 errors generated`, yet the error output needs 519 lines to tell me that. The output consists of 32520 bytes. To be fair, a longer or shorter path to the compiler changes this a bit. Here are the first 13 lines for illustration:

```
In file included from /usr/local/gcc/10.2.0/include/c++/10.2.0/↩
    algorithm:62,
                 from printSorted0/main17.cpp:1:
/usr/local/gcc/10.2.0/include/c++/10.2.0/bits/stl_algo.h: In ↩
    instantiation
    of 'void std::__sort(_RandomAccessIterator, _RandomAccessIterator↩
        , _Compare)
    [with _RandomAccessIterator = std::_List_iterator<int>; _Compare ↩
        = __gnu_cxx::__ops::_Iter_less_iter]':
/usr/local/gcc/10.2.0/include/c++/10.2.0/bits/stl_algo.h:4859:18:
    required from 'void std::sort(_RAIter, _RAIter) [with _RAIter = ↩
        std::_List_iterator<int>]'
printSorted0/printSorted0.cpp:4:12:   required from 'void PrintSorted↩
    (T) [with T = std::__cxx11::list<int>]'
printSorted0/printSorted0.cpp:22:16:   required from here
/usr/local/gcc/10.2.0/include/c++/10.2.0/bits/stl_algo.h:1975:22:
    error: no match for 'operator-' (operand types are 'std::↩
        _List_iterator<int>' and 'std::_List_iterator<int>')
 1975 |     std::__lg(__last - __first) * 2,
      |                 ~~~~~~~^~~~~~~~~
In file included from /usr/local/gcc/10.2.0/include/c++/10.2.0/bits/↩
    stl_algobase.h:67,
                 from /usr/local/gcc/10.2.0/include/c++/10.2.0/↩
                     algorithm:61,
                 from printSorted0/main17.cpp:1:
/usr/local/gcc/10.2.0/include/c++/10.2.0/bits/stl_iterator.h:500:5:
    note: candidate: 'template<class _IteratorL, class _IteratorR> ↩
        constexpr
        decltype ((__y.base() - __x.base())) std::operator-(const std↩
            ::reverse_iterator<_Iterator>&, const std::↩
            reverse_iterator<_IteratorR>&)'
...
```

Some of us are used to errors like this and manage to quickly understand what the problem is. I already revealed it, `std::list` does not meet the `random_access_iterator` concept. Well, we can see a lot of `iterator...` in the errors. The question is, do concepts help here?

The Standard comes with a `random_access_iterator` concept, we can use this and replace **`typename`** in `PrintSorted`. That alone makes the function clearer to users, but I mentioned that before already.

```cpp
template<random_access_iterator T>
void PrintSorted(T c)
{
  std::sort(c.begin(), c.end());

  for(const auto& e : c) { std::cout << e << ' '; }

  std::cout << '\n';
}

void sortedVector()
{
  std::vector<int> v{30, 4, 22, 5};

  PrintSorted(v);
}
```

Listing 1.61

By doing that and leaving everything else unchanged the error output is reduced to 16 lines, stating that there is 1 error.

```
printSorted1/list.cpp:5:3: error: no matching function for call to '↩
    PrintSorted'
  PrintSorted(l);
  ^~~~~~~~~~~
printSorted1/printSorted1.cpp:2:6: note: candidate template ignored: ↩
    constraints not satisfied
      [with T = std::__1::list<int, std::__1::allocator<int> >]
void PrintSorted(T c)
     ^
printSorted1/printSorted1.cpp:1:10: note: because 'std::__1::list<int↩
    , std::__1::allocator<int>
      >' does not satisfy 'random_access_iterator'
template<random_access_iterator T>
         ^
printSorted1/main20.cpp:9:34: note: because '!requires (std::__1::↩
    list<int,
      std::__1::allocator<int> > t) { t.sort(); }' evaluated to false
concept random_access_iterator = not requires(I t)
                                 ^
1 error generated.
```

To make the example work we provide an overload for `PrintSorted` which checks whether a type has a `sort` method. In that case, instead of calling `std::sort` this version of `PrintSorted` calls the method `sort` of that type.

```cpp
template<typename T>
concept HasSortMethod = requires(T t)
{
  t.sort();
};

template<HasSortMethod T>
void PrintSorted(T c)
{
  c.sort();

  for(const auto& e : c) { std::cout << e << ' '; }

  std::cout << '\n';
```

Listing 1.62

```
}
```

Listing 1.62

The interface is clear, the error messages are down to the point. A lot of people I spoke to say that the significant thing about concepts is that they shorten the error messages. I agree. For me the most compelling reason is the ability to express the interface a type has to comply to takes away a lot of theses minor errors which blow up to page-wise error output. But, if at some point we got the interface wrong, the error messages are way more helpful than ever before. The example above is probably one which I will make with concepts again and will be happy about the brief error message. There are other usages where I will no longer have to guess and get the interface right the first time, then I have no need for the short error messages.

Cliff notes

- `constexpr` function can be called in a nested requirement, but the parameters shall not be from the parameter list of a requires-expression.
- A concept always yields a boolean value, never a type.
- Missing template arguments are injected by the compiler, if the compiler can deduce them, from the left to the right.
- A `constexpr` function can be called in a nested requirement, but the parameters shall not be from the parameter list of a requires-expression.
- Remember, a requires-clause and a requires-expression are two different things.
- Always test your requirements, otherwise the errors you are getting are hard to track to the root cause.
- C++20 allows `auto` as function parameter. Such a function becomes a function template. The parameter can be constrained by a concept. This syntax is called terse-syntax.
- The trailing requires-clause helps to create conditional methods, even special members, without code overhead.
- With C++20 we can prefix all occurrences of `auto` with a concept.
- Every `typename` or `class` in a template head can be replaced with a concept.
- Always test your concepts, otherwise the errors you are getting are hard to track to the root cause.
- Concepts can be the interface definition of an object.
- The subsume rules apply only to concepts. They can consume each other. However, all expressions are atomic constraints.

Chapter 2

Coroutines

A coroutine is a function that can suspend itself. The term coroutine is well-established in computer science since it was first coined in 1958 by Melvin Conway [2]. It is just that C++ needed a very long time to add coroutines as a language feature. In this chapter, we will look at what coroutines are and how they influence and change the way we (can) write code.

2.1 Regular functions and their control flow

Before we look at coroutines, let's first have a look at regular functions and their control flow. Because, let's face it, we have written tons of code without coroutines in C++, so to understand what they are for and where they can improve our code, we need to first understand the limits of regular functions.

Consider the following example and think briefly for yourself what is wrong with that small piece of code?

```cpp
for(int i = 0; i < 5; ++i) { UseCounterValue(i); }
```

Listing 2.1

We all may conclude a bunch of different things we would individually do differently. However, I hope you agree that one issue with that piece of code is the missing separation between creating the numbers and passing them to a function. In other words, we have a mixture of generation and usage. By looking at a broader picture where `UseCounterValue` is called later again, the issue becomes more visible:

```cpp
for(int i = 0; i < 5; ++i) { UseCounterValue(i); }

// other code

for(int i = 5; i < 10; ++i) { UseCounterValue(i); }
```

Listing 2.2

We now have to write the `for`-loop twice, and we have to get the boundaries right. Doesn't sound easy knowing that the most common issue in computer science is the off-by-one error.

A more general view on this code reveals that we have an algorithm that generates the numbers from 1 to N. Then, we use the generated data in `UseCounterValue`. Such a separation would improve our code, because we then could reuse the `for`-loop. In addition a separation would make it more robust.

What alternatives do we have? Well, we can use a lambda with init-capture, but then making the lambda run only to 5 is hard. Another alternative is to write a function template that takes a callable as one parameter. This would allow us to pass the using-part to the generator. But then it is not easy to see what parameters this callable takes. Another approach is to use a function with a `static` variable that stores the current counter value and increments the value during `return`. We then use this function `counter` and pass the return value to `UseCounterValue`. Here is a possible implementation:

```cpp
int counter()
{
  static int i{0};
  return i++;
}

void UseCounter()
{
  for(int i = 0; i < 5; ++i) { UseCounterValue(counter()); }

  // other code
```

Listing 2.3

```cpp

  for(int i = 0; i < 5; ++i) { UseCounterValue(counter()); }
}
```

Listing 2.3

That works, but this solution is far from being nice. We can use `counter` only for one algorithm, as it counts up monotonically, and we have no way to reset the internal value of `counter`. Even if we would add some kind of reset mechanism, `counter` will never be usable in a multi-threading context.

Ideally, the algorithm's generator could just be resumed where it was suspended, keeping its state. We could also spawn thousands of instances of the same generator, each of them starting at the same initial value, and keeping track of its own state. This suspend and resume with state preservation is exactly what coroutines give us.

2.2 What are Coroutines

We have just looked at regular functions and their limitations. Now it is time to see what coroutines can do for us, or in other words, what are resumable functions. Figure 2.1 shows the difference in the control flow between a regular function and a coroutine control flow. As we can see there, a regular function is executed once in total. In contrast to that, a coroutine can suspend itself and return to the caller. This can happen multiple times and a coroutine is also able to call another coroutine.

2.2.1 Generating a sequence with coroutines

Instead of having a non-interruptible control flow where some kind of callback mechanism is required to signal a change to the caller, we can use coroutines to suspend and resume them. Here is such an implementation:

```cpp
IntGenerator (A) Returning a coroutine object
counter(int start, int end)
{
  while(start < end) {
    co_yield start; (B) Yielding a value and giving control back to the
        caller
```

Listing 2.4

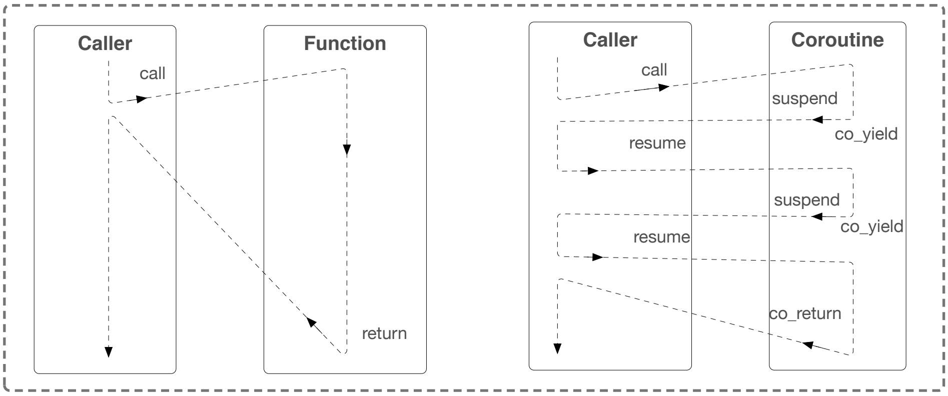


Figure 2.1: The difference in control flow between a regular function and a coroutine.

```
    ++start;
  }
}

void UseCounter()
{
  auto g = counter(1, 5);

  C This sequence runs from 1 to 5
  for(auto i : g) { UseCounterValue(i); }
}
```

Listing 2.4

As we can see, A does not simply return an `int`. `IntGenerator` returns some coroutine object. We will talk about this in a minute. There are other things we need to understand first. In B, we can see that coroutines bring us new keywords, in this case, `co_yield`. With that way of writing and separating generation from usage, we can even use a nice range-based for-loop to iterate over the values `counter` gives us. Those of you who are familiar with other languages that already support coroutines will see that; except for the `co_` part before `yield` and the special return type, the syntax looks straight like that in other languages.

Coroutines are functions that can suspend themselves and be resumed by a using function. Every time a coroutine reaches a `co_yield`, the coroutine suspends itself and yields a value. While suspended, the entire state is preserved and used again when resumed. This is the difference to the definition of a function as we know it.

2.3 The elements of Coroutines in C++

Before we are ready to write and understand coroutines in C++, we need to cover the elements a coroutine consists of and some terminology. While doing that we will look at implementation strategies for coroutines used in C++.

2.3.1 Stackless Coroutines in C++

Coroutines come in two flavors, stackfull and stackless coroutines. C++20 brings us stackless coroutines. What that means is the following. A coroutine can be seen as a transformation of a coroutine-function into an finite state machine (FSM). The FSM maintains the internal state, where the coroutine was left when it returned earlier, the values that were passed to the corotuine upon creation. This internal state of the FSM, as well as the values, passed to the coroutine, need to be stored somewhere. This storage section is called a coroutine-frame.

The approach C++20 implements is to store the coroutine-frame in astackless manner, meaning the frame is allocated on the heap. As we will later see, the heap-allocation is done by the compiler automatically every time a coroutine is created.

2.3.2 The new kids on the block: `co_await`, `co_return` and `co_yield`

We saw in our previous coroutine example that there is a new keyword `co_yield`. Aside from this, we have two others: `co_return` and `co_await`. Whenever we use one of these keywords in a function, this function automatically becomes a coroutine. In C++, these three keywords are the syntactic markers for a coroutine.

The difference between the three keywords is summarized in Table 2.1.

2.3.3 The generator

When we look at our initial example of a coroutine, we can see therein at Ⓐ a type `IntGenerator`. Behind this hides a special type required for a coroutine. In C++, we cannot have a plain return type like `int` or `std::string`. We need to wrap the

Table 2.1: Coroutine keywords

Keyword	Action	Type	State
`co_yield`	Output	`promise`	Suspended
`co_return`	Output	`promise`	Ended
`co_await`	Input	`awaitable`	Suspended

type into a so-called generator. The reason is that coroutines in C++ are a very small abstraction for an FSM. The generator gives us implementation freedom and the choice of how we like to model our coroutine. For a class or struct to be a generator type, this class needs to fulfill an API required to make the FSM work. There was a new keyword `co_yield` in the `counter` example, which suspends the coroutine and returns a value to the caller. However, during the suspension, the state of the coroutine needs to be stored somewhere, plus we need a mechanism to obtain the yielded value from outside the coroutine. A `generator` manages this job. Below you see the generator for `counter`.

```
template<typename T>
struct generator {
  using promise_type =
    promise_type_base<T, generator>; A The PromiseType
  using PromiseTypeHandle = std::coroutine_handle<promise_type>;

  B Make the generator iterable
  using iterator = coro_iterator::iterator<promise_type>;
  iterator begin() { return {mCoroHdl}; }
  iterator end() { return {}; }

  generator(generator const&) = delete;
  generator(generator&& rhs)
  : mCoroHdl(std::exchange(rhs.mCoroHdl, nullptr))
  {}

  ~generator()
```

Listing 2.5

```
  {
    C We have to maintain the life-time of the coroutine
    if(mCoroHdl) { mCoroHdl.destroy(); }
  }

private:
  friend promise_type; D As the default ctor is private
      promise_type needs to be a friend

  explicit generator(promise_type* p)
  : mCoroHdl{PromiseTypeHandle::from_promise(*p)}
  {}

  PromiseTypeHandle mCoroHdl; E The coroutine handle
};
```

Listing 2.5

At the very top at A, we see `using promise_type`. This is a name the compiler looks for. The promise type is slightly comparable with what we already know from the `std::promise` part of an `std::future`. However, probably a better view is to see the `promise_type` as a state controller of a coroutine, hence its `promise_type`, does not necessarily give us only one value. We will look at the `promise_type` after the `generator`.

At B, we see an `iterator`. This is due to our range-based for-loop needing a `begin` and `end`. The implementation of `iterator` is nothing special, as we will see after the `promise_type`.

Next, we look at C and E, as they belong together. In E, we see the coroutine handle. This handle is our access key to the coroutine state machine. This type, `std::coroutine_handle<T>` from the new header `<coroutine>`, can be seen as a wrapper around a pointer, pointing to the coroutine frame. A special thing about the coroutine frame is that the compiler calls `new` for us whenever a coroutine, and with that, a `generator` and `promise_type`, is created. This memory is the *coroutine-frame* The compiler knows when a coroutine is started. However, the compiler does not, at least easily, know when a coroutine is finished or no longer needed. This is why we have to free the memory for the coroutine-frame ourselves. The easiest way is to let generator free the coroutine-frame in its destructor, as we can see in Ⓒ. The memory resource is also the reason why generator is move-only.

We left out Ⓓ so far, the constructor of the generator, and the friend declaration. If you look closely, you will see that the constructor of generator is private. That is because generator is part of promise_type, or better promise_type_base, as you can see at Ⓐ. During the allocation of the coroutine-frame, the promise_type is created. Let's have a look at promise_type_base.

2.3.4 The promise_type

The promise_type using in the generator implementation is a hook for the compiler. Once a compiler sees a promise_type in a class, it uses the using alias to look up the type behind it, checking whether this type fulfills the promise-type interface. First, here is an implementation of promise_type_base:

```
template<typename T, typename G>
struct promise_type_base {
  T mValue; Ⓐ The value yielded or returned from a coroutine

  auto yield_value(T value) Ⓑ Invoked by co_yield or co_return
  {
    mValue = std::move(value); Ⓒ Store the yielded value for access
         outside the coroutine

    return std::suspend_always{}; Ⓓ Suspend the coroutine here
  }

  G get_return_object() { return G{this}; }; Ⓔ Return generator

  std::suspend_always initial_suspend() { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }
  void return_void() {}
  void unhandled_exception() { std::terminate(); }
  static auto get_return_object_on_allocation_failure() { return ↩
      G{nullptr}; }
};
```

Listing 2.6

We can first see at A that `promise_type_base` stores a value. This is the value we can yield with `co_yield` or `co_return` from the body of a coroutine to the caller. In B, we see as part of the promise-type API the function `yield_value`. With each call to `co_yield` or `co_return`, this function `yield_value` is called with the value that was used with `co_yield` or `co_return`. In `promise_type_base`, we use this hook to store the value in `mValue`, as at this point, we are still within the coroutine and the coroutine-frame. D ensures that the coroutine gets suspended after we return from `yield_value`.

We can see a bunch of other promise-type `API` methods in Listing 2.6. For now, the important one is `get_return_object`. Here we can see how the promise-type interacts with `generator`. Function `get_return_object` is called when the coroutine is created. This is what we store as `IntGenerator`. You can see `IntGenerator` as our communication channel into the coroutine FSM. The way the channel is created here is that `promise_type_base` passes its `this`-pointer to the constructor of `generator`. There is a way to obtain an `std::coroutine_handle` from a promise-type via `std::coroutine_handle<T>::from_promise`, which is exactly what `generator` does in the constructor. As you can see with this implementation, having `generator`'s constructor `public` makes no sense.

All this interaction between the different parts of a coroutine is depicted in Figure 2.2 using the `counter` example as a base.

2.3.5 An iterator for `generator`

The last part that we haven't looked at so far is the implementation of `iterator`, so let's do that now.

Listing 2.7

```cpp
namespace coro_iterator {
  template<typename PT>
  struct iterator {
    std::coroutine_handle<PT> mCoroHdl{nullptr};

    void resume()
    {
      if(not mCoroHdl.done()) { mCoroHdl.resume(); }
    }

```

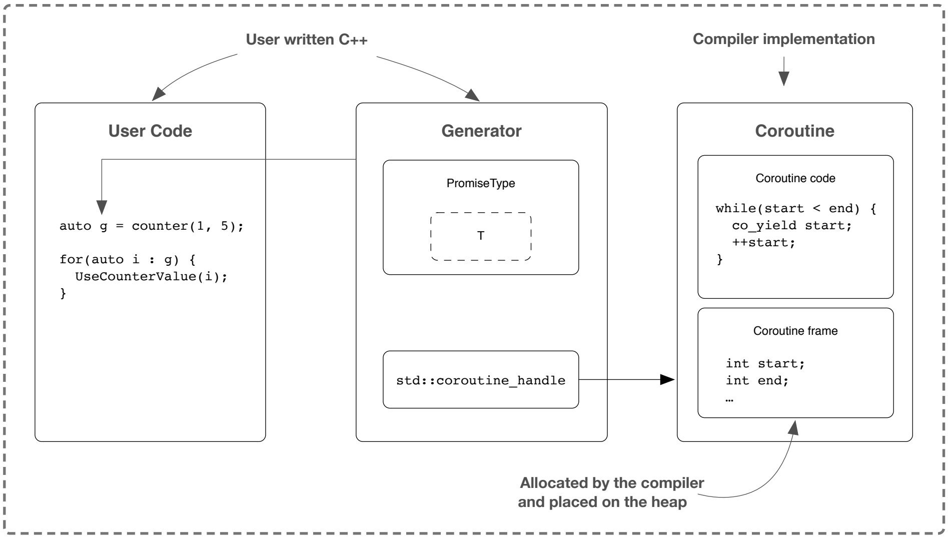


Figure 2.2: How the different coroutine elements interact with each other.

```
    iterator() = default;
    iterator(std::coroutine_handle<PT> hco)
    : mCoroHdl{hco}
    {
      resume();
    }

    void operator++() { resume(); }
    bool operator==(const iterator&) const { return mCoroHdl.done↩
        (); }
    const auto& operator*() const { return mCoroHdl.promise().↩
        mValue; }
  };
} // namespace coro_iterator
```

Listing 2.7

As you can see, the implementation of `iterator` is nothing special. This implementation is not different from any other iterator. One thing to point out here is that `iterator` does not call `destroy` on the coroutine handle. The reason is that generator controls the life-time of the coroutine-handle. iterator is only allowed to have a temporary view of the data.

2.3.6 Coroutine customization points

Figure 2.3 shows the flow of a coroutine and the customization points. We have already seen and used most of them in our example counter. There are probably two important additional customization points. We start with get_return_object_on_allocation_failure. The existence of this static method controls which operator new is called. For a PromiseType with that function, the operator new(size_t, nothrow_t) overload gets called. Should operator new return a nullptr, then get_return_object_on_allocation_failure is invoked to obtain some kind of backup object. This prevents the program from crashing due to a nullptr access. However, the coroutine will, of course, not run. Instead, such a coroutine looks like it has already finished. Without get_return_object_on_allocation_failure and a nullptr returned from the new-call, we end up with a nullptr access, which we can catch as an exception. With that, we can control the behavior on a failed allocation.

The second customization point we haven't talked about yet is unhandled_exception . We will do so in Section 2.9.

As we have a lot of implementation freedom with all the customization points, Table 2.2 summarizes all customization points and their return types.

2.3.7 Coroutines restrictions

There are some limitations in which functions can be a coroutine and what they have to look like.

- constexpr-functions cannot be coroutines. Subsequently this is true for consteval-functions, which we will see later, as well.
- Neither a constructor nor a destructor can be a coroutine.
- A function using varargs. A variadic function template works.
- A function with plain auto as return-type, or with a concept type, cannot be a coroutine. auto with trailing return-type works.

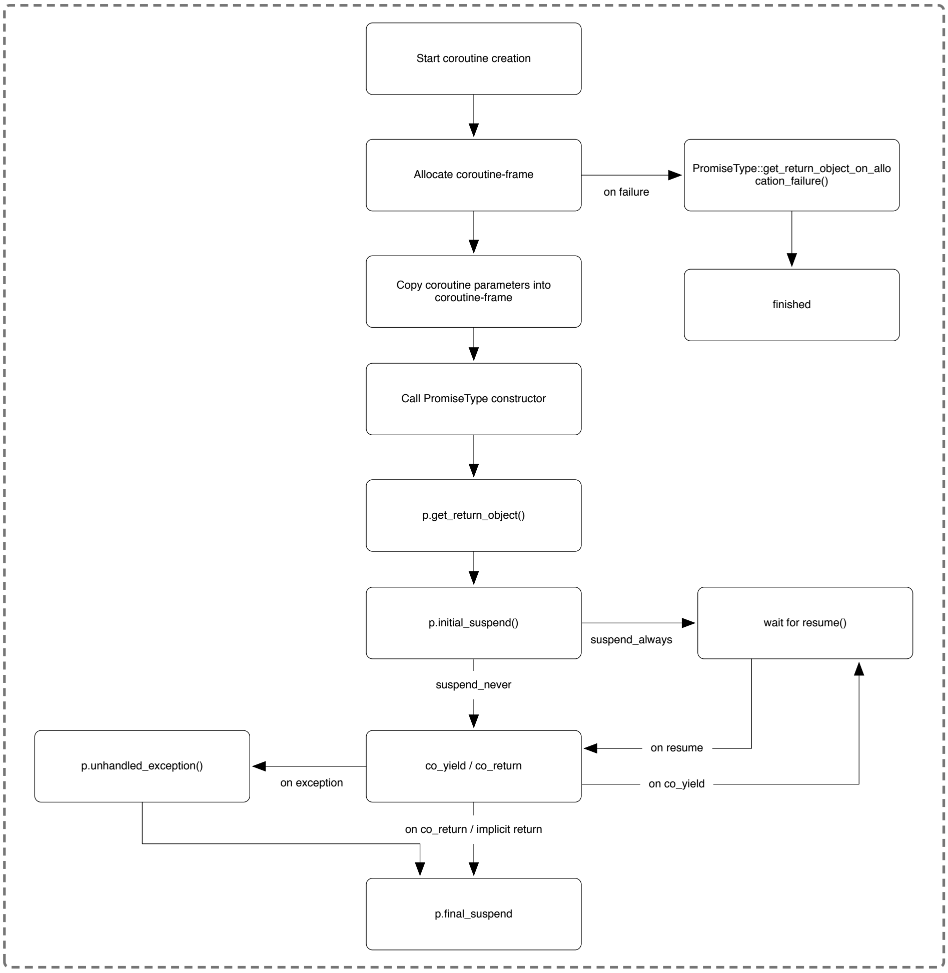


Figure 2.3: The coroutine state machine and the `PromiseType` customization points.

- Further, a coroutine cannot use plain `return`, it must be either `co_return` or `co_yield`.

- And last but not least, `main` cannot be a coroutine.

Lambdas, on the other handm can be coroutines.

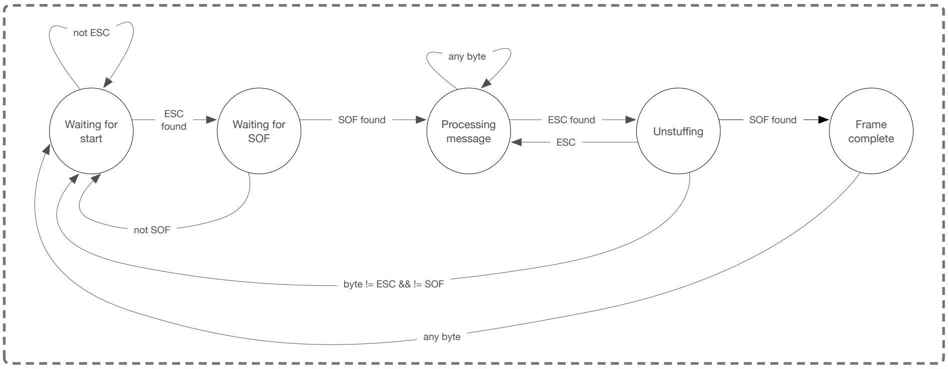


Figure 2.4: State machine diagram of the byte-stream parser protocol.

2.4 Writing a byte-stream parser the old way

Now that we have seen the elements of a coroutine, let's apply them to an example. We will see how coroutines can improve our code, using asexample the writing of a parser.

Suppose that we like to parse a data-stream containing the string "`Hello World`" for this example.

In the past, I often have written a data-stream parser. The task is to parse a stream of arbitrary data that is divided into frames. The trick is to detect the start and end of a frame. Most of the time, I used some variation of what Tanenbaum describes in his book *Computer Networks* [3] to implement the parser and the protocol.

Now, what do we have there? A protocol that contains a special `ESC` byte (not to be confused with American Standard Code for Information Interchange (ASCII) ESC, 0x1B), which stands for escape. `ESC`'s job is to escape all other protocol bytes. In the payload, each byte that has the same value as `ESC` is escaped with `ESC` as well. That way, the protocol can contain the full width of a byte and not only printable characters. With this marker, we can search in a byte-stream for the beginning of a frame. A frame starts with `ESC` + `SOF`, where `SOF` stands for Start Of Frame. Figure 2.4 shows the statechart and control flow of a parser for this protocol.

The two protocol flags are encoded as follows:

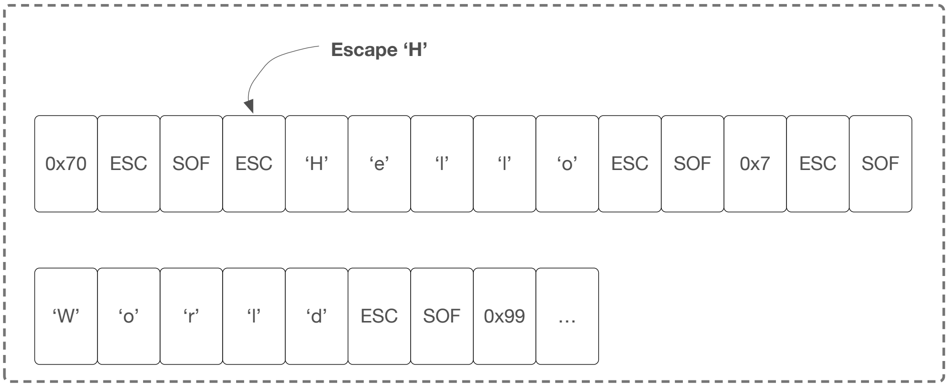


Figure 2.5: Input stream for the parser.

```cpp
static const byte ESC{'H'};
static const byte SOF{0x10};
```

Listing 2.8

Using 'H' as ESC was a deliberate choice. That way, we need to escape ASCII data when we assume that we use our parser with the input stream as shown in Figure 2.5. This is the case for the string `Hello World` we like to process.

How can we implement this protocol? There are various ways. Most likely we tend to create a class to track the state. Literature also tells us about various patterns we can apply, such as the state pattern. Below, I show the likely more unusual approach, an implementation in a single function using `static` variables.

```cpp
template<typename T>
void ProcessNextByte(byte b, T&& frameCompleted)
{
  static std::string frame{};
  static bool inHeader{false};
  static bool wasESC{false};
  static bool lookingForSOF{false};

  if(inHeader) {
    if((ESC == b) && not wasESC) {
      wasESC = true;
```

Listing 2.9

```cpp
      return;
    } else if(wasESC) {
      wasESC = false;

      if((SOF == b) || (ESC != b)) {
        // if b is not SOF discard the frame
        if(SOF == b) { frameCompleted(frame); }

        frame.clear();
        inHeader = false;
        return;
      }
    }

    frame += static_cast<char>(b);

  } else if((ESC == b) && !lookingForSOF) {
    lookingForSOF = true;
  } else if((SOF == b) && lookingForSOF) {
    inHeader = true;
    lookingForSOF = false;
  } else {
    lookingForSOF = false;
  }
}
```

Listing 2.9

One view on this implementation is that the code is somewhat ugly. I'm not really using C++ here. It is more like C. No classes and nasty `static` variables. We cannot have two `Parse` routines working at the same time. In fact, we even have no way to reset `Parse` once the function was called the first time. That all yells class! I know. But if you look at `Parse`, you can see that everything is there. Whether the states are really better traceable in a class design is not always the case. I chose this implementation on purpose, not because of the ugliness but because this implementation is and should be all we need. However, with normal functions, this is what we end up with. Then came Object Oriented Programming (OOP), and we started using classes in C++. That was many years ago. Now we have C++20 and coroutines. Let's see how this code looks if we transform it to coroutines in the next section.

2.5 A byte-stream parser with Coroutines

The task is to improve the parser from the former section by using coroutines. We already have seen some of the pieces we need to refactor the byte-stream parser from regular functions into coroutines. This time, let's start by looking at how we can implement the `Parse` function by the use of coroutines. Once this is done, we develop the relevant pieces to make `Parse` work.

2.5.1 Writing the `Parse` function as coroutine

The heart of this whole byte-stream parser is the `Parse` function. Of course, `Parse` is a coroutine. `Parse` returns the type `FSM`, a different kind of generator. At this point, we can see the power coroutines give us. But see for yourself before I walk you through the code. Here is the implementation of `Parse`:

```
FSM Parse()
{
 while(true) {
   byte b = co_await byte{};
   std::string frame{};

   if(ESC == b) {
     b = co_await byte{};

     if(SOF != b) { continue; } (A) not looking at a start sequence

     while(true) { (B) capture the full frame
       b = co_await byte{};

       if(ESC == b) {
         (C) skip this byte and look at the next one
         b = co_await byte{};
```

Listing 2.10

```

          if(SOF == b) {
            co_yield frame;
            break;
          } else if(ESC != b) { D out of sync
            break;
          }
        }

        frame += static_cast<char>(b);
      }
    }
  }
}
```

Listing 2.10

Remember, in the pure function implementation before, we had to keep track of some, or more accurately, a lot of states. This was in the form of `frame` and whether we were already in the header. Keeping track of whether the last byte was an `ESC` was also necessary. All this state is now automatically maintained by the compiler in the coroutine-frame. Thanks to `co_await`, we can simply suspend `Parse` and wait for the next byte to come in. As you can see, we use `co_await` in four places. First, whenever the outer **`while`**-loop starts and later when we receive the first `ESC` byte. The two remaining calls are while the parser is in a frame and processes the data. Handling the case of `ESC` inside a frame is simple and straightforward, thanks to `co_await`. In all these places where we needed additional variables to track state, we can now simply suspend and resume the coroutine at the same point with the exact same state. No additional state-keeping necessary, except for the data in the frame, which upon completion is, of course, `co_yield`'ed to the caller of `Parse`. The caller can now ask via the coroutine handle, which is returned by `Parse` for the result.

To me, this implementation looks much more readable than the pure function implementation. Especially if you are looking at how nicely readable the control flow in `Parse` is now. Everything is in one place. No dubious callbacks are required to signal a caller a state change. Simple things like resetting the `std::string`, which holds the frame data, is now managed automatically by the C++ life-time thanks to the **`while`**-loop.

Another great benefit about this implementation is that `Parse` can now be used multiple times. Thanks to the state for each new call to `Parse` stored in a coroutine-frame, we can easily have dozens or hundreds of `Parse` coroutines active in parallel, parsing different data-streams. That all without the need to write a class. I hope at this point you can understand why I chose the ugly function approach before.

2.5.2 Creating an Awaitable type

In `Parse`'s implementation above, we first saw `co_await` to wait in a coroutine for data from outside the coroutine. This data is provided asynchronously. Invoked as above, the compiler looks whether our `PromiseType` also provides the interface for an Awaitable type. A PromiseType must provide a method `await_transform` to be an Awaitbale type. This method takes a type as a single argument and returns an Awaiter. An Awaiter works the same as we saw it before with PromiseType. There are a couple of symbols the compiler tries to look up in that type. If they are found, the type is an Awaiter. If not, we will get a compile error. Table 2.3 provides an overview of the three methods `bool await_ready()`, `void await_suspend( coroutine_handle<>)`, and `void await_resume()` of an Awaiter interface.

You may have noticed that in `Parse`, we have `co_await byte`. This tells the compiler to look for either an `operator co_await(byte)` or for `await_transform( byte)`. The difference is whether our PromiseType is also an Awaiter and we can use `co_yield` and `co_await` inside the coroutine, or if we need to invoke another type with `co_await`. For now, let's use the version where the PromiseType is an Awaiter as well. This type I like to call `async_generator`.

The `async_generator` or, more precisely, the Awaitable-part needs to store the awaited data. From the interface, we already saw that an `Awaiter` needs to implement, we know that there is an `await_ready` method which checks whether the value is already present. This sounds much like a `std::optional` use-case.

```
template<typename T>
struct awaitable_promise_type_base {
 std::optional<T> mRecentSignal;

 struct awaiter {
  std::optional<T>& mRecentSignal;
```

Listing 2.11

```cpp

    bool await_ready() { return mRecentSignal.has_value(); }
    void await_suspend(std::coroutine_handle<>) {}

    T await_resume()
    {
      assert(mRecentSignal.has_value());
      auto tmp = *mRecentSignal;
      mRecentSignal.reset();
      return tmp;
    }
  };

  [[nodiscard]] awaiter await_transform(T) { return awaiter{↩
      mRecentSignal}; }
};
```

Listing 2.11

Aside from the Awaiter interface, the async_generator needs to provide await_transform as part of the PromiseType. Other than that, we can more or less copy the implementation of generator we already developed. One difference is the promise_type, which now passes awaitable_promise_type_base. The second is that we need a way to retrieve the data yielded by the coroutine and to pass data to the coroutine. Here is a possible implementation of async_generator.

```cpp
template<typename T, typename U>
struct [[nodiscard]] async_generator
{
  using promise_type = promise_type_base<T, async_generator,
                                         awaitable_promise_type_base<U>>;
  using PromiseTypeHandle = std::coroutine_handle<promise_type>;

  T operator()()
  {
    A the move also clears the mValue of the promise
    auto tmp{std::move(mCoroHdl.promise().mValue)};

```

Listing 2.12

```
    B but we have to set it to a defined state
    mCoroHdl.promise().mValue.clear();

    return tmp;
  }

  void SendSignal(U signal)
  {
    mCoroHdl.promise().mRecentSignal = signal;
    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
  }

  async_generator(async_generator const&) = delete;
  async_generator(async_generator && rhs)
  : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)}
  {}

  ~async_generator()
  {
    if(mCoroHdl) { mCoroHdl.destroy(); }
  }

private:
  friend promise_type; C As the default ctor is private G needs to be a
      friend
  explicit async_generator(promise_type * p)
  : mCoroHdl(PromiseTypeHandle::from_promise(*p))
  {}

  PromiseTypeHandle mCoroHdl;
};
```

Listing 2.12

The `async_generator` comes with two member functions, `GetResult` and `SendSignal`. The former retrieves the yielded value, while the latter sets an awaited value. With that, we have the two required communication channels. There is a third change that's more subtle. I left the iterator part out. The `async_generator`, as we need it for Parse, does not need to be iterable. This is a design choice. Other use-cases might require an iterator.

2.5.3 A more flexible promise_type

With the awaiter_promise_type that we saw in the previous section, we have a decision to make. There are now two placed where we need a Promise-Type, in async_generator and generator. One option is to reimplement promise_type_base and duplicate the code into awaiter_promise_type to make awaiter_promise_type a full working PromiseType. I'm no fan of such an approach. We can do better with a slight modification to promise_type_base. We can let promise_type_base derive from awaiter_promise_type. That way, we can keep the two independent implementations without any duplication. With the inheritance approach, promise_type_base can derive from other types as well.

What we have to do is to allow promise_type_base to derive from multiple base classes. Why multiple? Just to be flexible in the future. Variadic templates allow us this easily. All we have to do is to add a variadic-type template parameter Bases to the template-head of promise_type_base and let promise_type_base derive from these bases if there are any. Thanks to variadic templates, this parameter pack can also be empty, and our code compiles. Here is the upgraded version:

Listing 2.13

```
template<typename T, typename G,
    typename... Bases> (A) Allow multiple bases for awaiter
struct promise_type_base : public Bases... {
  T mValue;

  auto yield_value(T value)
  {
    mValue = value;
    return std::suspend_always{};
  }

  G get_return_object() { return G{this}; };

  std::suspend_always initial_suspend() { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }
```

```cpp
  void return_void() {}
  void unhandled_exception() { std::terminate(); }
};
```

Listing 2.13

As you can see, we needed to change only the first two lines of `promise_type_base` as described above, and we are done. Much, much better than duplicating code.

2.5.4 Another generator the FSM

So far, we looked at a simple generator that yields a value at some point. This is what the former implementation of `generator` did. A look at `Parse` reveals that we need more than that this time. `Parse` does not only yield a value every time a frame is complete. `Parse` also uses `co_await` to suspend and wait for the next available byte. To make this possible, the generator this time needs to satisfy another interface, the `awaiter`.

```cpp
using FSM = async_generator<std::string, byte>;
```

Listing 2.14

2.5.5 Simulating a network byte stream

To make the byte-stream parser work, we need some kind of simulator again. This time, instead of using just an `std::vector` again, we introduce another coroutine `sender`, which is shown below.

```cpp
generator<byte> sender(std::vector<byte> fakeBytes)
{
  for(const auto& b : fakeBytes) { co_yield b; }
}
```

Listing 2.15

It returns the now well-known `generator`-type, here of type `std::byte`. As parameter, `sender` takes our former `std::vector`. Inside the coroutine, a range-based for-loop is used to iterate over the vector's elements yielding each element. This is comparable to a network data-stream that just runs until the connection is terminated.

When looking at these few lines of code, there is something very important to notice. As you can see, `sender` takes `fakeBytes`, our `std::vector` parameter, as a copy. This is no oversight. It is intentional! Why not by `const &`? The reason is that data passed to a coroutine is not necessarily copied into the coroutine. For example, in case the data passed to a coroutine lives longer than the coroutine, you can pass the data by `const &`, and the coroutine frame will only contain a reference to that data. As you probably know, if the coroutine lives longer than the data to which that reference points, we are looking at Undefined Behavior (UB). The safest way for a generic coroutine is to take parameters by copy and use move-semantics at the call side for efficiency. In case you are totally sure what you are doing and your future self and all your colleagues are sure forever as well, pass the data by `const &`.

2.5.6 Plugging the pieces together

At this point, we have all the pieces we need to create the full picture of the program. Remember, Figure 2.5 shows the data we like to simulate. We put them into two `std::vector`'s, use our former `sender` and `Parse` implementation, and we are more or less done. Below is a function which loops over a stream of bytes and feeds them into `Parse`.

```
void ProcessStream(generator<byte>& stream, FSM& parse)
{
  for(const auto& b : stream) {
    parse.SendSignal(b); A Send the new byte to the waiting Parse
        coroutine

    B Check whether we have a complete frame
    if(const auto& res = parse(); res.length()) { HandleFrame(res↩
        ); }
  }
}
```

Listing 2.16

In A we see each byte gets fed into `Parse`. After that, we check whether a frame is already complete by receiving the result of `p` by calling the call-operator. We know that the result is a `std::string`. With this knowledge, we can simply check whether the returned value has a length greater than zero. Once a frame is complete, `HandleFrame` is called in B. Equipped with this function, we can now process two different data-streams.

```
std::vector<byte> fakeBytes1{0x70_B,
                             ESC,
                             SOF,
                             ESC,
                             'H'_B,
                             'e'_B,
                             'l'_B,
                             'l'_B,
                             'o'_B,
                             ESC,
                             SOF,
                             0x7_B,
                             ESC,
                             SOF};

auto stream1 = sender(std::move(fakeBytes1)); Ⓐ Simulate the first
     network stream

auto p = Parse(); Ⓑ Create the Parse coroutine and store the handle in
     p

ProcessStream(stream1, p); Ⓒ Process the bytes

Ⓓ Simulate the reopening of the network stream
std::vector<byte> fakeBytes2{'W'_B, 'o'_B, 'r'_B, 'l'_B, 'd'_B, ↩
     ESC, SOF, 0x99_B};
auto stream2 = sender(std::move(fakeBytes2)); Ⓔ Simulate a second
     network stream

ProcessStream(stream2, p); Ⓕ We still use the former p and feed it
     with new bytes
```

Listing 2.17

With Ⓐ, we create the `sender` and pass the first bytes to it as a `std::vector`. This creates the first coroutine, which we store in `stream1`. Next, in Ⓑ, we create the `Parse` coroutine and save the handle to `Parse`'s generator in `p`. We pass `stream1` and `p` to `ProcessStream` for processing. This is the simulation of the first part of the network byte-stream. Let's suppose that at this point, the connection gets terminated.

In D, we reopen the stream by passing the second part of bytes to `sender` and store the coroutine in `stream2` as E shows. The interesting part now is F. We can reuse `p` our `Parse` coroutine. The coroutine still has the former state preserved and is able to continue where the coroutine was left before the simulated stream was disrupted. If you look closely, you can see that the disruption occurred, in fact, at a point where `p` already saw `ESC` + `SOF` and, with that, was in a frame.

2.6 A different strategy of the `Parse` generator

While all of what I showed so far is fine and works, remember that we are talking about C++? One reason for the burden of implementing the coroutine interface in `generator` and `promise_type_base` I gave you was implementation freedom. Wouldn't it be a shame if I showed you only one way to implement our byte-stream parser example? Yes, it would! So let's look at a slightly different way of how to implement the generator for `Parse`.

One thing that could look odd are the lines `co_await byte` in `Parse`. We know now that these lines work, but telling at a first glance where `co_await` gets its data from is hard. How about we have something down the line `co_await stream`? Have a look at the following alternative implementation of `Parse`:

```
FSM Parse(DataStreamReader& stream) A Pass the stream a parameter
{
 while(true) {
   byte b = co_await stream; B Await on the stream
   std::string frame{};

   if(ESC == b) {
     b = co_await stream;

     C not looking at a end or start sequence
     if(SOF != b) { continue; }

```

Listing 2.18

```
      Ⓓ capture the full frame
      while(true) {
        b = co_await stream;

        if(ESC == b) {
          Ⓔ skip this byte and look at the next one
          b = co_await stream;

          if(SOF == b) {
            co_yield frame;
            break;
          } else if(ESC != b) {
            Ⓕ out of sync
            break;
          }
        }

        frame += static_cast<char>(b);
      }
    }
  }
}
```

Listing 2.18

As we can see in Ⓐ, `Parse` in this implementation takes a parameter `DataStreamReader& stream`. This time by reference, which implies we have to ensure the lifetime of `stream` outlives that of `Parse`. In Ⓑ, we can see the variant of `co_await`. We now await on the stream. This makes the code a bit more clear where the data comes from, the `DataStreamReader`. What else do we have to change to make this code work?

Let's start with the return type of `Parse` `FSM`. The return type now uses `generator` instead of the `async_generator`.

```
using FSM = generator<std::string, false>;
```

Listing 2.19

This removes passing the type of the Awaiter, but there is another parameter `false`. So, `generator` has changed a little once again. Here is the altered `generator`:

```
template<typename T, bool IntialSuspend = true> Ⓐ New NTTP
struct generator {
  using promise_type =
    promise_type_base<T, generator, IntialSuspend>; Ⓑ Forward
        IntialSuspend

  using PromiseTypeHandle = std::coroutine_handle<promise_type>;
  using iterator = coro_iterator::iterator<promise_type>;

  iterator begin() { return {mCoroHdl}; }
  iterator end() { return {}; }

  generator(generator const&) = delete;
  generator(generator&& rhs)
  : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)}
  {}

  ~generator()
  {
    if(mCoroHdl) { mCoroHdl.destroy(); }
  }

  T operator()()
  {
    T tmp{};
    Ⓒ use swap for a potential move and defined cleared state
    std::swap(tmp, mCoroHdl.promise().mValue);

    return tmp;
  }

private:
```

Listing 2.20

```
  friend promise_type; (D) As the default ctor is private we G needs to
      be a friend
  explicit generator(promise_type* p)
  : mCoroHdl(PromiseTypeHandle::from_promise(*p))
  {}

protected:
  PromiseTypeHandle mCoroHdl;
};
```

Listing 2.20

The new template-parameter (A) is an NTTP called `InitialSuspend`. This parameter is directly forwarded to `promise_type_base`, as we can see in (B). Other than that, `generator` is the same as before. Now, let's have a look at what is new in `promise_type_base`. Here is the updated implementation:

```
template<typename T,
         typename G,
         bool InitialSuspend> (A) Control the initial suspend
struct promise_type_base {
  T mValue;

  std::suspend_always yield_value(T value)
  {
    mValue = value;
    return {};
  }
  auto initial_suspend()
  {
    if constexpr(InitialSuspend) { (B) Either suspend always or never
      return std::suspend_always{};
    } else {
      return std::suspend_never{};
    }
  }

  std::suspend_always final_suspend() noexcept { return {}; }
```

Listing 2.21

```cpp
  G get_return_object() { return G{this}; };
  void unhandled_exception() { std::terminate(); }
  void return_void() {}
};
```

Listing 2.21

As we can see, `promise_type_base` uses the template-parameter `InitialSuspend` to switch the return-type of `initial_suspend`. We can now control whether the coroutine suspends directly after creation or runs until a `co_yield`, `co_return`, or `co_await` statement is reached. The reason for the switch is that in the case of the generator for `Parse`, we like the coroutine to run to the first `co_nnn` statement, but in the case of `sender` where we use the range-based for-loop, we like to suspend the coroutine at creation and resume it when the range-based for-loop starts.

But where is the `co_await`-part? Remember that we had an `awaitable_promise_type_base` before? `await_transform` signaled the coroutine FSM what to do for a `co_await`. Until now, we haven't seen that part anymore. The reason is that in this approach, we split the generator. We now have a generator that only yields values, much like the implementation in our very first example `counter`. `DataStreamReader` implements the `co_await`-part. Here is this implementation:

```cpp
class DataStreamReader { Ⓐ Awaitable
public:
  DataStreamReader() = default;

  Ⓑ Using DesDeMovA to disable copy and move operations
  DataStreamReader& operator=(DataStreamReader&&) noexcept = ↩
      delete;

  struct Awaiter { Ⓒ Awaiter implementation
    Awaiter& operator=(Awaiter&&) noexcept = delete;
    Awaiter(DataStreamReader& event) noexcept
    : mEvent{event}
    {
      mEvent.mAwaiter = this;
    }

```

Listing 2.22

```
    bool await_ready() const noexcept { return mEvent.mData.↩
        has_value(); }

    void await_suspend(std::coroutine_handle<> coroHdl) noexcept
    {
      mCoroHdl = coroHdl; D Stash the handle of the awaiting coroutine.
    }

    byte await_resume() noexcept
    {
      assert(mEvent.mData.has_value());
      return *std::exchange(mEvent.mData, std::nullopt);
    }

    void resume() { mCoroHdl.resume(); }

  private:
    DataStreamReader& mEvent;
    std::coroutine_handle<> mCoroHdl{};
  };

  E Make DataStreamReader awaitable
  auto operator co_await() noexcept { return Awaiter{*this}; }

  void set(byte b)
  {
    mData.emplace(b);
    if(mAwaiter) { mAwaiter->resume(); }
  }

private:
  friend struct Awaiter;
  Awaiter* mAwaiter{};
  std::optional<byte> mData{};
};
```

Listing 2.22

Class `DataStreamReader` is the Awaitable in this implementation. That means that `DataStreamReader` provides the `Awaiter`-type. We can see in Ⓑ that `DataStreamReader` is neither copy- nor moveable. To save lines, I used Peter Sommerlad's approach called Destructor defined Deleted Move Assignment (DesDeMovA) [4] to get the behavior. The idea of his approach is to delete a single special member function, the move-assignment operator and by that all other special member functions for move and copy are implicitly deleted. Moving on, Ⓑ shows us the implementation of `Awaiter`. This type is also neither copy- nor moveable, and has a constructor that takes a reference to a `DataStreamReader`. The reason for this approach is to keep the `Awaiter`-type small in terms of data. The data we like to promote with `co_await` is stored in `DataStreamReader`.

Another difference is `await_suspend`. Here at Ⓓ, we stash the coroutine handle. Before the generator knew the handle, this time, `DataStreamReader` doesn't. All other parts of `Awaiter` are already known or nothing special.

The crucial piece is Ⓔ, the implementation of `operator co_await`. This makes `DataStreamReader` an Awaitable. Here we see another customization point where we can control what our type show does and how. Table 2.4 shows the three different ways to create an Awaitable. What we haven't discussed is providing a global `operator co_await`.

The using code of `Parse` changes slightly with the new implementation approach. Listing 2.23 shows this, this time without the byte's definition, they remain the same as before.

```
auto stream1 = sender(std::move(fakeBytes1));

DataStreamReader dr{}; Ⓐ Create a DataStreamReader Awaitable
auto p = Parse(dr); Ⓑ Create the Parse coroutine and pass
    the DataStreamReader

for(const auto& b : stream1) {
  dr.set(b); Ⓒ Send the new byte to the waiting DataStreamReader

  if(const auto& res = p(); res.length()) { HandleFrame(res); }
}

```

Listing 2.23

```
auto stream2 = sender(std::move(fakeBytes2)); D Simulate a
    second network stream

for(const auto& b : stream2) {
  dr.set(b); E We still use the former dr and p and feed it with new
      bytes

  if(const auto& res = p(); res.length()) { HandleFrame(res); }
}
```

Listing 2.23

At A, an object of the new type `DataStreamReader` is created and in B passed to `Parse`. We can see the difference in usage now at C. The bytes are now passed to the Awaitable `DataStreamReader`. The result, a complete frame, is still obtained by `Parse`, and with that, the variable `p`. Here we can see the separation we created with this implementation approach. The two parts are now decoupled. We can use the same `DataStreamReader` and pass it to another `Parse` or similar coroutine.

2.7 Using a coroutine with custom `new` / `delete`

The coroutines we looked at and used so far worked perfectly. The compiler did allocate the memory for them, for us, and we needed to call `destroy` on the coroutine-handle to bring the compiler into deallocating the memory again. But what if we like more fine-control? What if we like to provide a custom **`new`** and **`delete`** because our environment does not allow dynamic allocations from a global heap? Well, thanks to the customization points, this is easy. All we need to do is to provide the desired functions for our PromiseType. No magic required. Even for a PromiseType, the compiler follows the usual rules looking up an **`operator new`** in a class before going to the global **`operator new`**. Below you see an implementation of our former `promise_type_base`, which has the two operators.

```
template<typename T, typename G, bool InitialSuspend>
struct promise_type_base {
  T mValue;

```

Listing 2.24

```
  std::suspend_always yield_value(T value)
  {
    mValue = value;
    return {};
  }
  auto initial_suspend()
  {
    if constexpr(InitialSuspend) {
      return std::suspend_always{};
    } else {
      return std::suspend_never{};
    }
  }

  std::suspend_always final_suspend() noexcept { return {}; }
  G get_return_object() { return G{this}; };
  void unhandled_exception() { std::terminate(); }
  void return_void() {}

  Ⓐ Custom operator new
  void* operator new(size_t size) noexcept { return Allocate(size↩
      ); }

  Ⓑ Custom operator delete
  void operator delete(void* ptr, size_t size) { Deallocate(ptr, ↩
      size); }

  Ⓒ Allow new to be noexcept
  static auto get_return_object_on_allocation_failure() { return ↩
      G{nullptr}; }
};
```

Listing 2.24

Using a coroutine with a custom `new` is as simple as promised. We provide `operator new` for our PromiseType in Ⓐ, as well an `operator delete`. I made the choice to mark `operator new` as `noexcept` and therefore provide get_return_object_on_allocation_failure in C as a fallback mechanism for the compiler.

Everything else remains the same. No change of the using code is required. But there is more, right? We sometimes like to provide a custom allocator that should be used. The variant we have now still uses a single allocator.

2.8 Using a coroutine with a custom allocator

Suppose we like to use a custom allocator, which differs per invocation, but the PromiseType should remain the same. With that, we can get close to stackfull coroutines. Okay, maybe we have a little more work to do than we should for real stackfull coroutines, but it is doable. Once again, for this task, we have to update promise_type_base. We can provide an operator new template, which then picks the right allocator. Below is an implementation of promise_type_base doing that.

```cpp
template<typename T, typename G, bool InitialSuspend>
struct promise_type_base {
  T mValue;

  std::suspend_always yield_value(T value)
  {
    mValue = value;
    return {};
  }

  auto initial_suspend()
  {
    if constexpr(InitialSuspend) {
      return std::suspend_always{};
    } else {
      return std::suspend_never{};
    }
  }

```

Listing 2.25

```
  std::suspend_always final_suspend() noexcept { return {}; }
  G get_return_object() { return G{this}; };
  void unhandled_exception() { std::terminate(); }
  void return_void() {}

  A Custom operator new
  template<typename... TheRest>
  void* operator new(size_t size, arena& a, TheRest&&...) ↩
      noexcept
  {
    return a.Allocate(size);
  }

  B Custom operator delete
  void operator delete(void* ptr, size_t size)
  {
    arena::GetFromPtr(ptr, size)->Deallocate(ptr, size);
  }

  static auto get_return_object_on_allocation_failure() { return ↩
      G{nullptr}; }
};
```

Listing 2.25

As you can see, this time A is a template. If you like, the trick we employ here is that the second parameter to `operator new` is of the type of our allocator. The template is there for the possible remaining parameters. This `new` gets called with exactly the parameter types and order as we call our coroutine during setup. We can use the reference to `arena` to call `Allocate` on that arena.

Arena

The term arena refers to a large, contiguous piece of memory, often an `unsigned char` array, that is allocated only once and then used to do allocations using that already-allocated memory. Arenas are usually pre-allocated during start-up in form of an array. They can often be found in time-critical systems where using the global heap can cause timing differences or when we like to avoid out-of-memory situations for a subcomponent and allocate all the memory that the subcomponent requires to run during start-up.

The `operator delete` part is a bit more tricky. At this point, we don't have a reference to `arena` handy. The way we can free the memory in `operator delete` is to store a pointer to the original arena during `Allocate` in a hidden memory part after the data. Which is exactly what `Allocate` does for us as well. In `delete`, we know the size of a memory block as well, because we use a `delete` overload which has `size` as second parameter. This allows us to jump to the correct offset and retrieve the pointer to the matching arena, and use that to call `Deallocate`.

```
arena a1{};
arena a2{};

A Pass the arena to sender
auto stream1 = sender(a1, std::move(fakeBytes1));

DataStreamReader dr{}; B Create a DataStreamReader Awaitable
auto p = Parse(a2, dr); C Create the Parse coroutine and pass
     the DataStreamReader

for(const auto& b : stream1) {
  dr.set(b); D Send the new byte to the waiting DataStreamReader

  if(const auto& res = p(); res.length()) { HandleFrame(res); }
}

auto stream2 =
  sender(a1, std::move(fakeBytes2)); E Simulate a
       second network stream

for(const auto& b : stream2) {
  dr.set(b); F We still use the former dr and p and feed it with new
       bytes

  if(const auto& res = p(); res.length()) { HandleFrame(res); }
}
```

Listing 2.26

The using part changes slightly. We now need to pass an arena to `sender` and `Parse`, as you can in A and C. Aside from this, the rest of the code doesn't know anything about the arena and can be used as before.

2.9 Exceptions in coroutines

So far, we looked at the happy path of coroutines, which means that we ignored exceptions. However, as exceptions are one of the pillars of C++, we cannot ignore them, not even in coroutines. We already saw a customization point for exceptions in the form of the PromiseType's function `void unhandled_exception()`.

This customization point allows us to control the behavior of a coroutine in the event of an exception. There are two different stages where an exception can occur:

1. During the setup of the coroutine, i.e., when the `PromiseType` and `Generator` are created.
2. After the coroutine is set up and about to or already runs.

The two stages are fundamentally different. In the first stage, our PromiseType and Generator are not completely set up. An exception that occurs during that stage is directly passed to our calling code. To catch any exception, let's first add a `try-catch` block around the heart of our parser:

```
try { A Wrapp it in a try-catch block
  auto stream1 = sender(std::move(fakeBytes1));

  DataStreamReader dr{};
  auto p = Parse(dr);

  for(const auto& b : stream1) {
    dr.set(b);

    if(const auto& res = p(); res.length()) { HandleFrame(res); }
  }
} catch(std::runtime_error& rt) { B Listen for a runtime error
  PrintException(rt);
```

Listing 2.27

```
}
```

As you can see, the implementation is a slightly down-stripped version. At this point, we do not care about the simulated reconnect of the stream. Other than that, all that code is now wrapped in a `try`-`catch` block which acts on an `std::runtime_error` exception. This is freely chosen and the one I will throw in the following examples. Of course, other exceptions are possible as well.

Now should an exception occur in the first stage, we end up directly in B. As with every other exception, all objects already allocated in the `try`-block are destroyed. Our coroutine is unusable at this point. We are at this stage roughly as long as the evaluation of `initial_suspend` is not finished. After that, we are in stage 2.

Option 1: Let it crash

Now the customization point `unhandled_exception` comes to play. Each exception that occurs in our coroutine's body or in the other customization points of the PromiseType will cause a call to `unhandled_exception`. Here we can decide what to do. The codeless approach is to leave `unhandled_exception` empty. If `unhandled_exception` returns to our coroutine `FSM`, the compiler shuts down the coroutine by calling `final_suspend`. This call to `final_suspend` is outside the compiler-generated `try`-`catch` block, which is the reason why `final_suspend` must be marked as `noexcept`. To recap, leaving `unhandled_exception` masks the exception, and your program will likely crash shortly after.

Option 2: Controlled termination

This leads us to a second possible implementation of `unhandled_exception`, call either `std::terminate` or `abort` in the body. This terminates the program effectively and gives you a chance to set a break-point with a debugger to see the call stack.

Option 3: Re-throw the exception

A third approach is the re-throw the exception in the body of `unhandled_exception` . That way, the exception reaches the outer `try`-`catch` block, which we added in our former example. This allows us to deal with the exception outside of the coroutine.

As before, the coroutine is still unusable, and all objects get destroyed, but we have a chance for a re-run with different input values or so.

Of course, we can also do more things in `unhandled_exception`, regardless of which of the options we implement. We can always write a dedicated log message or similar things, and do whatever suits our environment best afterward. This is the freedom we have, thanks to the customization points.

Cliff notes

- Be careful when passing parameters to a coroutine. Parameters with `const &` do not get their life-time extended in the coroutine frame. In such a case, you must ensure that the data lives longer than the coroutine or use copy semantics in coroutine parameters.
- Coroutines in C++ are stackless coroutines. The coroutine frame is stored on the heap.
- The heap allocation for the coroutine frame is handled by the compiler for us.
- We have a new operator `operator co_await`, which is called by `co_await`.
- The customization points allow us a very flexible coroutine implementation.
- A `PromiseType` with an empty `unhandled_exception` will crash uncontrolledly.
- Using a coroutine with a custom allocator is possible.
- Most likely, C++23 will bring us a coroutine STL with pre-defined generators and Promise-Types.
- Calling `destroy()` on a non-suspended coroutine is UB.

Table 2.2: Coroutine customization points

Name	Required	Type	Returns
`get_return_object_on_allocation_failure()`		Ps	Generator
`get_return_object()`	y	P	Generator
`initial_suspend()`	y	P	`std::suspend_always` / `std::suspend_never`
`final_suspend() noexcept`	y	P	`std::suspend_always` / `std::suspend_never`
`unhandled_exception()`	y	P	`void`
`return_void()` / `return_value(T)`	y	P	`void`
`yield_value(T)`	y [a]	P	`std::suspend_always` / `std::suspend_never`
`await_transform(T)`		P	`Awaiter`
`T::operator co_await()`		Co	`Awaiter`
`operator co_await(T)`		Go	`Awaiter`
`operator new(size_t)`		Pg	`void*`
`operator new(size_t, std::nothrow)`		Pg [b]	`void*`
`template<typename... Ts> operator new(size_t, Ts...)`		P	`void*`
`operator delete(void*, size_t)`		Pg	`void`
`await_ready()`	y	A	`bool`
`await_suspend(std::coroutine_handle<>)`	y	A	`bool` / `void`
`await_resume()` [c]	y	A	`bool` / `void`

P = Promise
Ps = static in Promise
Pg = Promise or global
A = Awaitable
Go = Global operator for T
Co = Class operator for T
[a] For `co_yield`.
[b] Used when `get_return_object_on_allocation_failure()` is present.
[c] Marking it as `noexcept` changes code generation.

Table 2.3: Coroutine Awaiter interface

Method	As type
`bool await_ready()`	Signal whether the Awaiter already has data, if not, coroutine gets suspended.
`void await_suspend(coroutine_handle<>)`	Called when the coroutine is about to be suspended. The handle allows the Awaiter to wake the coroutine up later, with `void` suspended unconditionally.
`bool await_suspend(coroutine_handle<>)`	As above, but suspension depends on the return value.
`T await_resume()`	Obtain the operations result before resuming the coroutine. Note, `T` can be `void`.

Table 2.4: co await operator

Keyword	Action	Type
`T::operator co_await()`	Class operator for T	Stateful yield
`operator co_await(T)`	Global operator for T	Stateless yield
`auto await_transform(T)`	`promise_type`	Yield and await

Chapter 3

std::ranges

Chapter 4

Modules

Chapter 5

std::format: Modern & type-safe text formatting

Text formatting, typically, is often an essential part of programming. An example use-case is the localization of textual User Interfaces (UIs), where localization means translating words from one language to another and changing symbols such as the decimal separator. Often we have repetitions in our format arguments, such as multiple currency values with the same currency. Other use-cases are formatting log or debug messages.

For this chapter, let's pretend we are working in a finance-related job and therefore have to format stock-index information. First, let us look at the options that we had before C++20.

5.1 Formatting a string before C++20

String formatting before C++20 meant either using iostreams or `snprintf` or a library, the latter not really being C++ish. Both have their pros and cons, which we will see in this section. Suppose we are about to create one of these fancy stock market banners showing the different stock-index values and their changes, as are shown by all news channels.

We will use three different indices: DAX, Dow and S&P 500. The output contains the current stock index points, the delta points to yesterday and the delta in percent. Each index is printed on a dedicated line and all columns are perfectly aligned. This is an example of the desired output:

```
DAX          13108.50   55.55  0.43%
Dow          29290.00  209.83  0.72%
S&P 500       3561.50   24.49  0.69%
```

Before we start with the formatting, we need data to format. Therefore we create a `StockIndex` class which stores

- the name of the index,
- the last points,
- the current points.

Whenever we set the current points via `setPoints`, the methods automatically updates last points. Other than that, `StockIndex` has some access functions:

- `setPoints` as already said, update the points;
- `points` returns the current points;
- `pointsDiff` returns the difference between last and now in points;
- `pointsPercent` returns the difference between last and now in percent.

A possible implementation is given below.

```cpp
class StockIndex {
  std::string mName{};
  double mLastPoints{};
  double mPoints{};

public:
  StockIndex(std::string name)
  : mName{name}
  {}

```

Listing 5.1

```cpp
  std::string name() const { return mName; }

  void setPoints(double points)
  {
    mLastPoints = mPoints;
    mPoints = points;
  }

  double points() const { return mPoints; }

  double pointsDiff() const { return mPoints - mLastPoints; }

  double pointsPercent() const
  {
    if(0.0 == mLastPoints) { return 0.0; }
    return (mPoints - mLastPoints) / mLastPoints * 100.0;
  }
};
```

Listing 5.1

Equipped with this class, we are ready to create some stock indices and fill them with values that we can then format. For the stock-index creation, we implement a method, `GetIndices`. This enables us to use the same stock indices with different formatting methods.

```cpp
std::vector<StockIndex> GetIndices()
{
  StockIndex dax{"DAX"};
  dax.setPoints(13'052.95);
  dax.setPoints(13'108.50);

  StockIndex dow{"Dow"};
  dow.setPoints(29'080.17);
  dow.setPoints(29'290.00);

  StockIndex sp{"S&P 500"};
  sp.setPoints(3'537.01);
```

Listing 5.2

```
  sp.setPoints(3'561.50);

  return {dax, dow, sp};
}
```

Listing 5.2

Excellent, now we have all in place and are ready to format the data. As this is a C++ book, let's start with the C++ way, using iostreams.

5.1 C++14: A digit separator

The digit separator ' was introduced with C++14. It allows us to make numbers more readable for us humans. The digit separator has no influence on how the compiler sees or treats a number. We are also free on where to place the separator:

```
auto x = 2'000'000; Ⓐ The probably usual digit separation
auto y = 2'00'00'00; Ⓑ But we are free
```

Listing 5.3

5.1.1 Formatting a stock index with `iostreams`

Piece of cake, right? All the data is there. We just need to pass it to `std::cout`. Nice and simple, as string formatting should be. The code then is this:

```
for(const auto& index : GetIndices()) {
  std::cout << index.name() << " " << index.points() << " "
            << index.pointsDiff() << " " << index.pointsPercent()
            << '%' << '\n';
}
```

Listing 5.4

But wait, how does the output look if we execute the program (`a.out`) in a terminal? Not quite as it was supposed to be:

```
$ ./a.out
DAX 13108.5  55.55  0.425574%
Dow 29290  209.83  0.721557%
S&P 500 3561.5  24.49  0.692393%
```

How bad is the result? Well, it depends. How many differences with the desired formatting do you see?

First, the names are not aligned. S&P 500 is longer than DAX or Dow. Second, the points of Dow are without any decimal places. All other points show only one, but two were desired, except the percent difference. If you like, we can say that the percent difference has enough decimal places to cover the missing ones. They come with six decimal places each. Sadly four more than desired. Finally, the alignment of each column is broken as well. But we all are experienced iostreams-users, right? To be honest, I'm more in the `printf` camp, and have to look up all the specialties of iostreams all the time. Here is a version which does the formatting as desired:

```
for(const auto& index : GetIndices()) {
  std::cout << std::fixed;
  std::cout << std::setprecision(2); Ⓐ We need <iomanip> for this

  std::cout << std::setw(10) << std::left << index.name() << " "
           << std::setw(8) << std::right << index.points() << " "
           << std::setw(6) << index.pointsDiff() << " "
           << index.pointsPercent() << '%' << '\n';
}
```

Listing 5.5

As you can see, we have to set `std::fixed` and then `setprecision` to 2 for all that follows. The latter one requires an additional header, as Ⓐ shows. Then, to format the name, we need `std::setw` and `std::left`. Wait, what does `std::setw` do? It sets the desired width, of course. Why `std:setw` and not `std::setwidth`, you ask? Well, because the standard says so and we used `std::setw` for years. To align the numbers, we use `std::right`, and we get the desired output. Ok, granted, we need to know something about formatting to do it.

Whether function names are the best is always up for discussion. But do you think that this formatting is readable? Can you or one of your colleagues quickly spot what the output of that iostream statement will be? Not the actual numbers, of course, just roughly what will be printed. At least I can't. For me, seeing what we are formatting here and how the result should look like is very hard. Interesting function-name choices aside.

Oh yes, and the experts among us know that we just tampered with `std::cout`. All following calls to `std::cout` will also use only two decimal places. Probably not what they want. At this point, I don't like to get into the details on how to first backup and then restore the `std::cout` configuration. Let's just say some additional work is necessary and that missing the backup and restore details can cause you interesting trouble. The fun gets even better if you do not always write to `std::cout`, but sometimes to a different iostream. That can result in hiding the backup/restore need for a while and makes debugging and errors due to the lack of back/restore even more complicated.

There is more. What if we like to do this formatted printing multiple times in our project? Sure we can create a named function, and everyone uses this. But iostream is much better here. We can provide our own overload for `operator<<`. This way, we can pass an object of `StockIndex` to `std::cout`, and the `operator<<` takes care of consistent formatting. With that, users do not have to remember a formatting function, which I think is a great thing. The implementation is very much like what we had before, except that this time, the range-based for-loop body is moved into the `operator<<`. Due to the fact that the stream that this operator is supposed to write to is provided as an argument to the `operator<<` we need to change `std::cout` to the name of the out stream parameter, this is `os` in the code below. Here is such an implementation:

```cpp
std::ostream& operator<<(std::ostream& os, const StockIndex& ↩
    index)
{
  os << std::fixed;
  os << std::setprecision(2);

  os << std::setw(10) << std::left << index.name() << "  "
     << std::setw(8) << std::right << index.points() << "  "
     << std::setw(6) << index.pointsDiff() << "  "
     << index.pointsPercent() << '%' << '\n';

  return os;
}

```

Listing 5.6

```cpp
void WithIostreamOperator()
{
  for(const auto& index : GetIndices()) { std::cout << index; }
}
```

Listing 5.6

Localized formatting

Now, what if we turn the heat up once more. Say we like to have a locale-dependent decimal-place separator? We like to have the German notation. The decimal separator in German is a comma instead of a period. To achieve this, we need to set the locale of the iostream in question like in A in the following code:

```cpp
for(const auto& index : GetIndices()) {
  std::cout << std::fixed;
  std::cout << std::setprecision(2);
  std::cout.imbue(
    std::locale("de_DE.UTF-8")); A Apply the desired locale

  std::cout << std::setw(10) << std::left << index.name() << "  "
            << std::setw(8) << std::right << index.points() << "  "
            << std::setw(6) << index.pointsDiff() << "  "
            << index.pointsPercent() << '%' << '\n';
}
```

Listing 5.7

If we are changing the locale, that means changing the locale for the entire stream. We need to backup and restore the locale if necessary. A further design choice is whether to wrap the locale change in the `operator<<` or leave it to the user to change the locale of the stream before calling `operator<<`.

Formatting with iostreams - a summary

To summarize the iostreams experience: some of this formatting approach requires a good pair of eyes and some knowledge about which functions to call. Having `operator<<` is also great for consistent formatting. However, as soon as localization gets in, things get a bit more complicated.

One thing we haven't talked about directly, but seen all the time, is that iostream comes with a great benefit. Iostreams are type-safe. We cannot accidentally print a string as an `int`. This is the case where we didn't need to provide a format string. However, a format string would give us a better idea of how the output looks like. Which directly leads us to `printf`.

5.1.2 Formatting a stock index with `printf`

We stick with our example of printing stock indices. The desired output will be the same as before. Here is an implementation using `printf`:

```
for(const auto& index : GetIndices()) {
  printf("%-10s %8.2lf %6.2lf %4.2lf%%\n",
         index.name().c_str(),
         index.points(),
         index.pointsDiff(),
         index.pointsPercent());
}
```

Listing 5.8

The main difference is that we have a so-called formatting string, which is the first parameter of `printf`, followed by the values we like to be printed. I personally can read that formatting string way better than the iostream version before. All the formatting is in one place. While the actual values are unknown, we can see that the first parameter is left-aligned with a width of 10. Then follow three floating-point numbers, each with two decimal places and different widths. Yes, to see that, we need to know what `s` or `lf` stands for. Oh, and we have to escape the percent sign, as this character is used to signal a format argument. That is the reason for the two percent signs at the end. Write two, get one. We can conclude that different knowledge about the format arguments is required compared to iostreams.

But `printf` comes with another advantage, that is, atomicity. `printf` is a variadic function and takes all its arguments in a single call. That way, we can write them easier atomically without a risk of interleaved output, as we have with iostream. The POSIX version of `printf` mandates that the operation must be thread-safe. With iostreams, this isn't possible, because we are looking at multiple invocations of `operator<<`. Sadly, `printf` is an ancient C function using `varargs` to manage the variadic arguments. The format string and the format arguments there are used to cast the arguments to the type encoded in the format argument. No checks are possible to determine whether the argument matches the format argument. The missing type-safety or checking makes accidental printing of a string as an `int` very easy.

Formatting with `printf` - a summary

To recap, iostream has the advantage of type-safety while `printf` provides a readable format string and is only a single function call. Let's get totally crazy and say we combine these two approaches! How devious! But this exactly what Victor Zverovich did. He is the author of fmtlib and mastermind behind the `std::format` in the C++ standard.

5.2 Formatting a string using `std::format`

C++20, in fact, brings us the combination of iostream and `printf` in the form of `std::format`. As the name implies, `std::format` is a pure formatting facility. Instead of using `varargs`, `std::format` is a variadic template. Like `printf`, `std::format` takes a format string as the first argument. This is a `std:string_view` for best performance. The format string is a little like `printf`, but see for yourself. Below you see the stock index printing using `std::format`.

```cpp
for(const auto& index : GetIndices()) {
  std::cout << std::format("{:10} {:>8.2f} {:>6.2f} {:.2f}%\n",
                           index.name(),
                           index.points(),
                           index.pointsDiff(),
                           index.pointsPercent());
}
```

Listing 5.9

As we can see in this example, `std::format` uses curly braces to specify format arguments. Because `std::format` is a variadic template, the type of each argument is preserved. This allows us, in the most simplest case, to just write an open and closing curly brace. The library deduces the type and applies the default formatting for the type. The defaults are listed in Table 5.1. Automatic type deduction and formatting

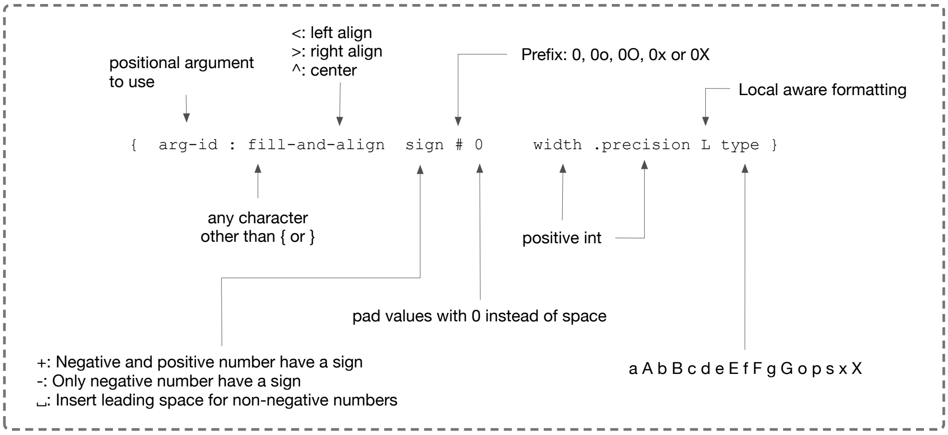

positional argument to use
<: left align
>: right align
^: center
Prefix: 0, 0o, 0O, 0x or 0X
Local aware formatting
{ arg-id : fill-and-align sign # 0 width .precision L type }
any character other than { or }
positive int
pad values with 0 instead of space
+: Negative and positive number have a sign
-: Only negative number have a sign
␣: Insert leading space for non-negative numbers
a A b B c d e E f F g G o p s x X

Figure 5.1: Standard format specifiers of `std::format`.

is the advantage we know from iostream. In our case, we need to specify the width of the string, which we do by `:10` inside the curly braces. We still omit the type and let the library deduce it.

For the following arguments, we need to specify the type, as we need to specify the decimal places along with the width. We can also see there how the alignment is controlled by `>`. What other options does `std::format` provide for us?

5.2.1 `std::format` specifiers

We already established that `std::format` uses curly braces to specify format arguments. In these braces, we can specify the type we expect and the format for it. Several samples are already shown in the last code example above. Should we lie to the system when explicitly naming a type, we get a `format_error` exception. We can further control the alignment, padding, the padding character, and whether the formatting uses the current system locale. Figure 5.1 gives a detailed overview of the options and their order. We will cover the options in the following sections.

By default, `std::format` comes with 17 different format specifiers, giving us full control over the desired output. We now have an option to print the binary representation of an integer. Floating-point numbers allow us a variety of different output formats, such as scientific or fixed. A full list, including the available prefixes, is provided in Table 5.1.

Table 5.1: Supported `std::format` specifiers

Type	Prefix	Meaning	Optional
`b`	`0b`	Binary representation	
`B`	`0B`	Binary representation	
`c`		Single character	x
`d`		Integer or char	x
`o`	`0`	Octal representation	
`x`	`0x`	Hexadecimal representation	
`X`	`0X`	Same as `x`, but with upper case letters	
`s`		Copy string to output, or `true`/`false` for a `bool`	x
`a`	`0x`	Print float as hexadecimal representation	
`A`	`0X`	Same as `a`, but with upper case letters	
`e`		Print float in scientific format with precision of 6 as default	
`E`		Same as `e`, just the exponent is indicated with `E`	
`f`		Fixed formatting of a float with precision of 6	
`F`		Same as `f`, just the exponent is indicated with `E`	
`g`		Standard formatting of a float with precision of 6	x
`G`		Same as `g`, just the exponent is indicated with `E`	
`p`	`0x`	Pointer address as hexadecimal representation	x

5.2.2 Escaping

With a format string, we look at a situation where we want the format string to contain a character that is the marker of a format specifier. For example, with `printf`, we have to escape

```cpp
std::format("Having the }} in a {}.", "string");
```

Listing 5.10

5.2.3 Localization

By default, all formatting of `std::format` is locale-independent. That means that the digit separator of a `float` does not change by changing the system's locale. The same is true for the thousands separator.

Should we want a certain argument to be printed in a localized format, we have to add `L` to the format specifier before the type. That way, the system's current locale is used to format the format argument. Should we need to change the locale only for one formatting and this should be a certain locale, we can use one of the `std::format` overloads, which takes a `std::locale` as the first argument.

Below we have a short program that uses the different ways to apply localization to the double π and the integer 1.024.

```
double pi = 3.14;
int i = 1'024;

auto us = "en_US.UTF-8"s;
auto locDE = std::locale("de_DE.UTF-8"); Ⓐ Create a German locale
auto locUS = std::locale(us); Ⓑ Create a US locale

std::cout << "double with format(loc, ...)\n";
std::cout << std::format(locUS, "  in US: {:L}\n", pi);
std::cout << std::format(locDE, "  in DE: {:L}\n", pi);

std::cout << "\nint with format(loc, ...)\n";
std::cout << std::format(locUS, "1'024 in US: {:L}\n", i);
std::cout << std::format(locDE, "1'024 in DE: {:L}\n", i);

Ⓒ Simulate a different system locale
std::locale::global(std::locale(us));
std::cout << "\nint with format(...) after setting global loc\n"↩
    ;
std::cout << std::format("1'024 in US: {:L}\n", i);
```

Listing 5.11

In Ⓐ and Ⓑ, we create two locales, one for Germany and one for US. These two locales are then used together with `std::format` and `:L` to format π and the integer. After that, in Ⓒ, a different system locale is simulated, and the integer is printed again.

This time with `std::format` without a dedicated locale. The resulting output of this program is the following:

```
$ ./a.out
double with format(loc, ...)
 in US: 3.14
 in DE: 3,14

int with format(loc, ...)
1'024 in US: 1,024
1'024 in DE: 1024

int with format(...) after setting global loc
1'024 in US: 1,024
```

To summarize, without `L`, the output of `std::format` is locale-independent. With `L` plain, `std::format` uses the system locale. For creating a specific localized format, we can set up a locale with `std::local` and pass this locale as the first argument to `std::format`.

5.2.4 Formatting floating-point numbers

The default formatting for floating-point numbers with `std::format` is interesting. The idea is to keep the provided value, but show only the required parts. What does that mean? For example, if you have a floating-point number that has trailing zeros after the decimal separator, they get cut. However, if there is only one after the decimal point, the following zero is kept. Here is some code that illustrates my words:

```cpp
double pi = 3.1400;
double num = 2.0;

std::string s = std::format("pi {}, num {}", pi, num);
```

Listing 5.12

Below you see the full output:

```
$ ./a.out
pi 3.14, num 2.0
```

As we can see, the output of the numbers is then 3.14 and 2.0. Exactly what we provided with no additional zeros added after the separator.

5.3 Formatting a custom type

Let's explore the power of `std::format` further. The `std::format` example lacks reusability. For iostream, we can provide an overload for our type of `operator<<`. `printf` does allow to register custom convert functions in the GNU libc, but that is somewhat clumsy. What does `std::format` provide for us there?

The good news is, `std::format` has a well-defined API for custom formatters, much like iostream. As `std::format` has no stream operator, there is a struct template `std::formatter<T>` which we need to specialize for our type. This type provides two methods we must provide:

- `constexpr auto parse(format_parse_context& ctx)`
- `auto format(const T& t, format_context& ctx)`

5.3.1 Writing a custom formatter

The first one, `parse`, allows us to parse the format specifier. For now, let's say that we don't like to do anything special there.

The second method, `format`, does the actual formatting of the data. For a first attempt, we simply move our existing `std::format` line into this method. In `format`, we then use `std::format_to` to format the data into the existing `format_context` `ctx`. In code, we have this:

```
template<>
struct std::formatter<StockIndex> {
  constexpr auto parse(format_parse_context& ctx) { return ctx.↩
      begin(); }

  auto format(const StockIndex& index, format_context& ctx)
  {
    return std::format_to(ctx.out(),
                          "{:10} {:>8.2f} {:>6.2f} {:.2f}%",
```

Listing 5.13

```
                        index.name(),
                        index.points(),
                        index.pointsDiff(),
                        index.pointsPercent());
  }
};
```

Listing 5.13

All we have to do now is to pass empty curly braces and a `StockIndex` object to `std::format`. Very much like iostream before, we have now a reusable and consistent way of formatting our stock-index data:

```
for(const auto& index : GetIndices()) {
  std::cout << std::format("{}\n", index);
}
```

Listing 5.14

Now, are you curious whether we can do better than with iostream? As an observant reader, you have spotted the `parse` method I did not get into so far.

5.3.2 Parsing a custom format specifier

The `parse` method gives us great fine control of the desired output. For this exercise, suppose we want more than just printing all the information in a stock index. We like to have the following options:

- Printing the entire information as before, using empty curly braces.
- Printing the entire information as before, but with a plus sign in front of the points and percentage difference, using `p` as a format specifier.
- Printing the name of the index and the current points, using `s` as a format specifier.

By parsing the format specifier provided as `format_parse_context` in `parse`, we can change the behavior of `std::format` based on this information.

```
template<>
struct std::formatter<StockIndex> {
  enum class IndexFormat { Normal, Short, WithPlus };
```

Listing 5.15

```
IndexFormat indexFormat{IndexFormat::Normal};

constexpr auto parse(format_parse_context& ctx)
{
  auto it = ctx.begin();
  auto end = ctx.end();

  if((it != end) && (*it == 's')) {
    indexFormat = IndexFormat::Short;
    ++it;
  } else if((it != end) && (*it == 'p')) {
    indexFormat = IndexFormat::WithPlus;
    ++it;
  }

  A Check if reached the end of the range
  if(it != end && *it != '}') { throw format_error("invalid ↩
      format"); }

  B Return an iterator past the end of the parsed range
  return it;
}

auto format(const StockIndex& index, format_context& ctx)
{
  if(IndexFormat::Short == indexFormat) {
    return std::format_to(
      ctx.out(), "{:10} {:>8.2f}", index.name(), index.points())↩
          ;

  } else {
    const std::string fmt{(IndexFormat::WithPlus == indexFormat)
                          ? "{:10} {:>8.2f} {: >+7.2f} {:+.2f}%"
                          : "{:10} {:>8.2f} {:>6.2f} {:.2f}%"};

```

Listing 5.15

```
      return std::format_to(ctx.out(),
                            fmt,
                            index.name(),
                            index.points(),
                            index.pointsDiff(),
                            index.pointsPercent());
    }
  }
};
```

Listing 5.15

Because the last case, s, differs in the number of arguments, we need a dedicated call to `std::format_to` there, which we see in the `if`-branch. Then, in the `else`-branch we deal with the empty curly braces and p. These two formats differ only in the presence or absence of the plus sign. The plus sign is why we need 7 instead of 6 characters for the p format. We keep the code short by introducing a variable `fmt` holding the format string. `fmt` contains either the default format or the one with the plus sign.

And here is how we can use our custom formatter for `StockIndex` with `std::format`:

```
for(const auto& index : GetIndices()) {
  std::cout << std::format("{}\n", index);
}

for(const auto& index : GetIndices()) {
  std::cout << std::format("{:s}\n", index);
}

for(const auto& index : GetIndices()) {
  std::cout << std::format("{:p}\n", index);
}
```

Listing 5.16

In my opinion, this is so much better than tampering via ominous functions with the iostream, risking to corrupt the stream if we do not reset the configuration.

Re-adding the L option to obtain the current system locale to format the floating-point number is possible as well:

```cpp
template<>
struct std::formatter<StockIndex> {
  enum class IndexFormat { Normal, Short, WithPlus };

  IndexFormat indexFormat{IndexFormat::Normal};

  bool localized =
    false; Ⓐ New member to track whether the formatting is localized

  constexpr auto parse(format_parse_context& ctx)
  {
    auto it = ctx.begin();

    auto isChar = [&](char c) { Ⓑ Helper to search for a character
      if((it != ctx.end()) && (*it == c)) {
        ++it;
        return true;
      }

      return false;
    };

    if(isChar('L')) { localized = true; } Ⓒ Localized formatting

    if(isChar('s')) {
      indexFormat = IndexFormat::Short;
    } else if(isChar('p')) {
      indexFormat = IndexFormat::WithPlus;
    }

    if(it != ctx.end() && *it != '}') { throw format_error("↩
        invalid format"); }

    return it;
  }

```

Listing 5.17

```
  auto format(const StockIndex& index, format_context& ctx)
  {
    const std::string locFloat{localized ? "L" : ""}; D Add localized
    const std::string plus{(IndexFormat::WithPlus == indexFormat) ? "+" : ""};

    if(IndexFormat::Short == indexFormat) {
      const auto fmt = std::format("{{:10}} {{:>8.2{}f}}", locFloat);
      return std::format_to(ctx.out(), fmt, index.name(), index.points());

    } else {
      const auto fmt{
        std::format("{{:10}} {{:>8.2{0}f}} {{:>{1}7.2{0}f}} {{:{1}.2{0}f}}%",
                    locFloat,
                    plus)};

      return std::format_to(ctx.out(),
                            fmt,
                            index.name(),
                            index.points(),
                            index.pointsDiff(),
                            index.pointsPercent());
    }
  }
};
```

Listing 5.17

Writing the custom formatter is still straightforward, just this time a bit more code. As the approach is to generate different format strings and simply call `std::format_to` for all the different combinations, we need to cover all that.

We can now invoke our custom formatter with `L` for local dependent formatting, or `p` to have a plus sign before the number, and `s` controls if the format is short.

```
for(const auto& index : GetIndices()) {
  std::cout << std::format("{:Ls}\n", index);
}

for(const auto& index : GetIndices()) {
  std::cout << std::format("{:Lp}\n", index);
}
```

Listing 5.18

5.4 Referring to a format argument

For the next element of `std::format`, suppose we have a couple of shares we need to format. Each share has a name, a current price, and a price delta to the last report. All prices are in EURO. As before, we like to have a share per line. The output then should look like the following:

```
Apple €       119.26 €  +23.40
Alphabet €   1777.02 €  -10.21
Facebook €    276.95 €   +5.32
Tesla €       408.50 €  -31.50
```

These are also the data we will use. We only need to alter our previous version slightly and end up with this:

```
for(const auto& share : GetShares()) {
  std::cout << std::format("{:10} {:>8.2f€} {:>+8.2f€}\n",
                           share.name(),
                           share.price(),
                           share.priceDelta());
}
```

Listing 5.19

I know that the task is trivial. However, one thing often bothered me in the past, the duplication of the EURO sign. Luckily `std::format` takes care of this as well. Did you notice that I never told you why the format specifier starts with a colon? What can be before the colon? An argument index is the answer. We are no longer bound to provide the arguments in the same order as the format string lists them.

We can mix and, more importantly, reuse them! Have a look at how that changes our former implementation:

```cpp
for(const auto& share : GetShares()) {
  std::cout << std::format("{1:10} {2:>8.2f}{0} {3:>+8.2f}{0}\n",
                           "€",
                           share.name(),
                           share.price(),
                           share.priceDelta());
}
```

Listing 5.20

We can see above that I reused the argument as index 0, the EURO sign, for the price as well as the price delta. I also made the EURO sign the first argument and added the matching indices to the other format specifiers.

Now, as soon as we start using an argument index for only one of the format specifiers, we need to provide argument indices for all the format specifiers. It makes sense, I think, we're telling the library to do something out of order. It is only fair to be responsible for the full order.

5.5 Using a custom buffer

With both iostream and `printf`, one option was to write a formatted string into an existing buffer. There were special types like `std::stringstream` or `snprintf`, which do the job for these formatting options.

5.5.1 Formatting into a dynamically sized buffer

With `std::format`, there is a `std::format_to` which allows us to format a string directly into a dynamic buffer, for example, a `std::vector`:

```cpp
std::vector<char> buffer{};
std::format_to(std::back_inserter(buffer), "{}, {}", "Hello", "↩
    World");

for(const auto& c : buffer) { std::cout << c; }
```

Listing 5.21

```

std::cout << '\n';
```

Listing 5.21

The bad part about this way with a dynamic buffer like `std::vector` is that we get a new allocation for each character. Due to the design of `std::vector`, the existing data has to be copied over, and the former memory is freed. When formatting a large string like this, the performance penalty will come to the surface. But there is `std::formatted_size` to rescue. This function returns the number of characters the resulting string will take. Of course, without requiring allocations during the inspection. The former example now becomes this:

```
const std::string fmt{"{}, {}"}; Ⓐ The format string

Ⓑ Lookahead the resulting size in bytes
const auto size = std::formatted_size(fmt, "Hello", "World");

std::vector<char> buffer(size); Ⓒ Preallocate the required memory
std::format_to(buffer.begin(), fmt, "Hello", "World");
```

Listing 5.22

We first store the format string into a dedicated variable, `fmt`, as shown in Ⓐ. Then we use `fmt` together with the arguments to invoke `std::formatted_size`, as shown in Ⓑ. The result is then used to make the `std::vector` preallocate the required elements, shown in Ⓒ. After that, we have the same result in `buffer`, but with way fewer allocations and copies.

There is one caveat; we pass only `buffer.begin()`. `std::format` has no way of checking whether the size of the buffer is exceeded. There is also a `std::format_to_n`, which we will discuss next.

But let's stay with `std::format_to` for a moment. We can either write the formatted string into an empty buffer or append a string writing to an existing dynamically increasing buffer like a `std::vector`. For that, we use another library facility, `std::back_inserter`:

```
std::vector<char> buffer{'H', 'e', 'l', 'l', 'o', ','};
std::format_to(std::back_inserter(buffer), " {}", "World");
```

Listing 5.23

The difference now is that `std::back_inserter` calls `push_back` on the container, causing an allocation in the container if needed.

5.5.2 Formatting into a fixed sized buffer

The former example has its value, but what if we want to avoid all memory allocations? In that case, we can use, for example, a `std::array`, this time together with `std::format_to_n`.

```cpp
std::array<char, 10> buffer{};
std::format_to_n(buffer.data(),
                 buffer.size() - 1,
                 "{}, {}",
                 "Hello",
                 "World");

std::cout << buffer.data() << '\n';
```

Listing 5.24

The approach using `std::format_to_n` gives us not only safe formatting that stays in bounds, but also ensure that there are no memory allocations. `std::format_to_n` is also a safe way to format a string into a pre-allocated `std::vector`, passing only `begin()` as an iterator.

In C++20, `std::formatted_size` isn't `constexpr`. Should, for example, `std::formatted_size` become `constexpr` in C++23, we could then also determine the size of the `std::array` we used as `buffer` at compile-time. Of course, only if all arguments are available at compile-time as well.

5.6 Writing our own logging function

Suppose for our financial system, we need to log some conditions of the system for debugging purposes. We first create what every logger needs, log levels. We use a class enum for them. Thanks to the abilities of `std::format`, we can provide our own formatter to have a consistent and easy way of formatting the enum `LogLevel`.

```cpp
enum LogLevel { Info, Warning, Error };

template<>
struct std::formatter<LogLevel> : std::formatter<const char*> {
  inline static const char* LEVEL_NAMES[] = {"Info", "Warning", "Error"};

  auto format(LogLevel c, format_context& ctx)
  {
    return std::formatter<const char*>::format(LEVEL_NAMES[c], ctx);
  }
};
```

Listing 5.25

This time our `std::formatter` is simpler than in the previous examples. The reason is that we can derive from `std::formatter<const char*>`. This pre-defined formatter already brings the `parse` method, and as long as we don't need custom format specifiers, we can go with what is already there.

Our log function should be called `Log`, and as the first parameter, we pass the log level, followed by the format string and the format arguments. A first version can look like the code below.

```cpp
template<typename... Args>
void log(LogLevel level, std::string_view fmt, Args&&... args)
{
  std::clog << std::format("{}: ", level)
            << std::format(fmt, std::forward<Args>(args)...) << '\n';
}
```

Listing 5.26

This is a working version, easy to create, and thanks to `std::formatter`, the enum is printed as a string as well. Yet, there is potential for optimizations. The way `Log` is currently written will likely result in a larger binary. For every combination of `args`, a new `Log` function is created by the compiler, and a `std::format` counterpart. That makes inline harder for the compiler. The good news is `std::format` brings the tools to help us out of that misery with `std::make_format_args`.

5.6.1 Prefer `make_format_args` when forwarding an argument pack

Whenever we have an argument pack that is forwarded to `std::format`, we can do two things to improve our binary size. First, we use `std::make_format_args` to create a type-erased wrapper. Second, we pass the result of `std::make_format_args` to `std::vformat` or one of its alternatives (e.g. `std::vformat_to`). The `std::vformat` versions expect a single `std::format_args` as a parameter after the format string. This is the type-erased version of the arguments we created with `std::make_format_args`. Thereby `std::vformat` is no variadic template and does not depend on the argument combination. Here it is:

```cpp
void vlog(LogLevel level, std::string_view fmt, std::format_args↩
    && args)
{
  std::clog << std::format("{}: ", level) << std::vformat(fmt, ↩
      args) << '\n';
}

template<typename... Args>
constexpr void log(LogLevel level, std::string_view fmt, Args↩
    &&... args)
{
  vlog(level, fmt, std::make_format_args(args...));
}
```

Listing 5.27

With that addition, we help the compiler to get the best binary size out for us. Now, using our `Log` function is similar to using `std::format`:

```cpp
void Use()
{
  int x{4};
  std::string share{"Amazon"};
  double d{3'117.02};

  log(LogLevel::Info, "Share price {} very high: {}", share, d);

  errno = 4;
```

Listing 5.28

```cpp
  log(LogLevel::Error, "Unknown stock, errno {}", errno);
}
```

Listing 5.28

There is one more thing we can think about optimizing, the format string.

5.6.2 Create the format specifier at compile-time

In this specific case, the question is to we really need the format specifiers? We have to type empty curly braces all over just to get `std::format` to apply the default formatter for that type. How about a log function that can be used like this:

```cpp
void Use()
{
  int x{4};
  std::string share{"Amazon"};
  double d{3'117.02};

  log(LogLevel::Info, "Share price", share, "very high:", d);

  errno = 4;
  log(LogLevel::Error, "Unknown stock, errno", errno);
}
```

Listing 5.29

We just pass the arguments in the order we like them to be written and omit the curly braces. What do we need to create this `Log` function, and is it doable? The answer is, yes, it is doable. However, `std::format` needs two curly braces; there is no way around. This leaves us with the task of creating the required number of curly braces. We know the number of braces we need. They are equal to `sizeof...(Args)`, the number of elements in the parameter pack. The other good news is, this value is known at compile-time. Consequently, we can write a `constexpr` function that generates the format string for us at compile-time. Let's call the brace generation function `makeBraces`. Here is the implementation:

```cpp
template<size_t Args>
constexpr auto makeBraces()
{
```

Listing 5.30

```
  Ⓐ Define a string with empty braces and a space
  constexpr std::array<char, 4> c{"{} "};
  Ⓑ Calculate the size of c without the string-terminator
  constexpr auto brace_size = c.size() - 1;
  Ⓒ Reserve 2 characters for newline and string-terminator
  constexpr auto offset{2u};
  Ⓓ Create a std::array with the required size for all braces and newline
  std::array<char, Args * brace_size + offset> braces{};

  Ⓔ Braces string length is array size minus newline and string-terminator
  constexpr auto bracesLength = (braces.size() - offset);

  auto i{0u};
  std::for_each_n(braces.begin(), bracesLength, [&](auto& element↩
      ) {
    element = c[i % brace_size];
    ++i;
  });

  braces[braces.size() - offset] = '\n'; Ⓕ Add the newline

  return braces;
}
```

Listing 5.30

Note that makeBraces takes a single NTTP, Args, which donates to the number of arguments in total. That way, makeBraces is independent of the argument combination. This saves us compile-time. The argument count can be easily retrieved with sizeof...(Args) at the call-site. Excellent!

The implementation of Log has to be changed slightly. We need to call makeBraces:

```
void vlog(LogLevel level, std::string_view fmt, std::format_args↩
    && args)
{
  std::clog << std::format("{}: ", level) << std::vformat(fmt, ↩
      args);
```

Listing 5.31

```
}

template<typename... Args>
constexpr void log(LogLevel level, Args&&... args)
{
  A Make the format string
  constexpr auto braces = makeBraces<sizeof...(Args)>();

  vlog(level, std::string_view{braces.data()}, std::↩
      make_format_args(args...));
}
```

Listing 5.31

This allows us to rely on the default formatters for the types. For a log function, especially for debug logs, a great thing. Variables can quickly be logged without the burden of producing a format string. However, there are probably cases where special formatting is better. How do you like `std::format` so far? Is there more we can do?

5.6.3 Formatting the time

We are doing great so far, but what would be a log message without a time stamp? From what you have seen so far, how hard will adding a timestamp using `std::format` be? The answer is, a piece of cake. Below you see code which adds a timestamp.

```
void vlog(std::string_view fmt, std::format_args&& args)
{
  std::time_t t = GetTime();
  std::clog << std::format("[{:%Y-%m-%d-%H:%M:%S}] ", *std::↩
      localtime(&t))
         << std::vformat(fmt, args);
}
```

Listing 5.32

Yes, we can use `chrono` types and supply format specifiers to these types to format the output with `std::format`.

```
$ ./a.out
```

```
[2020-11-18-19:09:07] Info Share price Amazon very high: ↩
    3117.02
[2020-11-18-19:09:07] Error Unknown stock, errno 4
```

The date/time format specifiers used there are compatible with those from other languages. I assume some of you found the entire `std::format` syntax familiar. The reason is that the syntax is mostly borrowed from Python, but C# has some equivalent way of formatting strings.

Cliff notes

- `std::format` gives us type-safe formatting with the clarity of a format string. We can use iostream to output the result.
- Formatting arguments can be reused.
- With `std::format_to_n`, we can format string into an existing buffer without additional memory allocations.
- Use `std::format_to` only with `std::back_inserter` to prevent buffer overflows.
- Custom type can be registered to `std::format` by specializing `std::formatter<T>`.
- We can have a rich set of custom specifiers.

Chapter 6

Three-way comparisons: Simplify your comparisons

The previous chapters have shown some of the enormous improvements provided by C++20. In this chapter, I present the new spaceship operator, which helps us write less code when defining comparisons. Writing comparisons becomes easier and, by default, more correct.

The name spaceship comes from how this operator looks: `<=>`. The operator looks like one of those star-fighters in a famous space-movie series. The operator is not something entirely new; something like it is available in other languages. For C++, it helps to follow the principle of writing less code and let the compiler do the work.

C++ is a language which allows a developer to write less code. For example, we need not write `this` before every method or for every member access. Calls to the constructors and destructors happen automatically in the background according to some rules. With C++11's `=default`, we can request the compiler's default implementation for special member functions, even if under normal circumstances the compiler would not do so. Not having to write special member functions, as simple as a default constructor can be, is something I consider highly valuable.

There was at least one dark corner where the compiler did not help us to the same extent. Whenever we had in the past a class which required comparison operators, we were required to provide an implementation. Sure, if there was some special business going on, then it was sensible to do so, the compiler could not know that. However, consider a basic case where you needed to provide an `operator==` which did something special. For consistency reasons, you usually would have provided `operator!=` as well. But what was its implementation? Well, in all cases I can remember, the implementation was this:

```
bool operator!=(const T& t) { return !(*this == t); }
```

Isn't that a little sad? That is not special code. It is absolutely trivial, but has to be written, reviewed and maintained. Let's see how C++20 tackles this corner.

6.1 Writing a class with equal comparison

Let's start with imagining we have to write some kind of medical application. Whenever you enter a hospital you are identified by a unique Medical Record Number (MRN). This number is used during your entire stay to identify you, because your full name may not be unique. That is even true for my name, which is not common in German. The same goes for my brother. We focus on the implementation of an MRN class. At this point we do not care about all the access functions, just a class with a data member holding the actual value. It shall be default-constructible and constructible by a `uint64_t` which is the internal type of the MRN where the value is stored. Implementing a class `MedicalRecordNumber` can be like this:

```
class MedicalRecordNumber {
public:
  MedicalRecordNumber() = default;
  explicit MedicalRecordNumber(uint64_t mrn)
  : mMRN{mrn}
  {}

private:
  uint64_t mMRN;
};
```

Listing 6.1

Of course, we want two MRNs to be comparable to each other. They can either be equal, which means it is the same patient, or not equal, if the two numbers belong to different patients. There should be no ordering between different MRNs, since they are generated in an unknown order to prevent malicious actions. Objects of the class, as defined so far, cannot be compared with anything. The following trivial code, which tests whether two objects represent the same person, fails to compile:

```
MedicalRecordNumber mrn0{};
MedicalRecordNumber mrn1{3};
const bool sameMRN = mrn0 == mrn1;
```

The compiler does not know how to compare `MedicalRecordNumber` with `MedicalRecordNumber`. Adding the member function `operator==` inside the class makes the previous example work.

```
bool operator==(const MedicalRecordNumber& other) const
{
  return other.mMRN == mMRN;
}
```

Listing 6.2

With that, the former comparison works. But to make not equal (`!=`) work as well, we have to provide an operator for that too:

```
bool operator!=(const MedicalRecordNumber& other) const
{
  return !(other == *this);
}
```

Listing 6.3

6.1.1 Comparing different types

What we have done so far, adding and implementing `operator==`, is still simple. Now, let's say that the MRN should also be comparable to a plain `uint64_t`. This requires us to write yet another pair of equality operators. However, this is not all. Let's think about which comparison combinations we can have. We would like to compare an MRN object with the plain type `uint64_t`. How about the other way around? Sure, that should work as well. This is consistent behavior. Speaking in code, the following should compile:

```cpp
const bool sameMRNA = mrn0 == 3ul;
const bool sameMRNB = 3ul == mrn0;
```

That means we need to add 4 more (2 for ==, 2 for !=) functions to our class, adding up to 6 total methods exclusively for equality comparisons. Two for == and two for !=. We end up with this:

```cpp
Ⓐ The initial member functions
bool operator==(const MedicalRecordNumber& other) const
{
  return mMRN == other.mMRN;
}

bool operator!=(const MedicalRecordNumber& other) const
{
  return !(*this == other);
}

Ⓑ The additional overloads for uint64_t
friend bool operator==(const MedicalRecordNumber& rec, const ↩
     uint64_t& num)
{
  return rec.mMRN == num;
}

friend bool operator!=(const MedicalRecordNumber& rec, const ↩
     uint64_t& num)
{
  return !(rec == num);
}

Ⓒ The additional overloads with swapped arguments for uint64_t
friend bool operator==(const uint64_t& num, const ↩
     MedicalRecordNumber& rec)
{
  return (rec == num);
}
```

Listing 6.4

```cpp

friend bool operator!=(const uint64_t& num, const ↩
    MedicalRecordNumber& rec)
{
  return !(rec == num);
}
```

Listing 6.4

In addition to the two original methods A, we needed two additional overloads B, defined as `friend`-functions, plus two more overloads for swapped arguments C. The boiler-plate code just increased by a lot.

The `friend`-trick

The reason for the `friend`-functions here is not simply to make `==` and `!=` work, but also the comparison to `uint64_t`. Without this `friend`-trick, the following would compile:

```cpp
const bool sameMRN = mrn0 == 3ul;
```

but the other way around wouldn't:

```cpp
const bool sameMRN = 3ul == mrn0;
```

With the operators as friends taking two arguments, they are considered during Argument Dependent Lookup (ADL), because one of the comparison objects is of type `MedicalRecordNumber`.

6.1.2 Less hand-written code with operator reverse, rewrite and `=default`

Do you enjoy writing such code? At this point we have 6 comparison functions from which 4 are simple redirects. C++20 enables us to apply `=default`, which we gained with C++11, here as well, requesting the compiler to fill in the blanks. It makes this code much shorter:

```cpp
bool operator==(const MedicalRecordNumber& other) const = ↩
    default;
bool operator==(const uint64_t& other) const
{
  return other == mMRN;
}
```

Listing 6.5

From six functions down to two, where we can request the compiler to provide one of the two functions for use by using `=default`. That is a reduction I consider absolutely worthwhile. But wait, did I cheat? What about the `operator!=`, they are missing, so I clearly cheated, and we should need instead four functions, which would not be a big reduction. Plus the `friend`-trick is gone, so clearly this code should not work as before. The good news is, I did not cheat, and the code works exactly as before. In C++20 we only need the two functions provided above, period. The reasons for it are two new abilities of the compiler operator reverse and rewrite, which are explained in detail in Section 6.6. Without knowing more about these two abilities, you are already good to go and write your reduced equality comparisons.

6.2 Writing a class with ordering comparison, pre C++20

In the sections before we looked at equality comparison, which is one part. Sometimes we need more than to check for equality, we like to order things. Every the time we need to sort unique objects of a class, it involves the operators `<`, `>`, `<=`, `>=` and the equality comparisons `==` and `!=` to establish an order between the objects which are sorted.

Sticking with the example to write a medical application, think about a class which represents a patient name. Names are comparable to each other, they can be equal (or not) and sorted alphabetically. In our case, the names are stored in a class `String` which is a wrapper around a `char` array and it should offer ordering comparison. It does its job by storing a pointer to the actual string containing the name and its length. Our class is called `String` class and its constructor takes a `char` array. The length is determined by a constructor template. To fulfill the requirements of the patient name, this class should be comparable for equality and provide ordering such that for two `String`-instances, we can figure out which is the greater one, or if they are the same.

As you can see and might have experienced yourself, we need to implement all six comparison functions in order to achieve this. A common approach here is to have one function which does the actual comparison, let's name it `Compare`. It returns `Ordering`, a type with three different values, less (-1), equal (0), or greater (1). These three values are the reason for another name of the spaceship operator, this is sometimes referred to as three-way comparison. In fact, the Standard uses the term three-way comparison for the spaceship operator.

Back to the `String` class. All six comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=` call `Compare` and create their result based on the return-value of `Compare`. At this point, `String` does not have a C++20 spaceship operator, but the result `Compare` gives and the fact that `String` provides all six comparisons makes `String` work as if with C++20 and the spaceship operator. Here it is emulated with pre C++20 code, the way we had to do it for many years.

```
class String {
public:
  template<size_t N>
  explicit String(const char (&src)[N])
  : mData{src}
  , mLen{N}
  {}

  A Helper functions which are there for completeness.
  const char* begin() const { return mData; }
  const char* end() const { return mData + mLen; }

  B The equality comparisons.
  friend bool operator==(const String& a, const String& b)
  {
    if(a.mLen != b.mLen) {
      return false; C Early exit for performance
    }

    return Ordering::Equal == Compare(a, b);
  }

  friend bool operator!=(const String& a, const String& b)
  {
    return !(a == b);
  }

  D The ordering comparisons.
  friend bool operator<(const String& a, const String& b)
```

Listing 6.6

```
  {
    return Ordering::LessThan == Compare(a, b);
  }

  friend bool operator>(const String& a, const String& b)
  {
    return Ordering::GreaterThan == Compare(a, b);
  }

  friend bool operator<=(const String& a, const String& b)
  {
    return Ordering::GreaterThan != Compare(a, b);
  }

  friend bool operator>=(const String& a, const String& b)
  {
    return Ordering::LessThan != Compare(a, b);
  }

private:
  const char* mData;
  const size_t mLen;

  E The compare function which does the actual comparison.
  static Ordering Compare(const String& a, const String& b);
};
```

Listing 6.6

String has the equality comparisons (== and !=) B and the ordering comparisons (<, >, <=, >=) D. The Compare method is private E. Compare E returns Ordering , which as said before, can be seen as a class enum with the three values Equal, LessThan, and GreaterThan. The possibly hardest part is the implementation of Compare itself. The rest is just noise. We ignore implementing Compare at this point and focus on the comparison operators and how much code we have to write to enable comparisons for this, and any other class.

What if `String` should also be comparable to a `std::string`? The number of comparison operators increases by 12! The reason is that we need to provide both operator pairs:

- `const String& a, const std::string& b`
- `const std::string& a, const String& b`

This becomes a lot more boiler-plate code. In fact, for each type we like this class to be comparable with, the number of operators increases by 12. Say that we want it also be comparable to a c-style string, we end up with 30 operators! All of them simply redirecting to the `Compare` function. Ok, the `std::string` version would call `.c_str` on the object.

This is where the spaceship operator comes in, for consistent comparisons.

6.3 Writing a class with ordering comparison in C++20

The spaceship operator in C++ is written as `operator<=>` and has a dedicated return-type which can expressed as less than (-1), equal to (0), or greater than (1), more or less the same as the `Ordering` returned by `Compare` in Listing 6.6. This type, defined in the header `<compare>`, is not required for just the equality comparison functions `==` and `!=`, but is as soon as we want ordering. We will discuss the different comparison types in Section 6.4. With `=default`, we can request a default implementation from the compiler for the spaceship operator, like for the special member functions and the equality operators we saw before. With respect to the `String` example, this means throwing all the `operator@@` out, replacing it by a single `operator<=>`, and include the `compare` header like this:

Listing 6.7

```
#include <compare>

auto operator<=>(const String& other) const = default;
```

Isn't that great? From six functions down to one, and we are done.

6.3.1 Member-wise comparison with `=default`

The truth is, we are not done yet. Requesting the default spaceship or equality-operators will lead to a member-wise comparison done by the compiler for us. Our `String` class contains a pointer, which during a member-wise comparison is not deep-compared. Just the two pointer addresses are. It depends on your application whether this is the right thing. For example, in a scenario where the data behind a pointer is not relevant, only the address of the pointer `=default` can be enough. A memory management class like `shared_ptr` is an example. When comparing two `shared_ptr`, it is enough to know that the two pointers are different, to know that we are looking at two separate allocations. Whether the data of these two pointers is the same is a different question.

Whenever the data to which the pointer refers to should be compared, `=default` is the wrong solution. In such a case, two pointer addresses can be different, but the data behind them can be the same. For our `String` class, for example, two different pointers can still point to two strings, each containing the value `"Franziska"`, and by that, would be considered equal.

Member-wise comparison goes through the member variables in declaration order from top to bottom.

This is exactly the same behavior as for the special member functions. For our `String` class, it implies that there is a little more work to do.

This is how the final version looks:

```cpp
class String {
public:
  template<size_t N>
  explicit String(const char (&src)[N])
  : mData{src}
  , mLen{N}
  {}

  const char* begin() const { return mData; }
  const char* end() const { return mData + mLen; }

  auto operator<=>(const String& other) const
  {
```

Listing 6.8

```
    return Compare(*this, other); Ⓐ We already had this
  }

  bool operator==(const String& other) const
  {
    if(mLen != other.mLen) { return false; } Ⓑ We already had this

    return Compare(*this, other) == 0; Ⓒ Compare does the work
  }

private:
  const char* mData;
  const size_t mLen;

  static std::weak_ordering Ⓓ
  Compare(const String& a, const String& b);
};
```

Listing 6.8

In the implementation of the spaceship operator, we call the `Compare`-function Ⓐ. Same as in the pre-C++20 implementation. Because of the pointer `mData` in `String`, a user-provided `operator==` is required in addition to the spaceship operator. Defaulting `operator==` is not an option, as it would do a member-wise comparison. In `String`, we want to compare the contents of `mData` and not just the pointers.

What does the implementation of `operator==` look like? One option is that it calls the spaceship operator and with that invokes `Compare`. This works. However, think about whether this is the right thing to do. The operation itself works, as the spaceship operator returns the result for equality, after all. Yet, the implementation behind `Compare` has to consider the other cases, less than and greater than, as well. These additional cases might be a pessimization for the equality check. Why? Because some optimizations for the equality check are not possible, if `Compare` is called. One example is that for the equality check we can shortcut the check, if two objects are of different size. This is what Ⓑ does. This optimization was there from the beginning. Two strings cannot be equal, if one is shorter than the other. The equality operator is the only operator which can do this shortcut and return `false` if `a.mLen != b.mLen`. In the spaceship operator, and in our case `Compare`, there is no way to do this shortcut. To determine the result of `Compare` after we know that two strings have a different length, the question is which one is lexicographically less than or greater. Always defaulting to `Compare` or to `operator<=>` means, that even in a case where `a.mLen != b.mLen` is true, `Compare` processes parts of the two strings until a less than or greater result is found. With long strings you can measure the time difference. This is why, in our example, `operator==` calls the underlying function `Compare` C after the shortcut check.

There is one more difference, the return type of `Compare` D. It does now return `std::weak_ordering`, one of the new standard types from the `<compare>` header. Depending on the type, we like to select a different type. For example, if the data follows a strong ordering, we may want that strength. But let's talk about the different comparison categories first.

6.4 The different comparison categories

The different comparison category types the `<compare>`-header offers establish a system how to pick the right type for an implementation of the spaceship operator, or your own function as well. The types provided by the Standard consist of a strength and a category. The naming convention of the types follows the scheme `strength_category`.

6.4.1 The comparison categories

We first figure out the category. Category here means which comparison operations should a type support. We've seen these categories before, a class with only equality comparisons `MRN`, and `String` a class with ordering comparisons. The initial version of `MRN` had neither, also a valid use-case for types.

For a good type design, you can refer to the following rule. The question here is whether the type should be comparable only for equality, or should it also offer ordering? Earlier we had a type with the `MRN` that used only equality. Because two MRNs have no relationship to each other, they are totally unordered. We can only check whether two MRN objects represent the same person. Then there are other cases where a comparison should lead to an ordering of the values. This was the case for `String` that we saw earlier. A type without comparison operators is another option. Such types are unordered and not equality comparable.

After we figure out how the class should behave, we can decide which comparison operations a class needs. In general, every type should either overload

equality only `operator==` and `operator!=`;

ordering all comparison operators;

neither none of the comparison operators.

With that we have the comparison category and the required operators, whether they can be defaulted is a separate question.

6.4.2 The comparison strength: `strong` or `weak`

The next question is the category strength, whether it is *strong* or *weak*. This is only relevant if before we decided that the class provides ordering. Here the question for our type is, if we compare two objects, are they equal or equivalent? For example, `String` earlier is equal and so, the strength is *strong*. Should `String` compare the stored name only in a case-insensitive manner, the strength would be *weak* and the result of two strings being the same would not be equal but equivalent. Another class, for example, could access the stored named and do a case-sensitive comparison, which would lead to another result.

For comparison-strength decision, you can use the following rule of thumb. Use *strong* if everything that is copied, in the copy constructor, is also part of the comparison. If only a subset of what is copied is compared, use *weak*. Subset also means that, if the comparison is done in a special way, as in case-insensitive comparison of a string. The strength then is also *weak*. The type `strong_ordering` corresponds to the term total ordering in mathematics.

As a real-world example for a `strong_ordering` consider the Russian Matrjoschka dolls. They are made of wood and usually colorfully painted. They come in a egg-like shape and are nestable. You either start or end up with a single piece which contains all the others. They nest into each other perfectly. There is only one order of sorting them by size. It cannot be changed without damaging the dolls, and there are never two of the same size in a set. This represents a strong ordering.

The types `strong_ordering` and `weak_ordering` have three different possible values, `greater`, `equivalent`, and `less`.

6.4.3 Another comparison strength: partial ordering

There is also a third category, `partial_ordering`. It has the same three values as `weak_ordering`: `greater`, `equivalent` and `less`. But it has an additional value, `unordered`. We should use this whenever we have a type that is not fully orderable. An example is a class with a **`float`**. For the values 0 and −0 of a **`float`** there is no ordering between them. Similarly, the value Not a Number (NaN) of a **`float`** is not comparable to anything. Choose `partial_ordering` when all six comparison operators are needed, but from some values none of `a < b`, `a == b`, and `a > b` needs to be true. That means that all the three checks can be false at the same time. Because of that, keep in mind that no STL algorithms will work with a type with a spaceship operator which returns `std::partial_ordering`. The same is true for `std::set` or `std::map`. It means that for these tasks the return value is simply unordered.

We talk about equal, if we use equality comparison. When we use ordering comparison, this is equivalence.

These types can be compared to a numeric value. Table 6.1 lists all the types and their corresponding values.

Instead of the numeric values, you can also use the comparison functions, as listed in Table 6.2.

A real-world example for partial ordering is dressing. We have to put on underwear before we can put on pants. That part is orderable. It starts to get difficult when we talk about when to put on socks. We can put them on before or after putting on pants. And with which one do we start, left or right? Even if we establish a system like socks before pants, it doesn't help in which sock to put on first. But then, putting on shoes before socks isn't the right thing. In this example, we have parts in the clothing set which are unordered.

6.4.4 Named comparison functions

As we saw in the section before, the possible values of `std::weak_ordering` are less than (−1), equal (0), or greater (1). In the `String` class example in Subsection 6.3.1, we did compare the result of `Compare`, which is of type `std::weak_ordering` to zero. Of course, this is the same for checking the result of the spaceship operator.

Table 6.1: Comparison types and their values

Category	-1	0	$+1$	Non-numeric values
`strong_ordering`	less	equal	greater	
`weak_ordering`	less	equivalent	greater	
`partial_ordering`	less	equivalent	greater	unordered

This is one way to check the value of, in this case, `std::weak_ordering`. There is an alternative available, using one of the new named comparison functions in the namespace `std`, like `is_eq`. The named comparison functions are also available from the `compare`-header. Table 6.2 provides a complete list.

Table 6.2: Named comparison functions

Function	Operation
`is_eq(partial_ordering cmp)`	`cmp == 0`
`is_neq(partial_ordering cmp)`	`cmp != 0`
`is_lt(partial_ordering cmp)`	`cmp < 0`
`is_lteq(partial_ordering cmp)`	`cmp <= 0`
`is_gt(partial_ordering cmp)`	`cmp > 0`
`is_gteq(partial_ordering cmp)`	`cmp >= 0`

Applied to the `String` example, the implementation of `operator==` uses `is_eq` instead of comparing to zero:

```
bool operator==(const String& other) const
{
  if(mLen != other.mLen) { return false; }

  return std::is_eq( Ⓐ Using a named comparison function
    Compare(*this, other));
```

Listing 6.9

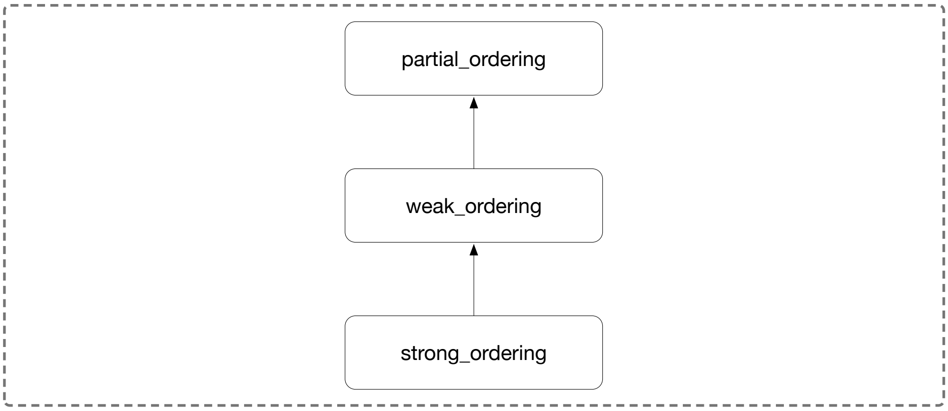


Figure 6.1: The different comparison categories and how they relate to each other. Arrows show an *is-a* implicit conversion ability.

```
}
```

Listing 6.9

Using the named comparison functions makes the code speak a little more. Other than that, both ways are equivalent, you are free to choose the one that fits better in your code-base or coding style.

6.5 Converting between comparison categories

There are conversion rules between the different categories. A closer look at Table 6.2 shows that all named comparison functions only expect `partial_ordering`. This is due to the ability of the different categories to be convertible into a less strict category. For example, `strong_ordering` can be converted into a `weak_ordering`. By that we loosen some requirements. The other way around isn't possible, of course, as there is no way to add requirements to a type.

Figure 6.1 gives a graphic illustration of how the different categories convert to each other. They follow the natural order. Something that has `strong_ordering` can be converted to something weaker, in this case `weak_ordering`. Something that has an order has an equivalence check as well. Two objects which are equivalent can be translated to an equality comparison. This is independently true, whether the dataset contains equivalent elements or not. The spaceship operator has a value for equality. For a certain dataset, this just may never be true.

From a `strong_ordering`, we can derive a `weak_ordering`, or even a `partial_ordering`. That's why these types are implicitly convertible to each other in this direction. The other way around does not work.

These implicit conversions happen automatically, for example, when we default `operator<=>` and let the compiler deduce the return-type. Below is a more concrete example.

```
struct Weak {
  std::weak_ordering operator<=>(const Weak&) const = default;
};

struct Strong {
  std::strong_ordering operator<=>(const Strong&) const = default↩
      ;
};

struct Combined {
  Weak w;
  Strong s;

  auto operator<=>(const Combined&) const = default;
};
```

Listing 6.10

We have the two types `Weak` and `Strong`. They both define a defaulted spaceship operator. `Weak` returns `weak_ordering` for the operator, while `Strong` returns `strong_ordering`. The third type `Combined` contains the former types as members. `Combined` declares a defaulted spaceship operator with return-type `auto`. The question is, what is the deduced type? The answer is that the compiler automatically determines the common comparison categories among all types. This type is then used as the deduced return-type. There is also a type-trait which can identify the common comparison category for you: `std::common_comparison_category`.

In case you would, for example, specify `strong_ordering` as return-type of the spaceship operator for `Combined`, the compiler would refuse to compile it.

6.6 New operator abilities: reverse and rewrite

The former examples work, even though we've only ever specified the `operator==`, which I think is a good thing. So how does the compiler manage to do the operator `!=` as well? Very simple, in C++20 the compiler can reverse or rewrite comparisons. If there is only a user-provided `operator==` the compiler assumes that the implementation of `operator!=` simply is the negated result of the == operation. So when we say not equal (not ==) we now get exactly what we asked for, thanks to the compiler ability to rewrite `!=` into `! operator==`. This is a comparison rewrite.

The C++20 compiler is even mightier than rewrites — it can also reverse operands, if there is a match in the overload set. This is the reason it is enough to provide the member version, even for `uint64_t` in the previous `MedicalRecordNumber` example. For example: it is possible to change the operands in the comparison like this:

```
const bool sameMRN = mrn0 == 3ul;
const bool sameMRN = 3ul == mrn0;
```

The only remark here is, that before C++20 we needed to provide a dedicated `operator==` to make the latter version work. Remember that we needed to provide these overloads via the `friend`-trick in the example in Subsection 6.1.1. In C++20, regardless which variant we pick, the compiler is allowed to reverse the operands, such that both variants end up as this:

```
const bool sameMRN = mrn0.operator==(3ul);
```

The benefit is that we no longer need the `friend`-function trick.

For every *primary* comparison, as shown in Table 6.3 in form `a.operator@(b)` the compiler may reverse it to `b.operator@(a)`, if this is the best match.

For every *secondary* comparison, as shown in Table 6.3 it allows the compiler to rewrite an expression, `a.operator!=(b)` then becomes `! a.operator==(b)`.

These two operations can appear combined too. For example, if we have `a.operator!=(b)` as before, but say `a` is just an `int`, then the compiler can do a reverse and rewrite leading to this transformation: `! b.operator==(a)`. It assumes that there is no matching `operator!=` in `b`.

You can now see why this feature is also called *consistent comparisons*. We automatically get the same result for `a @ b` as we do for `b @ a`, even in a case where `a` and `b` are of different types.

Table 6.3: Operator categories

	Equality	Ordering
Primary	`==`	`<=>`
Secondary	`!=`	`<`, `>`, `<=`, `>=`,

How does spaceship work

We basically need two comparison operations:

- Equal (==);
- Greater than (>), alternatively less than (<), see below.

With these two operations we can build the others:

- Not equal (!=), is the opposite of equal;
- Less than (<), is not equal and not greater than;
- Greater than or equal (>=), is the opposite of less than;
- Less than or equal (<=), is the opposite of greater.

This is knowledge that the compiler can also have, and it has it in C++20 where it may reverse operands and rewrite comparison expressions.

6.7 The power of the default spaceship

As we discussed before, we can default the spaceship operator. This helps a lot for simple classes or structs which act like aggregates. They have one or more types and wrap them inside because they provide additional functionality. As soon as we define even a `struct` with just a single `int` in it, we cannot compare two objects of the `struct`'s type. But the additional functionality is probably more about restricting read or write access, or providing a bounds check. Depending on what we like our type to model, we need to provide either 2 or 6 comparison operators. The implementation then is trivial. It is borderline worth writing. Even for a novice, there are much better, more meaningful tasks to do. We can forget or mix up the `const`, the `constexpr`, and so forth.

There is more. Every the time you write a wrapper-like class which wraps one or more types, you have to provide the comparison operators, if the wrapper should add something like access-control or bounds-checking. Say we model a Binary Coded Digit (BCD) number:

```
class BCD {
public:
  BCD(int v)
  : mValue{Adjust(v)}
  {}

  A Make it convertible to int
  operator int() const { return mValue; }

  B Provide at least equality comparison
  bool operator==(const BCD& rhs) const
  {
    return rhs.mValue == mValue;
  }

  bool operator!=(const BCD& rhs) const
  {
    return not(*this == rhs);
  }

private:
  int mValue;

  static int Adjust(int v);
};
```

Listing 6.11

A is just to make BCD convertible back to an `int`. You can also envision more additional methods, maybe you also provide the `operator+` and so forth. However, the central point is that without B this piece of code does not work. It wouldn't compile as there is no equal comparison operator. So we add them. You can imagine how the implementation of the other operators (<, >, ...) would look like. Trivial. Now, what happens if we add another member, `int` again, to store the significance of the digit? Sure, the process is again easy, just add the extra member. Oh yes, and please don't forget to adjust all the comparison operators. Should you forget it, which happens, the code compiles, but does not work as intended. How about this:

Listing 6.12

```
class BCD {
public:
  BCD(int v, int significance)
  : mSignificance{significance}
  , mValue{Adjust(v)}
  {}

  operator int() const { return mValue; }

  Ⓐ Provide at least equality comparison
  auto operator<=>(const BCD&) const = default;

private:
  int mSignificance; Ⓑ The additional member just works
  int mValue;

  static int Adjust(int v);
};
```

We added the new member and defaulted the spaceship operator, done. We got rid of the ridiculous boiler-plate code. Yet, there is more. Our code is now safer, and correct by default. Adding an extra member does not require touching the comparison operators at all.

Furthermore, think about making the class `constexpr` in the old world. Copy and past `constexpr` to all the comparison operators. Here is what you do with spaceship:

Listing 6.13

```
constexpr auto
operator<=>(const BCD&) const = default; Ⓐ Add constexpr
```

All that is left to do is adding `constexpr` one time Ⓐ and we are done with the operators.

6.8 Applying a custom sort order

We have seen that we can either implement the spaceship operator ourselves or default it. Sometimes we need a bit of a mixture. Consider this example:

```cpp
struct Address {
  std::string city;
  std::string street;
  uint32_t street_no;

  auto operator<=>(const Address&) const = default;
};
```

Listing 6.14

It models an Address containing a city street name and number. By opting in for the default spaceship-implementation when comparing two city objects, we get lexicographical order for city and street name. It then sorts the street number in descending order. Typically, we deal with street numbers in ascending order. We can correct this by providing our own implementation for `operator<=>`:

```cpp
struct Address {
  std::string city;
  std::string street;
  uint32_t street_no;

  auto operator<=>(const Address& rhs) const
  {
    A Sort city and street using their <=>
    if(const auto& cmp = city <=> rhs.city; cmp != 0) {
      return cmp;
    } else if(const auto& scmp = street <=> rhs.street;
              scmp != 0) {
      return scmp;
    }

    B Next, sort street_no ascending
    return rhs.street_no <=> street_no;
```

Listing 6.15

```
  }

  Ⓒ The default should be good enough here
  bool operator==(const Address&) const = default;
};
```

Listing 6.15

Thanks to the availability of the spaceship operator in the STL containers, we can invoke them for `std::string` Ⓐ. By combining it with C++17s `if`-with-init we can declare the comparison result in the head of the `if` head and directly use it as the `if`-condition. In case the result is not equal we have our ordering, and return the former obtained value `comp`.

After we've done this for the two `std::string` members, we can sort the street name in ascending instead of descending order. As Ⓑ shows, this is achieved by swapping the arguments to `<=>`. You can further see that the spaceship operator is defined for built-in types as well.

This is one case where it is ok to default `operator==` Ⓒ, as it does not depend on the order criteria we changed.

6.9 Spaceship-operation interaction with existing code

Suppose you have a legacy `struct` from pre-C++20 times. It consists of some members and the equality and less-than operators. Such a `struct` may look like this:

```
struct Legacy {
  int a;

  Ⓐ  These define a weak order
  bool operator==(const Legacy&) const;
  bool operator<(const Legacy&) const;
};
```

Listing 6.16

The two operators Ⓐ define a weak order. However, as it was written before C++20, it doesn't have a spaceship operator. Which at times was totally fine, of course. Now, if we have to write a new class `ShinyCpp20Class` which has a field of type `Legacy`. It should be ordering comparable, preferably by using the spaceship operator. The question is, how do we do that? What is the most efficient and painless way to write it? We can implement a spaceship operator which does the three-way comparison for each of the struct's fields. This is writing boiler-plate code. Thanks to a last-minute update to the Standard, we can just default the spaceship operator:

```cpp
class ShinyCpp20Class {
  Legacy mA, mB;

public:
  ShinyCpp20Class(int a, int b);

  std::weak_ordering (A)
  operator<=>(const ShinyCpp20Class&) const = default;
};
```

Listing 6.17

The only special thing is that we have to be specific about the return-type of `operator<=>`. We cannot just say `auto`. (A) shows that in this case we apply `weak_ordering` as the return type. This is what `Legacy` allows us. This gives us the ability to incorporate pre-C++20 code into new, code and apply the spaceship operator there.

A word about upgrading to C++20

When we say consistent comparisons, we mean it. This is a marvellous thing, it makes the language stronger and reduces the places where we can make mistakes.
C++20 does clean up some bad code we could have written before C++20, sadly leading to some interesting situations. Have a look at the following code:

```cpp
struct A {
  bool operator==(B&) const { return true; } (A)
};

struct B {
  bool operator==(const A&) const { return false; } (B)
};
```

Listing 6.18

The piece of code above is questionable. Not because of the silly return true and false. Have a look at the parameter signatures. A takes a non-`const` parameter, which is quite questionable. On the other hand, B looks like a signature as it ought to be. Despite the missing `const` on A, this code works just fine, as long as it is not invoked with a const object of type `B`. What are the odds for that? Well, if there is a `const B` object, the code stops compiling. Assume that it compiled in C++17, it implies that it worked *correctly* in C++17, missing `const` or not.
Remember that in C++20 we have consistent comparisons? They are consistent even without the spaceship operator. Why? Because for the equality operator, it allows the compiler to reverse the arguments. When it does this with the code above, A becomes the better match in the overload set, because it does not have the `const`. Using B requires an internal `const`-cast of the compiler for object `A`. These additional actions make A the better match, as there is none required.
Now, the code has worked in C++17, but for the wrong reasons. There are more corner-case examples like this. All of them reveal inconsistencies. By upgrading your code to C++2,0 the compiler will point all these inconsistencies out to you. The assumption is that such things happen only very, very rarely.

Cliff notes

- Consistent comparisons are a valuable feature which can take a lot of boiler-plate code from our plate. At the same time it ensures that our comparisons are heterogeneous and with that consistent.
- The compiler may reverse every comparison of the form `a.operator@(b)`: the compiler may change it to `b.operator@(a)`, if this is the best match.
- For that reason you no longer need the `friend`-trick, just declare your comparison operators as member functions.
- The compiler also performs *rewrites* where `a.operator!=(b)` can be rewritten to `!a.operator==(b)`.
- We can `=default` all comparison operators which take the class itself as an argument.
- `constexpr` can be added or removed when using `=default`.
- Remember that `=default` does a member-wise comparison. This is the same meaning for the comparison operators as for the special member functions like the copy constructor.
- Member-wise comparison goes through all members in declaration order from top to bottom.
- When defaulting comparison operators, prefer the *primary* operators. Since they provide the full complement of all comparisons and the compiler can rewrite them, this is all you need.
- When you default `operator<=>`, you automatically also get the equality comparison operators. However, if you provide your own `operator<=>` implementation, also provide your own `operator==` version. Refrain from defaulting `operator==` in this case, as it still defaults to member-wise comparison.

- You need to include the `<compare>` header to get the new std-types.
- Invoke `<=>` when you ask for comparison, use `==` when you ask for equality.

Chapter 7

Lambdas in C++20: New features

Since lambdas were introduced into C++ with C++11 they improved with every new Standard since then. C++20 is no exception. There are several changes to make lambdas even more powerful.

7.1 [=, this] as a lambda capture

One of the brilliant things about lambdas is that a lambda can capture values from the surrounding context. They are a handy helper whenever we need to create a new type with only a subset of values and a specific action. What the correct and or best capture way is, is a difficult topic. Some of you prefer to list each variable used inside of the lambda and its capture-form explicitly. This ensures, that the lambda captures only intended variables. As soon as someone uses an additional variable, not part of the explicit capture-list, this results in an error and the compilation terminates. This strategy contains the risk, that if we ask the compiler to capture something it will be captured, regardless whether we use it in the end. We can end up with unused variables on the lambda, which create pressure on the system because they are expensive to copy.

The other strategy is to use capture-defaults, = or & and let the compiler figure out the used variables in the lambda's body. That way there are no unused, probably expensive variables. However, the downside is that variables that shouldn't be captured may end up being captured.

The capture-defaults come with another specialty. We need not only to know that = stands for capture-by-copy, while & donates to capture by-reference, but we also need to know what exactly capture-by-copy means. In this simple case it means that a certain variable c is copied into the lambda and with that an independent exact copy of the original c.

Let's see a simple example to prepare the way for what might go wrong with capture-by-copy. A lambda twice which multiplies the value of c by two.

```cpp
int c{3};

auto twice = [=] { return c * 2; };
```

Listing 7.1

There is nothing to question here. But what if that lambda is inside a class method and c is a member of this class? Assuming we write the same twice lambda with the same capture, what does it actually capture? Such a scenario might look like this:

```cpp
class SomeClass {
  int c{3};

public:
  void SomeCleverMethod()
  {
    auto twice = [=] {
      A Implicitly capturing this here instead of just c
      return c * 2;
    };

    // ...
    const auto v = twice();
  }
};
```

Listing 7.2

C++ Insights

C++ Insights is a clang based tool, showing C++ code with the view of a compiler. Making implicit conversions or template instantiation visible are just two among a lot of things it does. It is available as command-line version and has a web-front end at `https://cppinsights.io`

It is the same lambda, doing the exact same thing. However, in this case `=` copies `this` rather than `c` making the lambda point to the class and not a single integer. C++ Insights can visualize the details about this.

The lambda now has two `this`-pointers. Its own as every class has and the copy of `SomeClass`. Let's call the captured one `__this`. The accesses to `c` then happens via the `__this`-pointer inside the lambda. The issue is that in this case the lambda does not hold a copy of `c` rather than a copy of the `this`-pointer. However, `this` is a pointer and with that we only have a second instance of a pointer. It does not copy the data behind it. Below is a C++ Insights transformation of the code which visualizes the details. In the transformed code you can see that the compiler first A creates a class for us, the closure type, which has a single field `__this` B. This field is a pointer of type `SomeClass`. Later, when an instance of the lambda is created C the compiler passes `this` to the constructor.

```
class SomeClass {
  int c;

public:
  void SomeCleverMethod()
  {
    A Lambda internals created by the compiler
    class __lambda_8_18 {
    public:
      int operator()() const { return __this->c * 2; }

    private:
      SomeClass* B The captured object, a pointer to SomeClass
        __this;

    public:
```

Listing 7.3

```
      __lambda_8_18(SomeClass* _this)
      : __this{_this}
      {}
    };

    __lambda_8_18 twice = __lambda_8_18{this}; C Passing this
    const int v = twice.operator()();
  }
};
```

Listing 7.3

This implicit capture of the `this`-pointer can be a problem, for instance, if we accidentally return this lambda and the class itself goes out of scope. The data to which `__this` points to is no longer valid. The lambda then is a ticking time bomb. As soon as one accesses the contents of the lambda it is UB. C++17 gave us a mitigation for this case by adding a new capture variant `*this`, which stands for capturing the dereferenced `this`. The lambda then contains a deep copy of `this` and not just a duplication of the pointer. This makes the intention clear and provides us with a way to capture an actual copy of `this` which is great. However, there is now an asymmetry. Consider a code-review situation and you are about to review the code with the lambda in a class. One cannot be sure that the code does what we intend it to. As a programmer we have no way to express *yes I like to copy only the pointer of* `this` *and not the entire object*. The review is easier, if there is the `*this` capture present. Then we can assume that the author knew what they were doing. The absence itself doesn't come with the same clear meaning.

C++20 adds this missing piece by adding an additional capture option `this`. Now we can express whether the capture of this is just the pointer or a deep copy. In the example before we need to update the capture clause from `[=]` to `[=, this]` to ensure, that we can use the variable `c` and that `this` is captured as pointer.

```
auto twice = [=, this] A Explicit capture this as pointer
{ return c * 2; };
```

Listing 7.4

The behavior that `=` implies is an implicit capture of `this` is deprecated with C++20. This is for compatibility reasons to give users time to upgrade their code without breaking it directly. The chances are high, that the implicit `this` capture will be removed with, say, C++23. Upgrading now not just spares you a break later, it also makes code like this more precise and meaningful.

7.2 Default constructible lambdas

Now let's consider default constructible lambdas and how they can improve our code. Lambdas are considered default constructible, if they have no captures.

Let's say we have to write an application for a book publisher. A very simple one. We have books which have a title and an International Standard Book Number (ISBN). Then we have two prices, a normal price 34.95 and a reduced price 24.95. Both book and price are two independent structs in our code. We are asked to create a map between books and the related price. This map should be sorted by the ISBN. The application will probably use this map multiple times. We create a `using` -alias called `MapBookSortedByIsbn`. This alias defines the key type as `Book`. Only the value's type must be supplied by a user to allow more than just mapping a book to a price.

To make this `std::map` work, we need to define a custom compare function or provide a spaceship-operator for `Book`. As we possibly like to have different ways of sorting books, we use a custom compare function. This is essentially a one-liner, so we go for a lambda and name it `cmp`. To make `std::map` work with a custom compare function, we need to define it first and then tell `std::map` the type of it in the template-argument list. We can use `decltype` to get the type of `cmp`. Here is a possible implementation:

```
struct Book {
  std::string title;
  std::string isbn;
};

struct Price {
  double amount;
};

auto cmp = [](const Book& a, const Book& b) Ⓐ Compare lambda
```

Listing 7.5

```
{ return a.isbn > b.isbn; };

template<typename VALUE>
using MapBookSortedByIsbn = Ⓑ Typed map for multiple use
  std::map<Book, Ⓒ Use Book as type for the map
          VALUE, Ⓓ The value can be provided
          decltype(cmp) Ⓔ Type of the custom compare function
          >;
```

Listing 7.5

The idea of the `using`-alias which creates `MapBookSortedByIsbn` Ⓑ is that a potential user has to type less and we can provide a consistent element which acts like a type. But how easy is `MapBookSortedByIsbn` to use? We first have to pass the value type to the map, in this case `Price` to create the type. Then can add some items by using an initializer list. The tricky part is that we need to pass the `cmp`-function as an argument during construction of the map. Not only that, a user needs to pass `cmp` as a constructor argument, they also need to know the name of the function. The initial concept of having a type which says that is it a map of books sorted by ISBN implies that a user has not to know the name of the sort function. Passing a different function than `cmp` will lead to a compiler error in this case. If we had chosen a function instead of a lambda, passing an entirely different function would be possible. Only the function's signature has to match. With that, it would break the promise of the type's name. Luckily we chose the lambda and the compiler will tell a user about the mistake.

Here is an example which uses two books and a normal and reduced price. In the creation of the map Ⓐ, we add the two books Ⓑ but then have to know and pass the compare function Ⓒ.

```
const Book effectiveCpp{"Effective C++", "978-3-16-148410-0"};
const Book fpCpp{"Functional Programming in C++", "↩
    978-3-20-148410-0"};

const Price normal{34.95};
const Price reduced{24.95};

MapBookSortedByIsbn<Price> Ⓐ Use the map with Price as value
```

Listing 7.6

```
book2Price{
  {{effectiveCpp, reduced}, {fpCpp, normal}}, B Add some items to
      it
  cmp C Sadly we have to know and pass the cmp function
};
```

Listing 7.6

The question is, why has this to be so hard? Why do we need to pass cmd not only as a type to the std::map but also as a constructor argument? We should be able to skip the constructor argument, as std::map should be able to create an object of type cmp. The answer is, that std::map can default construct an object of the compare-type, if we provide no custom compare function. Now, here is the but, in C++17 and before, lambdas are not default constructible. The Standard defines them with a deleted default constructor. This restriction is lifted in C++20. We can drop naming the compare function from the constructors argument list. With that users do no longer need to know the name of the compare function and don't have to type it. This enables us to drop C from the former example. The improved version is shown below.

```
MapBookSortedByIsbn<Price> book2Price{
  {effectiveCpp, reduced},
  {fpCpp, normal}
  A cmp is gone
};
```

Listing 7.7

We have now a clean map to which a user needs to supply only their data.

7.3 Captureless lambdas in unevaluated contexts

What do you think about the previous example, is there something else to improve? I think yes. The compare function cmp or more precisely lambda. By now we created a callable, but what did we do with it? We used **decltype** to pass its type to std::map. So we never needed a callable, all we needed was a type of a lambda and stored it in cmp. As a result, we have a variable in the scope with the name cmp. What can we do about this? There are two different solutions that come to mind. First and probably most obvious, instead of creating a variable `cmp` move the lambdas definition directly into the `decltype`-expression when defining the alias.

```cpp
template<typename VALUE>
using MapBookSortedByIsbn =
  std::map<Book,
           VALUE,
           decltype([](const Book& a,
                        const Book& b) (A) Provide the lambda in-place
                    { return a.isbn > b.isbn; })>;
```

Listing 7.8

This seems to be the cleanest way. The compare definition is now inside of `MapBookSortedByIsbn` with no additional symbol. It is not contained in the surrounding scope. Whenever we need the compare definition only once this is the cleanest approach.

There is an alternative, say that we need the type of `cmp` more than once. We said above that we need the type. With the help of `using`-alias and the use of `decltype` we can create a type and use it later as a template-argument for the `std::map`:

```cpp
(A) Define a using alias with the compare lambda which can be reused
using cmp =
  decltype([](const Book& a, const Book& b) { return a.isbn > b.↩
      isbn; });

template<typename KEY, typename VALUE>
using MapSortedByIsbn = std::map<KEY,
                                 VALUE,
                                 cmp>; (B) Use the using-alias cmd
```

Listing 7.9

Both approaches seem natural solutions to the question. However, before C++20 they were not allowed because we could not define lambdas in unevaluated contexts. This is another restriction C++20 lifts and by that makes lambdas first-class citizens.

Please note that lambdas in a `decltype` expression, or in an unevaluated context in general, works only with captureless lambdas. Lambdas with captures need arguments during construction, the captures which are not available in an unevaluated context.

7.4 Lambdas in generic code

Now, a book publisher might do more than just books. They also publish magazines and other material, all with an ISBN. As they are different publishing types, we like to have different types for them. Instead of having just `Book` we like to also have say `Magazine`:

```cpp
struct Magazine {
  std::string name;
  std::string isbn;
};
```

Listing 7.10

With that additional type a new use-case is to have a more generic map that works with `Book` and `Magazine` such that the following code works:

```cpp
const Book effectiveCpp{"Effective C++", "978-3-16-148410-0"};
const Book fpCpp{"Functional Programming in C++", "↩
    978-3-20-148410-0"};

const Magazine ix{"iX", "978-3-16-148410-0"};
const Magazine overload{"overload", "978-3-20-148410-0"};

const Price normal{34.95};
const Price reduced{24.95};

MapSortedByIsbn<Book, Price> book2Price{{effectiveCpp, reduced},
                                        {fpCpp, normal}};
MapSortedByIsbn<Magazine, Price> magazine2Price{{ix, reduced},
                                                {overload, normal}};
```

Listing 7.11

Making the above code possible requires some updates to `MapBookSortedByIsbn` as it currently only works with books. To make `MapBookSortedByIsbn` more flexible, one part is requiring the type of the key as a template argument. With that we can reflect this new flexibility in a new name, renaming `MapBookSortedByIsbn` to `MapSortedByIsbn` as we can now create a map for `Book` and `Magazine`:

There is a second part we need to change to make the code compile, the compare lambda. In the previous versions it took two arguments, both of type `Book`. Needless to say, that will not compile with `Magazine` as the type. To make the previous version work, we can fall back to C++14's generic lambdas. The former `Book` parameters now become `auto` and the code compiles:

```cpp
template<typename KEY, typename VALUE>
using MapSortedByIsbn =
  std::map<KEY,
           VALUE,
           decltype([](const auto& a, Ⓐ Using a generic lambda
                       const auto& b) { return a.isbn > b.isbn; })>;
```

Listing 7.12

7.1 C++14: Generic lambdas

Generic lambdas are a C++14 feature, they allow `auto` as a parameter-type. Lambdas where the only place where `auto` was valid as a parameter-type prior to C++20. It essentially generates a templated call-operator where each `auto`-parameter becomes a template-type parameter and by that allows the lambda to be invoked with any type.

7.4.1 Lambdas with templated-head

The version we created in the last section works and is fine so far. Yet, there is something more to improve. We previously defined the two parameters the compare lambda takes to be of the same type, `Book`. Now they are both `auto` and with that can be independent types. The `std::map` will not instantiate them with different types, but the code can give a different impression. There are ugly workarounds with `decltype` to make both parameters of the same type. Don't do this. We can also decide to move away from the lambda and use a function-template instead. There we can express that the two parameters must be of the same type.

C++20 adds the ability to lambdas to have a template-head like functions or methods. This enables us to provide not only type-parameters but also NTTP for a lambda. We can also have a lambda that takes multiple parameters, all of the same template-parameter-type. Just like with regular templates. The syntax is the same as for other templates, except that we do not need to spell out the `template`. The template-head itself comes after the capture-list:

```
template<typename KEY, typename VALUE>
using MapSortedByIsbn =
  std::map<KEY,
           VALUE,
           decltype([]<typename T> Ⓐ Lambda with a template-head
                    (const T& a, Ⓑ Use T as in a regular template
                     const T& b) { return a.isbn > b.isbn; })>;
```

Listing 7.13

As you can see, with the template-head we can define a template parameter `T` and use it for both parameters of the lambda, `a`, and `b`. Now both parameters must be of the same type. The compiler checks it and we as users can see it when looking up the definition.

7.4.2 Variadic lambda arguments

There is more, since we now have a template-head we can apply the usual powers of templates to lambdas. Suppose we have a generic lambda which has a variadic number of arguments. Correct, this works for C++14's generic lambdas as well. Remember, under the hood they are templates the `auto`-parameter is a hint for that. This variadic generic lambda should now forward its arguments to another function. Why? Because we need to insert another argument first and then the passed arguments: a technique for lambdas which is often referred to as partial application.

Let's write a `print` function for a vehicle which allows the steering and brake system to log diagnostics. We make this `print` function a variadic template, taking an arbitrary number of arguments. This function can be used directly. However, to identify the log source, steering or brake system, we like to insert the origin as the first argument. The straight forward solution would be to write the origin we need as the first argument of each `print` invocation. That works, but buries some risks.

In the simplest solution, we duplicate the origin name repeatedly. There is a non-zero potential that people end up spelling the origin differently. A mitigation would be to create a global variable, for example `originSteering`, with the origin inside such that this variable then can be used. Better but now we have a global variable which has its own drawbacks like order of initialization. Plus `print` can still be invoked without an origin. Do you want to take a bet that there is only one place where this happened when you got a trace and need to identify where it came from? In my experience, chances are high that there is more than one place.

```
Ⓐ Passing differently spelled origins
print("Steering"s, "angle"s, 90);
print("steering"s, "angle"s, 75);

Ⓑ Declaring a global variable for the steering origin
static const auto originSteering{"Steering"s};

print(originSteering, "angle"s, 90); Ⓒ Ok, use of the global variable
print("steering"s, "angle"s, 75); Ⓓ Passing steering instead of
    Steering
```

Listing 7.14

All these approaches come with drawbacks. Even with a global variable for an origin users can still pass a plain string. The API of `print` is not obvious. This holds even in a case where a user passes the global origin variable, is the second parameter still part of an origin or is it part of the log message? Here lambdas can be a solution.

```
auto getNamedLogger(const std::string origin)
{
  return [=](auto... args) {
    print(
      origin,
      std::forward<decltype(args)>(
        args)... Ⓐ Forward the arguments using
             decltype to retrieve the type
    );
  };
}
```

Listing 7.15

Perfect forwarding

Since we have move-operations with C++11, we can perfect forward objects. The term donates to the fact that copies and with that potential allocations and deallocations are reduced, either for temporary objects, or for object we no longer intend to use.

As you can see, we have a function `getNamedLogger` which takes the origin as an argument and returns a generic variadic lambda which captures the origin as a copy. That way a user can create a named logger, pass the origin and can later invoke the returned object with a different number of arguments. The origin is always printed first. To make this solution more efficient, we can use perfect forwarding inside the lambda that `getNamedLogger` returns. We don't have to, but we can avoid some copies of strings or other resource intensive object when we forward them, saving additional allocations and copies. The example above is a possible C++17 implementation, which uses perfect forwarding.

Now, `getNamedLogger` can be used by calling `getNamedLogger` with, for example, "Steering" as parameter to create a logger for the steering gear. We store the result in a **`auto`**-variable `steeringLogger`. After that we can use and invoke `steeringLogger` like a regular function. We can pass one or multiple values which are processed by the `print` function inside the lambda `getNamedLogger` has created for us. The plus is that users refer to the variables `steeringLogger` and `breakLogger` wihtout the need to known about `getNamedLogger` and what value to pass as origin for which system.

```
auto steeringLogger = getNamedLogger("Steering"s); A Logger for
     steering
auto breakLogger = getNamedLogger("Breaks"s); B Logger for breaks

steeringLogger("angle"s, 90); C Log a steering related message
```

Listing 7.16

7.4.3 Forwarding variadic lambda arguments

Remember, that **`auto`** stands for a template parameter. We can apply **`decltype`** on it to get back the type and supply this to `std::forward`. Admittingly it is a small price to pay to bring **`decltype`** to the party, but it still is not as clean and simple as it should be. The goal is to write this like we would write a normal template, create a variadic pack called `Ts` and use this with `std::forward`. No **`decltype`** necessary. In C++20, all we have to do is to add the template-head and switch from **`auto`** as parameter type to the chosen `Ts`:

```cpp
auto getNamedLogger(const std::string origin)
{
  return [=]<typename... Ts>(Ts... args)
  {
    print(origin, std::forward<Ts>(args)...);
  };
}
```

Listing 7.17

Now, we have a version of `getNamedLogger` with all the powers of templates and lambda in one. It is clean and easy to write and read.

7.5 Pack expansions in lambda init-captures

In the previous example of `getNamedLogger` we assumed that only one parameter is passed when creating the logging function `origin`. This is a good pattern for a lot of use-cases, but what if `getNamedLogger` should accept more than one parameter? What if the number of parameters varies? Right, this sounds like a variadic template version of `getNamedLogger`. This could be useful, if the origin comprises multiple items. For example, think about creating a logger for the brake system. The initial origin is ”brake”. If we like to distinguish whether the log message came from the left or the right side and from the front or back we need three variables for a logger: ”brake”, ”left”, ”front”. There are other components where the initial single parameter `origin` is enough. A car has only one steering wheel, so potentially this is a case where one parameter is sufficient.

To fulfil the new requirement `getNamedLogger` needs to become a variadic template itself. All the origins are then captured by the lambda inside `getNamedLogger` and the pack is expanded to the `print` function which the lambda executes. Here is a possible implementation:

```cpp
template<typename... Origins>
auto getNamedLogger(Origins... origins)
{
  return [=]<typename... Ts>(Ts... args)
```

Listing 7.18

```cpp
  {
    print(origins..., std::forward<Ts>(args)...);
  };
}
```

Listing 7.18

Which just a few tweaks we managed to add the new requirements to `getNamedLogger`. There is a but. In Subsection 7.4.3 we enhanced the former solution of `getNamedLogger` to perfectly forward the variadic arguments which are passed to the lambda to the `print`-function. This was for the arguments to the lambda itself. By making `getNamedLogger` a variadic template and let the lambda capture the parameter pack of `getNamedLogger` there is another opportunity to either waste resources or to apply perfect forwarding. Assume that either we passed larger `std::string` objects as origins, perfect forwarding will safe additional allocations. The same is true for any type which needs to allocate its internal resources. To manage our resources well, we have to find a way to move `origins` into the lambda. Or more precisely, forward the arguments as not every may be valid to be moved.

Prior to C++20 there was no straightforward solution. Most of us naturally tried it by applying `std::forward` to the parameter pack in the capture list of the lambda using an init-capture, like this:

```cpp
return [_origins=std::forward<Origins>(origins...)]
                                  <typename... Ts>(Ts... args)
```

7.2 C++14: Lambda init-capture

Lambda init-captures where introduced with C++14. They allow to create a new variable inside the lambda during creation of it in the capture-list. The type of such an init-capture is deduced by `auto`. It is unnecessary to choose a different name than the one which is used to initialize the init-capture.

```
int x = 3;

auto l1 = [y = x Ⓐ A lambda init-capture, creating a new
    variable y with x as value
] { return y; };

auto l2 = [x = x Ⓑ The same name of x inside the lambda
] { return x; }; Ⓒ Here x refers to the x of the lambda, not
    one in the outer scope
```

Listing 7.19

There are other variations possible. In the end, a workable C++17 solution was to use `std::tuple` inside `getNamedLogger`. The lambda uses an init-capture `tup` to capture a `std::tuple` which contains the moved `origins` Ⓐ. Inside this first lambda another lambda is required Ⓑ. This second lambda is supplied to `std::apply` inside our original lambda to expand all the values of the wrapping tuple. In its parameter list `_origins` it receives the tuple elements as a pack. In code this looked like this:

```
template<typename... Origins>
auto getNamedLogger(Origins... origins)
{
  Ⓐ Create an init-capture of tuple and move origins into it
  return [tup = std::make_tuple(std::forward<Origins>(
          origins)...)]<typename... Ts>(Ts... args)
  {
    std::apply(
      Ⓑ A second lambda which is applies to the tuple values
      [&](const auto&... _origins) {
        print(_origins..., std::forward<Ts>(args)...);
      },
      tup);
  };
}
```

Listing 7.20

It works, but it is a hard to explain the solution for a simple use-case. C++20 allows us to use the syntax I said is the natural one. Pack expansions in init-captures allow us absolutely perfect forwarding of parameters and captures in case of a lambda. No additional distracting elements are required. This makes lambdas fit even better in the language. We can use them like any other element.

```cpp
template<typename... Origins>
auto getNamedLogger(Origins... origins)
{
  return [... _origins = std::forward<Origins>(
          origins)]<typename... Ts>(Ts... args)
  {
    print(_origins..., std::forward<Ts>(args)...);
  };
}
```

Listing 7.21

We create a new pack in the lambda capture list and move or forward all arguments from the variadic function pack into the lambda. The only difference to the natural code I showed above is that we need to tell the compiler that `_origins` is a pack. To do that, we need to add the ellipsis before the init-capture making it `..._origins`. If you remember that init-captures are implicitly **`auto`** variables you can think of this as writing **`auto`**`... _origins` which is consistent with how we use the ellipsis in other places.

```cpp
auto steeringLogger = getNamedLogger("Steering"s);
steeringLogger("angle"s, 90);

auto brakeLogger = getNamedLogger("Brake"s, "Left"s, "Front"s);
brakeLogger("force", 40);
```

Listing 7.22

In Listing 7.21 we see how we can use `getNamedLogger`. With `steeringLogger` we create a logger for the steering and use it below with additional arguments `angle` and `90`. The second logger object is `brakeLogger`. Here we see multiple parameters passed to `getNamedLogger` during creation. After that we can invoke `brakeLogger` with multiple arguments.

7.6 Restricting lambdas with Concepts

Concepts apply to lambdas as well as to templates. We have seen that C++20 gives us lambdas with a template-head. Regarding Concepts this means that we have three places where a restriction can appear. A lambda with a template-head can restrict the template parameters by using a Concept instead of either `typename` or `class` to declare a parameter. As with templates, the requires-clause can appear after the template-head, the same way as for a regular template. And last but not least a lambda can have a trailing requires-clause.

How does this apply to `getNamedLogger`? For example, the arguments that are passed when constructing a logger, when `getNamedLogger` is called are stored in the lambda. Things get interesting when we look at life-time of the variables passed to `getNamedLogger` and `getNamedLogger` itself. The arguments we pass to `getNamedLogger` must have a longer life-time than `getNamedLogger`, otherwise we have UB. However, this only matters if the arguments are pointers or references, or if `getNamedLogger` stores the arguments as pointers or references.

The implementation of `getNamedLogger` store the arguments as they are passed in. Pointers will be stored as pointers and so on. To reduce the risk of UB due to pointers or references, we can disallow them as arguments to `getNamedLogger`. We can put this restriction at `getNamedLogger` itself or at the lambda. Here, we put it in the most narrow scope, the lambda itself. More specifically we like to restrict `Origins` in the trailing requires-clause of the lambda. The type-trait to check is is `is_pointer`. In this case, `Origins` is a pack, so we need to apply the check to each parameter in the pack. Therefore, we can use `std::disjunction`, a type-trait available since C++17. It performs a logical OR on all the arguments passed to it.

```
template<typename... Origins>
auto getNamedLogger(Origins... origins)
{
  return
    [... _origins =
        std::forward<Origins>(origins)]<typename... Ts>(Ts... args
            )
    Ⓐ Trailing requires with disjunction and is_pointer to limit Origins
      to no pointers
```

Listing 7.23

```
    requires(not std::disjunction_v<std::is_pointer<Origins>...>)
  {
    print(_origins..., std::forward<Ts>(args)...);
  };
}
```

Listing 7.23

Logging usually has to be fast, however formatting floating point number usually isn't. For this reason, let's limit `getNamedLogger` to accept only arguments which are not floating pointer numbers. The difference is that this time the restriction must apply to the arguments passed to the lambda when it is invoked, namely to `Ts`. This time we also must put the restriction on the lambda, as the arguments to it are unknown at creation time. This still leaves us with two options, let's not forget it is C++ we are talking about. We can either constraint `Ts` with a type-constraint or use a requires-clause.

```
return [... _origins = std::forward<Origins>(origins)]<typename↩
    ... Ts>
  Ⓐ Requires-clause with disjunction of is_floating_point to limit Ts
  requires(not std::disjunction_v<std::is_floating_point<Ts>...>)↩
      (
    Ts... args)
  Ⓑ Trailing requires with disjunction and is_pointer to limit Origins to
      no pointers
  requires(not std::disjunction_v<std::is_pointer<Origins>...>)
{
  print(_origins..., std::forward<Ts>(args)...);
};
```

Listing 7.24

The alternative, using a type-constraint, is possible too. There is a Concept `floating_point` available with the STL which is nearly what we want. Sadly, we need a `not_floating_point` Concept. Let's create one and call it `NotFloatingPoint`. We can use the Concept `floating_point` and negate it or `is_floating_point` we used in the requires-clause before, however that this time it is simpler. The Concept applies to each parameter in the pack automatically so we need not do it ourselves and can drop `std::disjunction`.

```
template<typename T>
concept NotFloatingPoint = not std::is_floating_point_v<T>;
```

Listing 7.25

All that is left to do once we have the `NotFloatingPoint` concept is to replace `typename` in the lambda template-head with it, and we are done.

```
return [... _origins = std::forward<Origins>(origins)]
  A Type-constraint NotFloatingPoint to restrict all parameters not to be
      floats
  <NotFloatingPoint... Ts>(Ts... args) requires(
    not std::disjunction_v<std::is_pointer<Origins>...>)
{
  print(_origins..., std::forward<Ts>(args)...);
};
```

Listing 7.26

As you can see, we can apply Concepts to lambdas as to a normal template. With them we constraint the lambda, disallowing floating point types and disallow pointers as arguments as well. Thanks to Concepts, we do not need to commit to one single type or a base class.

C++20 makes lambdas even more powerful. Some restrictions were lifted which makes lambdas appear more and more as first-class citizens. Captures where refined and Concepts integrate well with them. Table 7.1 summarizes the possible captures and in which Standard they were introduced.

Cliff notes

- C++20 makes lambdas blend in to the language even more.
- Perfect forwarding of lambda arguments is clean thanks to lambdas with template-head.
- With pack-expansions allowed in lambda init-captures even captures packs can be perfectly forwarded.
- Lambdas interact with Concepts the same way as every other element of C++.

Table 7.1: Lambda Captures

Capture	Description	11	14	17	20
`[]`	Empty lambda	X	X	X	X
`[foo]`	Copy `foo`	X	X	X	X
`[&foo]`	`foo` as reference	X	X	X	X
`[=]`	Copy all variables used in the lambda body	X	X	X	X
`[&]`	All variables used in the lambda body as reference	X	X	X	X
`[this]`	Data and members of the surrounding class as reference	X	X	X	X
`[*this]`	Data and members of the surrounding class as deep copy	-	-	X	X
`[=, *this]`	Copy all variables used in the lambda body, `this` as deep copy	-	-	X	X
`[=, this]`	Copy all variables used in the lambda body, `this` by reference	-	-	-	X
`[y = x]`	Create y as new variable initialized by x		X	X	X
`[...y = pack]`	Create y as new pack, initialized by pack x				X

Chapter 8

Aggregate initialization

Initialization is a huge topic in C++. The number of ways to initialize an object sometimes seems to be endless. The discussion you can have in classes about the proper way to initialize an object is similarly widespread. C++11 aimed to address this issue by introducing the so-called uniform initialization, using curly braces. Sadly some things still didn't work or lead to surprising results. C++20 takes another step to make the initialization forms more consistent and, hopefully, beginner friendlier.

8.1 What is an aggregate

An aggregate is an array or a class that mainly composes other types. Because this is the main purpose, it can only have, per C++20:

- user declared-constructors. Please note that with this rule it is also not possible to create an aggregate with a defaulted conversion constructor, which is marked `explicit`;
- only `public` non-`static` data members;
- only base classes and functions which are not `virtual`.

That aggregates can have base classes was an update that C++17 brought us.

Another relaxation happened in C++14. Since then aggregates can have equal or braced initializers for non-`static` data members, allowing us to initialize members directly in the aggregate as shown in the following listing:

```
struct Aggregate {
  int a = 5; A Equal in-class initializer
  int b{7}; B Braced in-class initializer
};
```
Listing 8.1

8.1 C++11: User-provided vs. user-declared

What is the difference between a user-provided and a user-declared special member? If we just use =`default` to retain a default constructor, it is user-declared. Once we also provide the implementation, it is called user-provided.

```
struct UserProvided {
  UserProvided()
  { /* ... */
  }
};

struct UserDeclared {
  UserDeclared() = default;
};
```
Listing 8.2

8.2 Designated initializers

Now that we reiterated what aggregates are let's talk about one interesting change C++20 made. To some extent, designate initializers are a C compatibility fix in an area where C and C++ were different for as long as C++ exists.

Have a look at the following example:

```
struct Point { A Point is an aggregate
  int y;
```
Listing 8.3

```
  int x;
  int z;
};

const Point p1{3, 0, 4};
```

Listing 8.3

In this example, `Point` (A) is an aggregate. We can initialize it with braced-initialization, as shown with `p1`. Listing 8.3 puts the definition of `Point` and the declaration and initialization of `p1` very close together, but in real-world code, such locality is rare. Now imagine that `Point` would be defined in a header file, and you only come across `p1`. Can you tell which value does initialize which member? Or, more precisely, which value has `y` in `p1`? For all those thinking `0`, take another look at the code. I tricked you! Instead of declaring the members in the usual order x, y, z, I used $\mathbf{y}, x, z$! Why? Because I can, and things like this happen in real life. My point is, just by looking at the initialization as it is provided, we cannot tell what gets initialized. The same is technically true for classes, but we usually work with constructors and can fix the initialization order of members.

So what can we do? Write coding-guidelines would be a step but not a really helpful one. The example above is just one example where a natural assumption was not met. Hundreds of others are still out there.

8.2.1 Designated initializers in C

For all of you who use to program in C, you know a solution. C has designated initializers for `struct`s. There we can write the following:

```
struct Point { (A) Point is an aggregate
  int y;
  int x;
  int z;
};

const Point p1{.y = 3, .x = 0, .z = 4};
```

Listing 8.4

The part `.x = 0` is called a designated initializer. It gives us the power to name the member we wish to initialize explicitly. For all those who are thinking and what's new about this, I use it in my C++ code all the time. Compilers provided this feature as a compiler-extension, but it was not part of the C++ standard. I remember years back, I worked on a new C++ project and had used designated initializers more or less all the time when I hit a compile error that took me long to understand. The compiler in the new project did not support designated initializers. Well, as that compiler was freshly on the market, its name was Clang, I assumed the implementation was incomplete. It took me a while to understand that GCC provided designated initializers are a compiler-extension and that it was not C++.

8.2.2 Designated initializers in C++20

The good news now is that C++20 brings us designated initializers in C++. The example I shared above works in C++20 without any compiler-extension. For those of you who know how they work in C, there are a couple of differences, mostly because C++ has objects with constructors.

- All or none. If we opt-in for initializing an aggregate with designated initializers, all values we provide must use the designated initializer syntax.
- The designated initializers must appear in the declaration order of the data members. The compiler evaluates them from left to right.
- Designators must be unique. In C you can list the same designator multiple times which seems to have no benefit in C++ and only causes questions like how many times is the constructor called for that designators type.
- We can use brace-or-equal initialization in C++, while C allows only equal initialization.
- When we need to nest designators, we need equal-or-brace initialization.

Please note that regardless of whether we use designated initializers, the curly braces surrounding all the initializers form a braced initialization, prohibiting narrowing.

Braced initialization

One advantage is braced initialization is that it prevents narrowing conversions at compile-time. With that, we cannot accidentally lose precision.
The second feature is that it always performs a default or zero initialization. This means that either a default constructor for an object and its subobjects is called or it is initialized with zero or an equivalent value. For example, the equivalent value for 0 for a pointer is `nullptr`.
The third feature of braced initialization is that it prevents the most vertexing parse problem. Below you see an example of the most vertexing parse issue.

```cpp
const Point p1();
```

Listing 8.5

Here we see an attempt to initialize `p1` using parenthesis. However, the compiler sees this as the declaration of a function with the name `p1` returning a `Point` that takes no arguments. There is no way to express with parentheses that we like an object to be default or zero-initialized.

One element that works in C but not in C++ is using designated initializers for array elements.

Let's go over the different forms we can use in code. We reuse `Point` and add another aggregate `NamedPoint`, which contains a `std::string` and a `Point`.

```cpp
struct Point {
  int y;
  int x = 2; Ⓐ Using in-class member initialization
  int z;
};

struct NamedPoint {
  std::string name;
  Point pt;
};

Ⓑ Initializing with designated initializers
const Point p0{3, 0, 4};

Ⓒ Initializing all members with designated initializers
const Point p1{.y = 3, .x = 0, .z = 4};
const Point p2{.y{3}, .x{0}, .z = 4};
```

Listing 8.6

```

D Initializing a subset with designated initializers
const Point p3{.y = 3, .z = 4};

E Different order as defined in Point will not compile
// const Point p4{.x = 0, .y = 3, .z = 4};

F Designated initializers appearers more than once, will not compile
// const Point p5{.y = 3, .y = 4};

G Nested designated initializers
const NamedPoint p6{.name = "zero", .pt{.y{0}, .z{0}}};

H Designated initializers for the outer aggregate
const NamedPoint p7{.name = "zero", .pt{0, 0, 0}};
```

Listing 8.6

First, this time we use in-class member initialization for the member x in A. We remember that this is allowed for aggregates since C++14.

Now our first initialization is the hopefully least surprising p0 in B. Here we initialize the aggregate without designated initializers.

In C, we first initialize all members of Point for p1 with designated initializers using the equal sign. However, we can use braced initialization as well, as shown in p2. Mixing the two initialization forms is possible as well.

Next, in D, p3 is initialized with only a subset of its members by designated initializers. This is especially helpful if an aggregate provides a default value for its members. In this case, x is default initialized with in-class member initialization to 2. The designated initializers allow us to omit x and only initialize all other members. This is a huge improvement over the earlier standards, and finally, in-class member initializers for aggregates make much more sense.

In E and F, we see two examples that do not compile. In the case of E, designated initializers' order does not match those in Point. Remember, I chose to put y before x and still kept this order. So sadly, we cannot use designated initializers to write the initialization in our natural order, whatever that may be. We have to match the order of the definition in the class. Then F illustrates a case where a member is named twice. This code will not compile as well. My personal opinion is that this is fine. What does that duplication mean anyway? The risk of a bug because a user wanted `y` to be `3` is none zero.

Finally, in G, we look at nested designated initializers. Because we started designated initialization with `.name`, we need to use it for `pt` as well. You can read it like we initialize `pt` and now use designated initializers in `pt` again to initialize only a subset of the available members. Of course, we don't need to use designated initializers for `pt`, as H shows.

8.2.3 Initializing a subset of an aggregate with designated initializers

It is always great to have a new language feature, but often the question arises: now what is the benefit of the feature, and what can I do with it. In the former section, we've already seen that they can help us initialize an aggregate with in-class member initializers. Let's take a closer look at what that implies. We once again reuse the `Point` aggregate the way it was presented in the former example, where it uses in-class member initialization for its member `x`. In that former example, we saw `p3`, which initializes only `y` and `z`, leaving `x` with the in-class member initialization value. You may have noticed it, `p3` was `const`. Without designated initializers to achieve the same result, we have to write the code like this:

Listing 8.7
```
Point p3{};
p3.y = 3;
p3.z = 4;
```

We need to create a non-`const` version of `p3` and then initialize the desired members. By doing that, not only we lost the `const`ness, it is also no longer possible to create this variable entirely in global scope as a `static` variable, for example. Of course, the other solution is the keep the `const`ness and duplicate the values of the in-class member initialization by specifying all members:

Listing 8.8
```
const Point p3{3, 0, 4};
```

If we initialize all the members of the aggregate, we achieve the same result as with the designated initializers, as we can see in Listing 8.8. So there is only a minor advantage. Oh, wait! Did you see it? Naturally, I initialized `x` with 0, but the in-class member initialization uses 2! What is correct now? Whenever we approach such a piece of code, we cannot easily tell. In this case, I made the *mistake* on purpose. The code will compile but has the wrong values. My point is that designated initializers help us to list *only* the required members and let the other use their defaults. By that and listing the designated initializers such mistakes get effectively reduced.

8.2.4 Initialize a subset with designated initializers without in-class member initializers

Defaults are a good next question. What if I would not have provided a default for x as an in-class member initializer and leave that member out from the designated initializers? Like this:

```cpp
struct Point {
  int y;
  int x; A No in-class member initializer
  int z;
};

B Initializing a subset with designated initializers
const Point p3{.y = 3, .z = 4};
```

Listing 8.9

What is the value of x of p3? The answer is, as the surrounding braces are curly braces, we are looking at braced-initialization which ensures, that all unnamed members are initialized with default or zero initialization. In the case here, we have an `int` , it is zero initialization. Even with designated initializers, we have no uninitialized members. Whether the value an unnamed member has after initialization matches our expectation is a different question.

Utilizing Return Value Optimization thanks to designated initializers

Return value optimization

The term Return value optimization (RVO) refers to an old optimization technique of compilers. Have a look at the following example:

```
struct NonCopyableOrMoveable { A Type is neither copy nor
    movable
  NonCopyableOrMoveable() = default;
  NonCopyableOrMoveable(const NonCopyableOrMoveable&) = ↩
      delete;
  NonCopyableOrMoveable(NonCopyableOrMoveable&&) = delete;
  NonCopyableOrMoveable& operator=(const ↩
      NonCopyableOrMoveable&) = delete;
  NonCopyableOrMoveable& operator=(NonCopyableOrMoveable↩
      &&) = delete;
  ~NonCopyableOrMoveable() = default;
};

NonCopyableOrMoveable RVO()
{
  return {}; B This is where RVO happens
}

void Use()
{
  C The return-object is created directly at myValue
  auto myValue = RVO();
}
```

Listing 8.10

We have a struct `NonCopyableOrMoveable` A which, as the name indicates, is neither copyable nor movable because we deleted the required special members. Yet this code compiles fine. The reason is that the compiler performs an optimization, instead of creating the return-object on the stack in the function `RVO` B it creates it at the place where the return-object is assigned C. That way it saves us a potentially expensive assignment operation. Since C++17 this optimization is mandatory.

We can take advantage of designated initializers to get the benefits of RVO by creating an object in the `return`-statement, as illustrated in A.

```
Point GetThePoint() A Returning a Point utilizing RVO
{
```

Listing 8.11

```
  return {.y = 3, .z = 4};
}
```

Listing 8.11

Another application is when pushing elements to a `std::vector` or another STL container.

```
auto GetVectorOfPoints()
{
  std::vector<Point> points{};

  points.emplace_back(
    Point{.y = 5, .z = 6}); A Create a Point in-place of a container

  return points;
}
```

Listing 8.12

The pattern in A is the same as for the `return`-statement. Thanks to designated initializers, we can create the object with only a subset of its members initialized directly in a `emplace_back` call.

One additional advantage is readability. By explicitly naming the members and their values, code can become more readable and less error-prone.

8.2.5 Named arguments in C++: Aggregates with designated initializers

The advantage we saw above with `const` that we can create and initialize an aggregate thanks to designated initializers with a single statement has more advantages. Above I only talked about making the object `const`, but there are more places where the creation of an object in a single statement is beneficial.

Consider the following function `FileAccess`, which opens a file either for writing or only for reading and, can close it:

```cpp
void FileAccess(bool open, bool close, bool readonly);

void Use()
{
  FileAccess(true, false, true);
}
```

Listing 8.13

Above along with the function, we see with `Use` how calling that function looks. I doubt that someone can tell which `bool` is for what and if the values make sense here. Think about a code-review situation where you have to review that code and sign it off for production. There are coding styles that require you to put the parameter's name as a comment next to the value. LLVM, for example, does this. The using code then looks the following:

```cpp
FileAccess(/*open*/ true, /*close*/ false, /*readonly*/ true);
```

Listing 8.14

The compiler ignores comments. They may get wrong over time. Maintaining all the call-sites and ensuring that the parameter comments there still reflect the current situation is hard. Reading such inline comments as above is hard for me, but that may be just me because I'm not used to this style. The fact that the comments can get wrong over time is way more important.

What if we, instead of having three individual parameters, all of the type `bool`, provide a single aggregate `FileAccessParameters` which holds all the parameters as members? Then `FileAccess` takes that aggregate as a parameter like so:

```cpp
struct FileAccessParameters {
  bool open;
  bool close;
  bool readonly;
};

void FileAccess(const FileAccessParameters& params);
```

Listing 8.15

At the calling-side, we can, of course, just pass a temporary `FileAccessParameters` object, which we create as a parameter of the function. That way, we have won nothing. However, we can now use designated initializers when creating the object. Here is this version:

```
FileAccess({.open = true, .close = false, .readonly = true});
```

Listing 8.16

The code becomes clear. No unnecessary comments in the code which, may be wrong. Using an aggregate as a function parameter together with designated initializers can be seen as the poor men's solution to named arguments in C++.

An aggregate with designated initializers can be a nice way to model optional arguments for a function. While with default parameters, we always have to fill them from left to right, regardless of whether we like to set the first argument, we can omit members of an aggregate as we have seen. This increases flexibility in such cases.

For a large number of parameters, I think aggregates with designated initializers are a good option. However, I also like to point out here that using strong types instead of `bool`s all over the place can increase your code's robustness even more.

8.2.6 Overload resolution and designated initializers

Another element C++ has that C doesn't is overload resolution. This happens if we have a function that takes an aggregate, and we decide to use curly braces only and do not name the type explicitly. Here is an example that illustrates this situation:

```
struct Point2D {
  int X; A Note that this is a capital X
  int y;
};

B Two functions which overload
void Add(const Point& p, int v);
void Add(const Point2D& p, int v);

void Main()
{
  Add({.X = 3, .y = 4}, 3); C Fine, selects Point2D
  Add({.y = 4, .x = 3}, 3); D Fine, selects Point

  E Will not compile as soley .y is ambiguous
```

Listing 8.17

```
  // Add({.y = 4}, 3);
}
```

Listing 8.17

Here `Point` is a before with the three members `y, x, z`. The two functions, `Add` (B), are for either `Point` or a new aggregate `Point2D`. The latter one uses a capital `X` (A). If we now use the three different ways to call `Add` as shown in (C) and (D), they do work because of `X` and `x`. This helps the compiler to resolve to which type we refer. Without that, the overloads are ambiguous, as (E) illustrates. With just `.y`, which is fine as a designated initializer, the compiler finds `Point` as well as `Point2D` as a possible type to create when calling `Add`. This is an ambiguity the compiler cannot resolve. Simplified, if we are able to distinguish which type is meant by looking at the initialization, the compiler is as well.

The reason for this behavior here is that during overload resolution, the order of the designators doesn't matter. This is why it does not help us to start with `.x=3, .y=4` to tell the compiler that we like a `Point2D` created.

8.3 Direct-initialization for aggregates

C++20 makes initialization a bit more consistent, although it is still a C++ topic that can easily fill books. Since C++11, we have braced initialization for a lot of cases in addition to parentheses. Since then, there is always a debate on which way to use and which is the best. It is a bit unfortunate that the curly braces initialization is also referred to as uniform initialization. The dream at the time was to provide programmers with only one form of initialization that supersedes all the others and is uniformly usable. It turned out to be only a dream.

8.3.1 Initialization forms: Braced or parenthesis initialization

Braced initialization has the benefits of preventing narrowing conversions and performing a default or zero initialization. But it also has some drawbacks. Here is a popular one:

```
std::vector<int> v1(35); (A) Using parentheses
std::vector<int> v2{35}; (B) Using curly braces
```

Listing 8.18

In the example, we see an attempt to initialize a `std::vector` with parentheses in Ⓐ and curly braces in Ⓑ. The first initialization creates a `std::vector` with 35 elements all set to zero. The second form creates a `std::vector` with 1 element having the value 35. We can conclude that this is a minor difference for an expected program flow. The reason is that braced initialization is also referred to as list initialization. It has the ability to trigger a constructor, which takes a `std::initializer_list`. `std::vector` is just one example where this leads to huge confusion, and it is hard to see through.

There is another slightly more subtle case. We've already played with the `Point` aggregate. Let's now assume we like to create a `unique_ptr` of `Point`. The knowledgeable programmers that we are, we know to always use `std::make_unique`. Ok, that may be a bit to boring for you, stay with me for another minute. Here is the code I expect you all know and have written dozens of times:

Listing 8.19

```cpp
auto pt = std::make_unique<Point>(4, 5);
```

Now the question is, not what it does, but does it compile? Don't search for missing semicolons or stuff like this. The C++ grammar is correct. Yet the answer from your beloved compiler in pre-C++20 mode is a nice but hard to read error message:

```
In file included from <source>:1:
include/c++/v1/memory:2793:32: error: no matching constructor for ↩
    initialization of 'Point'
    return unique_ptr<_Tp>(new _Tp(_VSTD::forward<_Args>(__args)...))↩
        ;
                               ^    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:11:18: note: in instantiation of   function template ↩
    specialization 'std::make_unique<Point, int, int>' requested here
  auto pt = std::make_unique<Point>(4, 5);
                 ^
<source>:3:8: note: candidate constructor (the implicit copy ↩
    constructor) not viable: requires 1 argument, but 2 were provided
struct Point {
       ^
<source>:3:8: note: candidate constructor (the implicit move ↩
    constructor) not viable: requires 1 argument, but 2 were provided
<source>:3:8: note: candidate constructor (the implicit default ↩
    constructor) not viable: requires 0 arguments, but 2 were ↩
    provided
1 error generated.
Compiler returned: 1
```

It puzzled me many times to recall what I did wrong again. The reason for this error is, that internally when initializing `Point` `make_unique` uses parentheses to initialize the freshly created object. You can reduce it to the following line of code:

```cpp
const auto* pt = new Point(4, 5);
```

Listing 8.20

It is just that the rules of C++17 and prior do not allow parentheses for aggregate initialization and what we have here is an aggregate. The lack of parentheses for an initializer list makes it impossible to easily write a generic perfect forwarding function.

Generic perfect forwarding

If you desire to write a generic perfect forwarding function pre C++17, you can use `std::is_contructible` in a `constexpr if`. You can use parenthesis in the true branch, and in the false branch, you use curly braces. However, the drawback is that it now becomes very hard to tell which constructor is invoked, the usual one or the one taking a `std::initializer_list`.

I saw cases where people falsely concluded from that message that they should provide a constructor for their type. It works but for the wrong reason. The type with a user-provided constructor is no longer an aggregate.

C++20 adds support for aggregate initialization from a parenthesized list and makes the code above work without additional adjustments. It does initialize the aggregate nearly like a braced initializer list. Except that narrowing conversions are possible. We can say that what parentheses initialization does remains the same. We just got more places where we can use it. That said, the most vertexing parse issue still remains in C++20.

In Listing 8.21, you see an overview of the different initialization forms and their results in C++20.

```cpp
struct Point {
  int y;
  int x;
  int z;
};

struct Nested {
  int i;
```

Listing 8.21

```
  Point pt;
};

struct LifeTimeExtension {
  int&& r; Ⓐ Notice the r-value reference
};

Ⓑ Initialization of an aggregate
Point bPt{2, 3, 5};
Point pPt(2, 3, 5); Ⓒ Since C++20

Ⓓ Initialization of an array
int bArray[]{2, 3, 5};
int pArray[](2, 3, 5); Ⓔ Since C++20

Nested bNested{
  9,
  {3, 4, 5}}; Ⓕ Initialization of nested aggregate with nested braces
// Nested pNested(9,( 3,4,5));Ⓖ Nested parentheses are a different thing

Point bDesignated{.y = 3}; Ⓗ Designated initializers works
// Point pDesignated(.y=3);Ⓘ Designated initializers are not supported

// Point bNarrowing{3.5};Ⓙ Does not allow narrowing
Point pNarrowing(3.5); Ⓚ Allows narrowing

Point bValueInit{}; Ⓛ Default or zero initialization
Point pValueInit(); Ⓜ Still a function declaration

Ⓝ Initialization of a built-in type
int bBuiltIn{4};
int pBuiltIn(4);

LifeTimeExtension bTemporary{4}; Ⓞ Ok
LifeTimeExtension pTemporary(4); Ⓟ Dangling reference
```

Listing 8.21

8.3.2 Aggregates with user-declared constructors

At the beginning of this chapter, we saw the definition of aggregates in C++20, so you may wonder about this subsection. Careful readers have noticed that in C++20, `struct`s or `class`es, as well as `union`s, with user-declared constructors are no longer qualify as aggregates. We are looking at a breaking change here. Classes, `struct`s, or `union`s with user-declared constructors did classify as aggregates in C++17. Consider the following code and remember we are talking about C++17, your existing code-base:

```
struct NotDefaultConstructible {
  int x;

  Ⓐ Prevent default construction
  NotDefaultConstructible() = delete;
};
```

Listing 8.22

The `struct` `NotDefaultConstructible` yells with its name and the delete default constructor in Ⓐ at us that objects of this type are not default constructible. The name and the action are in sync here. One reason for such code is to prevent users from creating uninitialized objects on the stack with the intention to initialize the object later properly. However, then some value gets forgotten, and you spend a day or more chasing a very wired bug that changes all the time slightly, depending on the random values on the stack. That is time not well spent. Now, after such a chase, we come up with `NotDefaultConstructible`. In reality, we cannot name all our types with this name, but I like to overstate it here a little. More important than the name of the type is the deleted default constructor in Ⓐ. Below are two attempts to create an object of type `NotDefaultConstructible` on the stack without initializing `x` explicitly. Look at it and answer the question, what do you think Ⓒ does?

```
// NotDefaultConstructible ndc1; Ⓑ This statement does not compile
     as intended

Ⓒ What do you think does this statement does?
NotDefaultConstructible ndc2{};
```

Listing 8.23

So hands up, who thinks that ndc2 does rightfully not compile? We did all we could to make it fail at compile-time, so what can possibly go wrong? Well, according to C++17s rules for aggregates, the braced initialization finds a way to default initialize ndc2. But wait, you say, there is always the C++98 trick making the special member `private`. The code below now does for sure not compile, right?

Listing 8.24

```
struct NotDefaultConstructible {
  int x;

private:
  Ⓐ Prevent default construction
  NotDefaultConstructible() = delete;
};

// NotDefaultConstructible ndc1; Ⓑ This statement does not compile
    as intended

Ⓒ What do you think does this statement does?
NotDefaultConstructible ndc2{};
```

Well, do you really think that it doesn't compile, or is that more a wish? It is the latter. Even with a deleted `private` constructor, Ⓒ does initialize ndc2. Why, you ask? Because braced initialization for aggregates can bypass the deleted constructor in C++17. I spare you the entire rules. Just take my word. This is the case. That the name of the types does not matter to the compiler is granted, but that our attempt to delete the default constructor does work for parenthesis but not for braced initialization is just unfortunate. This C++17 behavior caused a lot of surprised faces or laughter in my previous training classes.

The behavior is especially unfortunate as there were forces saying that everybody should use braced initialization, which is also called uniform initialization, which should always work. Well, from some point of view, we can say it did in C++17. C++20 took the liberty to fix this bizzare case, which is not backward compatible. Should you, by accident, have such code in your code-base, it will stop working in C++20, and you will get a compile error.

The C++20 behavior making both initialization forms produce the same result is more consistent. In addition, C++20 reduces the gap between parenthesis and braced initialization. The way it is done is that the type `NotDefaultConstructible` is no aggregate in C++20.

8.4 Class Template Argument Deduction for aggregates

8.2 C++17: Class Template Argument Deduction

Class Template Argument Deduction (CTAD) is a feature introduced with C++17 that makes the compiler deduce class template arguments as it does for function template parameters. In the listing below, we can create a `std::vector` without specifying the type of the vector ourselves. The compiler deduces the type from the arguments provides to initialize the `std::vector`:

```cpp
std::vector a{2, 3, 4}; Ⓐ Using std::vector without specifying
    a type
```

Listing 8.25

Imagine a situation where you are working with POSIX functions from C++ code. Say we like to use `open` to open a file. The `open`-call may fail for various reasons, including the file does not exist or the process does not have permission to open it. All these conditions are signaled by the return value, which is `-1` if the `open`-call fails, or a valid file descriptor otherwise. If the call to `open` fails, there are situations where we like to know the exact reason. If so, we need to inspect the global variable `errno`. Since C++11 `errno` is thread-safe, that reduces the number of issues in a multi-threaded program.

But how do we transport the result and the `errno` value at the time up our call-stack if we do not like to sprinkle the POSIX API all over our code-base? Say we like to create an abstraction function `Open` which calls on POSIX platforms `open`. One approach for the return-type of `Open` is to provide an aggregate, which consists of an entry for `errno` and one holding the actual return value. Let's call this aggregate `ReturnCode`.

```cpp
struct ReturnCode {
  int returnCode;
  int err;
};
```

Listing 8.26

```

auto Open(const char* fileName)
{
  return ReturnCode{open(fileName), errno};
}
```

Listing 8.26

That looks good. Now that we have an abstraction for the POSIX `open` let's create another one for `read`. The interface of `read` is much like the one of `close`, except that it takes a file-descriptor instead of a filename and a buffer to read the data to. Oh, and there is one more slight difference, the return-type of `open` is `ssize_t`, not `int`. The one can be larger than the other. To be type-safe here, we need another aggregate for `read`. If well look further, for example, `lseek` we find another return-type `off_t`. Of course, we can create we new aggregate for each of these functions, but a more generic approach would be a good thing. Let us make the return-value in `ReturnCode` a type template parameter and with that `ReturnCode` a template. The change is easy:

```
template<typename T>
struct ReturnCode {
  T returnCode; A Make the return code's type generic
  int err;
};

auto Open(const char* fileName)
{
  B We need to specify the type of the template parameter
  return ReturnCode<int>{open(fileName), errno};
}
```

Listing 8.27

While the change itself is easy, you can see an additional change in Listing 8.27. We need now to specify the type of that template parameter. If we do not do that, we get the following error from the compiler:

```
<source>:18:10: error: no viable constructor or deduction guide for ↩
    deduction of template arguments of 'ReturnCode'
  return ReturnCode{open(fileName), errno};
         ^
```

```
<source>:10:8: note: candidate function template not viable: requires↩
     1 argument, but 2 were provided
struct ReturnCode {
       ^
<source>:10:8: note: candidate function template not viable: requires↩
     0 arguments, but 2 were provided
1 error generated.
ASM generation compiler returned: 1
<source>:18:10: error: no viable constructor or deduction guide for ↩
    deduction of template arguments of 'ReturnCode'
  return ReturnCode{open(fileName), errno};
         ^
<source>:10:8: note: candidate function template not viable: requires↩
     1 argument, but 2 were provided
struct ReturnCode {
       ^
<source>:10:8: note: candidate function template not viable: requires↩
     0 arguments, but 2 were provided
1 error generated.
Execution build compiler returned: 1
```

Specifying the type explicitly is for sure one approach, but if we think in generic code where that type may change with the platforms `open` equivalent having to name the type in the wrapper is not what I want. We are looking for a simple solution, so no **`decltype`**ing. This is a situation for CTAD. We can write our own so-called *deduction guide* which tells the compiler how to deduce the type. In the error message above, we see that the compiler already talked about a deduction guide. This helps the compiler do exactly the same as it does when it comes to function templates. There the compiler can deduce the function templates parameters on its own. For easy cases and classes, the compiler is able to generate a so-called implicit deduction guide. This makes it easier for us as users. Most things work out of the box. However, in this case, we have an aggregate, and CTAD in C++17 works only for classes automatically. Here is a C++17 version with a user-provided deduction guide:

Listing 8.28

```cpp
template<typename T>
struct ReturnCode {
  T returnCode;
  int err;
};

```

```cpp
A Deduction guide
template<typename T>
ReturnCode(T, int) -> ReturnCode<T>;

auto Open(const char* fileName)
{
  B Works without specifying a type
  return ReturnCode{open(fileName), errno};
}
```

Listing 8.28

As you can see in A, we can now omit the type. This makes this whole construct much more generic.

8.3 C++17: Class Template Argument deduction guides

A deduction guide tells the compiler how to instantiate a class template. It is a hint for the class template argument deduction the compiler needs to perform, much like it did pre-C++17 for function template arguments already. The syntax is

```cpp
TypenName(Parameters) -> TypeName<Parameters>
```

I omitted the template-head above to make it more readable. You can read it like a function declaration with a trailing return type. Or you can see the first part as the constructor of a class, which tells with the arrow how to instantiate that class, give that the constructor is called with the set of parameters in that order.

However, the fact that we need to write this explicit deduction guide is a difference to classes. Let's try out an exercise. Say instead of using an aggregate, we use a class with private members, and we provide a constructor. For the sake of completeness, we then also provide the now necessary access functions:

```cpp
template<typename T>
struct ReturnCode {
  ReturnCode(T t, int e)
  : returnCode{t}
  , err{e}
  {}

  auto GetReturnCode() const { return returnCode; }
  int GetErrno() const { return err; }
```

Listing 8.29

```

private:
  T returnCode;
  int err;
};

auto Open(const char* fileName)
{
  B Works without specifying a type
  return ReturnCode{open(fileName), errno};
}
```

Listing 8.29

The compiler's reward for all our effort is that it now generates an implicit deduction guide for the version of `ReturnCode` modeled as a class. You cannot say all that effort was without a reward! But you can say that it is insane that we have to change the entire design just to get support from the compiler now for free. Well, the two presented ways are the options we had in C++17. And compared to the class version, providing an explicit deduction guide seems to be a much better choice. Without that asynchrony, probably nobody would complain or struggle with the two lines we have to write for that deduction guide. Then there is still the argument that less code is more, and the more code we can shift to the compiler, the less code we have to maintain, the less errors we carry in our code. This goes a long way if you consider future maintenance, bug fixing, and so on. Having the compiler to generate these for us is a win, and that is exactly what we now can get in C++20. Here is the C++20 version, back with an aggregate.

```
template<typename T>
struct ReturnCode {
  T returnCode;
  int err;
};

auto Open(const char* fileName)
{
  return ReturnCode{open(fileName), errno};
```

Listing 8.30

```
}
```

Listing 8.3

It looks exactly like the initial version we haven seen with the deduction guide, except that this time the compiler provides an implicit deduction guide for us, the same way as it already did for classes.

Cliff notes

- C++ designated initializers differ slightly from what is available in C.
- Designated initializers help us to keep a variable `const` and reduce code duplication by using existing in-class member initializers as defaults.
- Designated initializers from a braced initialization, which always gives us default or zero initialization for all unnamed members without in-class member initializers.
- C++ requires the designated initializers to be in the order as the members are declared in the aggregate.
- Designated initializers do not work with the new parenthesized initialization of aggregates. They always must be using curly braces on the outside.
- Designated initializers can be used to emulated named arguments in C++.
- C++17's deduction guides can help us omit type for class templates the same way as function templates. With C++20, the implicit deduction guides are generated the same way for both classes and aggregates by the compiler.

Chapter 9

Class-types as non-type template parameters

In this chapter, we will first look back at what NTTPs are and what we could do with them in the previous standards. With that knowledge, we will look at the new application areas with the lifted restrictions C++20 gives us.

9.1 What are non-type template parameters again

C++ has three different kinds of templates parameters:

- type template parameters
- non-type template parameters NTTP
- template template parameters

Probably the most common type is the first one, a type template parameter. We have already seen them in multiple places. For example, in Chapter 1 we used them at the start in the `Add` example. We later used them again in Section 5.6 when we built our own logging function. As the name already implies, we provide types for them and can use these types later. The compiler can also deduce the type of the arguments, for example, in the case of a function template. We saw in Section 8.4 that this is possible for classes since C++17 as well, and C++20 added support for aggregates.

NTTPs are the form of parameters where we provide values as template parameters. Numbers are a common way to parametrize a template. Below you see an example that uses both type and non-type template parameters. The example shows a short version of what you may know from the standard as `std::array`.

```cpp
template<typename T,  A A type template parameter
         size_t N>  B A non-type template parameter
struct Array {
  T data[N];
};

Array<int,  C Passing int as type parameter
      5>  D Passing 5 as non-type parameter
  myArray{};
```

Listing 9.1

This example shows a class template `Array`, which takes a type and non-type template parameter. `Array` uses these two to create an array of type `T` and size `N`. In C, we pass `int` as a type parameter. Here we are only allowed to pass types. Then in D, we pass `5` as an NTTP. A value as we can see, or even more precisely a constant. As templates are instantiated during compile-time, all the parameters we pass to them must be known at compile-time.

As we will not need template template parameters in this chapter, I skip an explanation for them here.

9.2 The requirements for class types as non-type template parameters

NTTPs always needed to be something constant. Before C++20, we could use integral or enumeration types. Both are known values at compile-time. Pointer to objects or pointers to function are allowed as well. These objects and the pointers to them are known at compile-time. We can also pass references to objects or functions.

For the first case, the object must have a `static` storage duration. `std::nullptr_t` was allowed as well. Prior to C++20 the missing forms are floating-point numbers and classes.

The reason for the limitations is that the compiler needs to compare two template instantiations and figure out whether they are equal.

As time has gone by and technologies have improved, by now encoding floating-point numbers in class template names for compiler vendors is feasible. For us programmers, this means that we can now use floating-point numbers as NTTPs like we could use `int` before.

One reason for the reservation all these years was to identify when two template arguments are equivalent. This equivalence is required to identify whether two instantiations of a template are the same. For integers, such equality is simple, compared to floating-point numbers. Two integers are equal if they have the same value. The same value for an integer also always implies the same bit-pattern. This is different for floating-point numbers. The differences start with the positive and negative zero a floating-point number can represent. Further, floating-point numbers are approximations. In some cases, the value differs slightly if a value is computed in different ways. For example:

$$0.6 - 0.2 \neq 0.4$$

Furthermore, floating-point numbers can represent infinite and NaN. The complicated thing is that there is not a single value that represents NaN. Several values can represent NaN. Infinite, on the other hand, can be positive or negative, so two different values.

All these properties of floating-point numbers make it difficult to use them as template arguments and get a consistent answer to whether two template instantiations are equal. One example is the number zero which can be positive or negative. Assume your application derives an action based on the sign, than the sign matters, even for the number zero.

The definition for floating-point template arguments in C++ is now that two template arguments are considered equivalent if their values are identical. In this case, value means bit-pattern. Here are some examples in code:

```
template<double x>
struct A; A A type which uses double as NTTP

B Different bit pattern results in different type
static_assert(not std::is_same_v<A<+0.6 - 0.2>, A<0.4>>);

C Different bit pattern results in different type
static_assert(not std::is_same_v<A<+0.0>, A<-0.0>>);

static_assert(+0.0 == -0.0); D IEEE 754 says they are equal

static_assert(
  std::is_same_v<A<0.1 + 0.1>, A<0.2>>); E Same bit pattern
```

Listing 9.2

At the top in A, we declare a template type which we use for our comparisons. In B, we see the same example as above where 0.6 – 0.2 is not equal to 0.4. The `static_assert`, together with `std::is_same` applied to `A`, our template type, verifies this. The same is true for the negative and positive zero in C. They have a different bit pattern and value. We are looking at something that may surprise some of you because the Institute of Electrical and Electronics Engineers (IEEE) standard defines that $-0.0 == +0.0$, as we can see in D. And lastly, in E, we see that if two calculations yield the same value, and with that bit-pattern, they are considered identical.

Keep this in mind when using floating-point types are NTTP.

9.3 Class types as non-type template parameters

Now that we can have class types as NTTPs, let's see what nice new things we can do with them. We need a literal type or, in our case, more appropriate a literal class. A class is a literal class if it has

- a `constexpr` destructor. Since C++20, the implicit destructor is `constexpr`;
- at least one `constexpr` constructor that is not a copy or move constructor. A closure type or an aggregate, which don't have a constructor, is also possible;

- only non-`volatile` non-static data members and base classes, which are literal types.

We can fulfill these requirements with, for example, a class that contains a `char` array and has a `constexpr` constructor that initializes this array. Like this:

```cpp
struct fixed_string {
  char data[5]{};

  constexpr fixed_string(const char* str) { std::copy_n(str, 5, ↩
      data); }
};
```

Listing 9.3

The type above, `fixed_string`, is a literal class type. Yet, we can say that `fixed_string` is a fairly unusable type. As presented, the size of the array is fixed. The way the constructor takes the argument does not allow to check whether the provides string is less than our `5` `char` array. While this is an example of a literal class, the result is quite unusable and unsafe.

9.3.1 A first contact with class types as NTTP

To make `fixed_string` usable, we need to make the type a template itself. We can still supply a class template, which is a literal class as an NTTP. That is still within the set of rules. How about this version:

```cpp
template<typename CharT, std::size_t N>
struct fixed_string {
  CharT data[N]{};

  constexpr fixed_string(const CharT (&str)[N])
  {
    std::copy_n(str, N, data);
  }
};
```

Listing 9.4

That looks much better. Our `fixed_string` now, thanks to the compiler, deduces the type and the size of the array we pass in the constructor. A potential initialization of `fixed_string` can be this:

```cpp
fixed_string fs{"Hello, C++20"};
```

Listing 9.5

Thanks to CTAD, we learned about that in Section 8.4, we do not need more code. The compiler figures out how to instantiate a `fixed_string` object.

Now we have our base-line. Let's see the new part and pass `fixed_string` as NTTP to a template. We use a class template called `FixStringContainer`, which takes a `fixed_string` and contains a method `print`. Guess what? This method prints out the contents of the string. Here it is:

```cpp
template<fixed_string Str> A Here we have a NTTP
struct FixedStringContainer {
  void print()
  {
    std::cout << Str.data << '\n'; B Use Str
  }
};

void Use()
{
  C We can instantiate the template with a string
  FixedStringContainer<"Hello, C++"> fc{};
  fc.print(); D For those who believe it only if they see it
}
```

Listing 9.6

In A, we see an NTTP. And for the first time, this NTTP is a class. We can use `Str`, the NTTP, inside `FixStringContainer` like any other NTTP. `Str` feels like a variable that is present inside `FixStringContainer`.

We can create an object of type `FixStringContainer` straight forward, as shown in C. Seeing a string between the angle brackets is probably unfamiliar. Finally then in D, the `print` function is invoked, printing the string. Now this shows an interesting mixture of compile-time and run-time. While the NTTP `Str` is supplied at compile-time, the template we access the contents later at run-time with `print`.

Before we move to a real-world example for good use of `fixed_string`, I first like to talk a briefly about compile-time data.

9.3.2 What compile-time data do we have

I often get the question why things should be made `constexpr`. The questions come from people who know and have understood the basics of `constexpr`. They know that the compiler can evaluate a constructor at compile-time in certain places, and with that setup an entire object at compile-time. The question is not about these details, people ask where to apply `constexpr`. Some say, if everything is `constexpr` and the compiler evaluates that at compile-time, we never need to run that program. We already have the answer. Totally true. We never have this situation. That is why people are looking for scenarios where `constexpr` makes sense.

When looking for places where we can meaningfully apply `constexpr`, we often end up with type-safe replacements for macros. Calculations of constants are another example. Sometimes this leads to an entire object created at compile-time. One thing that people often seem to overlook is, in my experience, all that data that is present in our code, and sometimes this data is directly there. No calculations. We have already seen multiple such cases. All the format strings in Chapter 5 are an example of data that is available at compile-time. Let's use that data in a value-adding way.

9.4 Building a format function with specifier count check

For a long time, something I wanted is a type-safe format or print function that checks at compile-time that the number of specifiers provided to match the number of arguments. I like to build such a function with you.

Why at compile-time? Because compile-time is the best place! At run-time, we need to ensure that our test-cases cover every use of a print function. As great as `std::format` is, the current design does the checking at run-time, acknowledging errors with an exception. I like to avoid exceptions as long as possible and instead find these kinds of errors as early as possible.

With `fixed_string`, we already have a good foundation for a static specifier checking `print` function.

What do we need to build such a function? Something like we've previously seen with `FixStringContainer` and a function that counts the identifiers. To make this implementation fit into a book, we use some simplifications. We assume `printf`-like identifiers starting with a percent sign. We further define that there are no escapes for the format string. With these requirements, we can build the following class template `FormatString`:

```
template<fixed_string Str> Ⓐ Takes the fixed string as NTTP
struct FormatString {
  static constexpr auto fmt =
    Str; Ⓑ Store the string for easy access

  Ⓒ Use ranges to count all the percent signs.
  static constexpr auto numArgs = std::ranges::count(fmt.data, ↩
      '%');

  Ⓓ For usability provide a conversion operator
  operator const auto *() const { return fmt.data; }
};
```

Listing 9.7

What do we have in these few lines of code? First, we see again `fixed_string` as NTTP, in Ⓐ. The class template `FormatString` then stores an instance of `fixed_string` in the form of a `static constexpr` variable as static member, as shown in Ⓑ. With the reduced specification for format specifiers, all we have to do to get the number of specifiers provided in a string is to count the percent signs. What better to use than an algorithm from the STL. Does using ranges also sound good to you? Well, then we use them. As we can see in Ⓒ, `numArgs` is used to count the number of percent signs in a string.

We see something that is not important for now, but there for the sake of usability, the conversion operator, as shown in Ⓓ. This is a good base. Now, how does the `print` function look?

9.4.1 A first `print` function

Our `print` function must be a variadic function template, that is for sure. `print` takes the format string as first argument, as usual. Just that in our case, this is not of type `const char*` but `FormatString`. That format string is followed by the optional arguments. To make our life easier, we use the abbreviated function syntax from Concepts, we have seen those in Section 1.10, and declare the first parameter with `auto`. Otherwise, we would have to write code that deduces `FormatString`. Then inside `print` in the first version, we simply call `printf`. This is how `print` looks:

```cpp
template<typename... Ts>
void print(auto fmt, const Ts&... ts)
{
  printf(fmt, ts...);
}
```

Listing 9.8

In Listing 9.8, we can now see the use of the conversion operator of `FormatString` to `const char*`. Without the conversion operator, we would have to pass `fmt.s`. Using `print` looks the following:

```cpp
print(FormatString<"%s, %s">{}, "Hello", "C++20");
```

Listing 9.9

Aside from that fact, that so far, we have only written a bunch of code, which does not yet improve anything, we now have also more to type when passing the format string to `print`. This is annoying. Improvements are fine, but they are not that great if they come with too many complications. Having to spell out the creating of a `FormatString` object all the time is such a complication. So before we move on with anything else, let's first figure out whether we can improve this. Otherwise, our solution would probably not accepted by users.

9.4.2 Optimizing the format string creation

Well, one obvious solution would be to shorten the name `FormatString` to a one or two-letter type name. Certainly doable, with a high chance of clashing with another type, created with the same shortening need. No, we don't do that. The name carries information. We just don't like to see and type this information at the call-side.

There is a much better solution available since C++11, and thanks to improvements in C++20, this solution is usable here. I talk about user-defined literals (UDLs). `_fs` seems like an appropriate short UDL

9.1 C++11: User-defined literals

User-defined literals allow use since C++11 to suffix data with a user-defined literal. This results than in the call of a special operator `operator""_myLiteral` which converts or constructs an object. They work only with constants like numbers or characters.

```cpp
template<fixed_string Str>
constexpr auto operator"" _fs()
{
  return FormatString<Str>{};
}
```

Listing 9.10

Here we can see that we once more use the new ability to provide a class type as an NTTP. For the UDL `_fs`, we define an operator template that takes a `fixed_string` and returns `auto`. Inside the operator, we return a new instance of `FormatString`, which gets the NTTP `Str` passed as a template argument.

```cpp
print("%s, %s"_fs, "Hello", "C++20");
```

Listing 9.11

This looks so much better. The long type name is gone as well as the curly braces. The solution with a UDL now looks like something that can be presented to users. So far, so good. The interface looks good, now let's see how to add the value-added part.

9.4.3 Checking the number of specifiers in a format string

With `FormatString` together with the UDL, we now have `_fs`, a nice way to create such an object and with `print` a variadic template that takes that as well as the format arguments. With `numArgs` in `FormatString`, we also already have the count of format specifiers in such an object. It is time to use this information. What better to use for such a check at compile-time than `static_assert`:

```cpp
template<typename... Ts>
void print(auto fmt, const Ts&... ts)
{
  A Use the count of arguments and compare it to the size of the pack
  static_assert(fmt.numArgs == sizeof...(Ts));
```

Listing 9.12

```

  printf(fmt, ts...);
}
```

Listing 9.12

In A, we see a `static_assert` that uses the static member `numArgs` in `FixedString` and compares `numArgs` to the number of elements in the parameter pack of `print`. Why does this check work, and simply passing a string doesn't in a `static_assert`? As you can see, `fmt` is a function parameter. You also likely know that parameters are never `constexpr`. Hence they cannot be used in a `static_assert`. Yet, I claim that the code above compiles. The reason why the code above compiles is that `fmt` only looks likes a function parameter, and ok, it is one too, but in the `static_assert`, we do not use the parameters run-time value. We use static members of the parameter's type! That is the difference here. You can compare the static members of `fmt` to static information in type-traits like `value` or `type`. Thanks to the design of `FormatString`, all the information we need is stored as a `static constexpr` value. This is *the* great thing about this! It looks like a function parameter, it can be passed like a function parameter, but it also contains static data members that are usable in a `static_assert`. This technique is not new. As I pointed out, it is the same as with type-traits. The awesome thing is that we can combine this existing technique in C++20 with strings. But, before I get too excited, let's move on.

9.4.4 Checking if type and specifiers do match

While the last section's achievement is already great, at least in my opinion, we can do more. From `std::format`, we know that verification whether a given specifier comes with a matching type, is possible. This is a check our `print` function should have as well.

We do take the easy road here once again to make the example fit into this chapter. The first thing we need to write our check is a helper function that knows which specifier belongs to which type. There are better ways to do that, less statically in a single function, but what I will present here gives you the picture.

```
template<typename T, typename U>
constexpr bool
```

Listing 9.13

```
  plain_same_v = Ⓐ Helper type-trait to strip the type down
  std::is_same_v<std::remove_cvref_t<T>, std::remove_cvref_t<U>>;

template<typename T>
constexpr static bool match(const char c)
{
  switch(c) { Ⓑ Our specifier to type mapping table
    case 'd': return plain_same_v<int, T>;
    case 'c': return plain_same_v<char, T>;
    case 'f': return plain_same_v<double, T>;
    case 's': Ⓒ Character strings are a bit more difficult
      return (plain_same_v<char, std::remove_all_extents_t<T>> &&
              std::is_array_v<T>) ||
             (plain_same_v<char*, std::remove_all_extents_t<T>> &&
              std::is_pointer_v<T>);
    default: return false;
  }
}
```

Listing 9.13

Above is a function template that takes a type template parameter and a `const char` character as a specifier. The type-trait `plain_same_v`, in Ⓐ, is a helper that uses `std::is_same_v` under the hood but removes all cv-qualifiers from the types because, at this point, we care only for the base type without any other qualification. If the pair, format specifier and type, is a match, the function `match` returns `true`, otherwise `false`. This is where we can add other specifier-type mappings. The way how `std::format` does this check, with class templates, is much more flexible, but I did like to keep this example short.

Now that we have that kind of mapping function, we need a way to get the specifier at a given index. This is a search for the percent character in the format string. We once again use the new ranges with `find` to find the needle in the format string. We do that in a loop so many times as the index provided.

```
template<int I, typename CharT>
constexpr auto get(const std::span<const CharT>& str)
{
  auto start = std::begin(str);
```

Listing 9.14

```
  const auto end = std::end(str);

  for(int i = 0; i <= I; ++i) { A Do it I-times
    B Find the next percent sign
    start = std::ranges::find(start, end, '%');
    ++start; C Without this we see the same percent sign
  }

  return *start; D Return the format specifier character
}
```

Listing 9.14

The function `get` is once again nothing special. We can implement `get` in C++11 mode, replacing `std::ranges::find` with `std::find`.

Now that we have some pieces at hand, the code that uses them is what we need next. Another function template, called `IsMatching`, to check that a specifier-type pair belongs together.

There are, as always in C++, many ways to write the function `IsMatching`. I will use a lambda with a template-head, as introduced in Subsection 7.4.1, together with `std::index_sequence`. This combination allows me to use fold-expressions and expand `match<Ts>(get<I>(str))` for each argument in the pack. Here you see the implementation:

```
template<typename CharT, typename... Ts>
constexpr bool IsMatching(std::span<const CharT> str)
{
  return [&]<size_t... I>(std::index_sequence<I...>)
  {
    return ((match<Ts>(get<I>(str))) && ...);
  }
  (std::make_index_sequence<sizeof...(Ts)>{});
}
```

Listing 9.15

I prefer the lambda way here and using a fold-expression. Instead, we can write a recursive function that splits up an argument each time and check that. The arguments are then checked from the most inner recursion to the outer, which takes more time for a large number of arguments than the fold-expression solution. Here

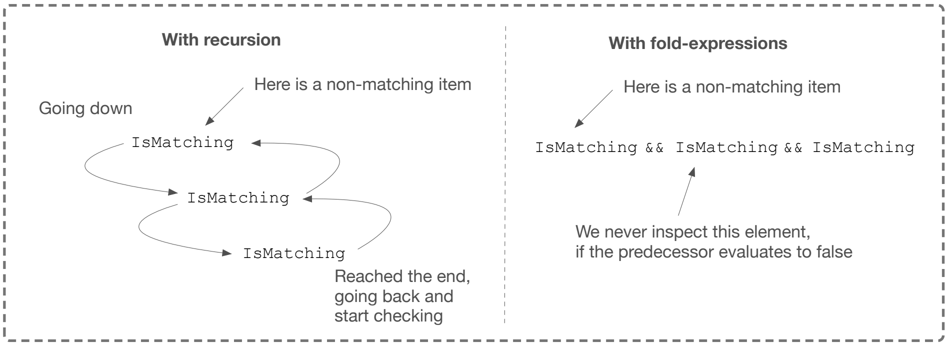

With recursion
Going down
Here is a non-matching item
IsMatching
IsMatching
IsMatching
Reached the end, going back and start checking
With fold-expressions
Here is a non-matching item
IsMatching && IsMatching && IsMatching
We never inspect this element, if the predecessor evaluates to false

Figure 9.1: Recursion vs. fold-expression to find the first non-matching item.

the check effectively ends as soon as the first element, going from the outermost to the innermost, isn't a match. Figure 9.1 illustrates the two approaches.

Excellent, we have all the building blocks we needed. All that is left for our specifier-matches-type-check is to use the parts in `print`:

Listing 9.16

```
template<typename... Ts>
void print(auto fmt, const Ts&... ts)
{
  Ⓐ Use the count of arguments and compare it to the size of the pack
  static_assert(fmt.numArgs == sizeof...(Ts));

  Ⓑ Check that specifier matches type
  static_assert(
    IsMatching<
      std::decay_t<decltype(fmt.fmt.data[0])>, Ⓒ Get the underlying
          type
      Ts... Ⓓ Expand the types pack
      >(fmt.fmt.data));

  printf(fmt, ts...);
}
```

As you can see, the specifier-matches-type-check is done with a second `static_assert`. That `static_assert` is a bit harder to read. First, when we call

`IsMatching`, we need to get the base type of `fixed_string`. But this time, with `fmt`, we don't have the type at hand. We need `decltype` to get the type of the first element of `fixed_string` in `FormatString`. Then we need, once again, to strip qualifiers and the arrayness from the type. That is the part we see in C. We can spare all that if we settle, for example, for `char` as type. However, I chose to use the generic approach. After we figured that type out, we supply the expanded pack of format arguments in D to `IsMatching`. Finally, we pass the `fixed_string` data to `IsMatching`. The reason why we have to provide the expanded format arguments not as function parameters is that the parameters are run-time values. However, their types aren't. Remember, at this point, it is all about types and not their contents.

Congratulations, we have a `print` function that checks at compile-time that:

- the number of format specifiers matches the number of arguments;
- all format specifiers match the corresponding provided argument.

Instant help by the compiler if we mix something up. This solution has another benefit. We now no longer need to build test-cases for each function using `print` to check whether a call with a specific parameter set throws an exception. Thanks to the compile-time part, everything is automatically checked every time and place when a `print` is used. The compiler must do that when instantiating `print`. That, of course, doesn't mean that you should stop writing unit-tests.

9.4.5 Enable more use-cases and prevent mistakes

What we have so far is excellent. Just, in a bigger project, there might be the need to provide a format string coming from a, for example, `char` buffer, or people try to call `print` with a `const char*`. With the current implementation, both will result in a probably hard to interpret compile error. Let's improve that. We start with the easier one, providing `print` for a `char` buffer. This is useful for cases when the format string is created during run-time.

Listing 9.17

```cpp
template<typename... Ts>
void print(char* s, const Ts&... ts)
{
  printf(s, ts...);
}
```

Listing 9.17

There is nothing special here. We just provide an overload for `print`, again a variadic template which passes all the data directly to `printf`. The part where we have to think briefly is, what type of format-string do we accept for that `print` overload? As you can already see in Listing 9.17 `print`, accepts a `char*`. At this point we assume, that all run-time created format-strings are not `const`.

The next step is to implement another overload for the case where people forget to add `_fs` our UDL. We like to prevent these accidents with a helpful error message. Below you see my choice of implementation.

```
template<typename... Ts>
constexpr bool always_false_v =
  false; Ⓐ Helper, returns always false

template<typename... Ts>
void print(const char* s, Ts&&... ts)
{
  Ⓑ Use the helper to trigger the assert whenever this template is
      instantiated
  static_assert(always_false_v<Ts...>, "Please use _fs");
}
```

Listing 9.18

In Ⓐ, we see a helper variable template, `always_false_v`, of which I wish we had it available in the STL. `always_false_v` ensures that regardless of what type of template arguments we pass to it, the variable gets set to `false` all the time. That could sound a bit crazy, I know. We see the use of `always_false_v` in a minute.

The `print` overload, again a variadic template, takes a `const char*` as format argument. The body of this overload contains only a `static_assert`, which informs users to add the UDL to their constant format string. In that `static_assert`, we also see our `always_false_v` helper to which the variadic parameter pack is passed. We need `always_false_v` here because otherwise, the `static_assert` would always trigger even when the compiler just parses the file in which this `print` is. `always_false_v` ensures that the `static_assert` fires only if `print` gets instantiated. We need to create a type dependency here, which `always_false_v` does.

And that's it! We now have a `print` function at our hands that does check at compile-time whether a format-specifier matches the provided type and that the number of format specifiers matches number of arguments provided.

Cliff notes

- NTTPs in C++20 are less restricted. We can pass literal class types as well as floating-point numbers as NTTP.
- Literal classes together with UDLs allow us to create compile-time strings passed as NTTPs.
- Floating-point types as NTTPs are compared bit-wise for equality.

Chapter 10

New STL elements

Chapter 11

Language Updates

Chapter 12

Doing (more) things at compile-time

Acronyms

ADL	Argument Dependent Lookup.
API	Application Programming Interface.
ASCII	American Standard Code for Information Interchange.
BCD	Binary Coded Digit.
CTAD	Class Template Argument Deduction.
DesDeMovA	Destructor defined Deleted Move Assignment.
FSM	finite state machine.
GP	Generic Programming.
IDE	Integrated Development Environment.
IEEE	Institute of Electrical and Electronics Engineers.
ISBN	International Standard Book Number.
MRN	Medical Record Number.
NaN	Not a Number.
NTTP	non-type template parameter.
OOP	Object Oriented Programming.
POSIX	Portable Operating System Interface.
RVO	Return value optimization.
SFINAE	substitution failure is not an error.
STL	Standard Template Library.
TCP/IP	Transmission Control Protocol / Internet Protocol.
TMP	Template Meta-Programming.
UB	Undefined Behavior.
UDL	user-defined literal.
UI	User Interface.

Bibliography

[1] H. Sutter, "GotW #94 Solution: AAA Style (Almost Always Auto)." [Online]. Available: https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

[2] D. Knuth, *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Pearson Education, 1997. [Online]. Available: https://books.google.de/books?id=x9AsAwAAQBAJ

[3] A. S. Tanenbaum, *Computer networks*, 5th ed. Prentice Hall PTR, 2011.

[4] P. Sommerlad, "C++Now 2019: Peter Sommerlad "Rule of DesDeMovA"." [Online]. Available: https://www.youtube.com/watch?v=fs4lIN3_IlA

Index