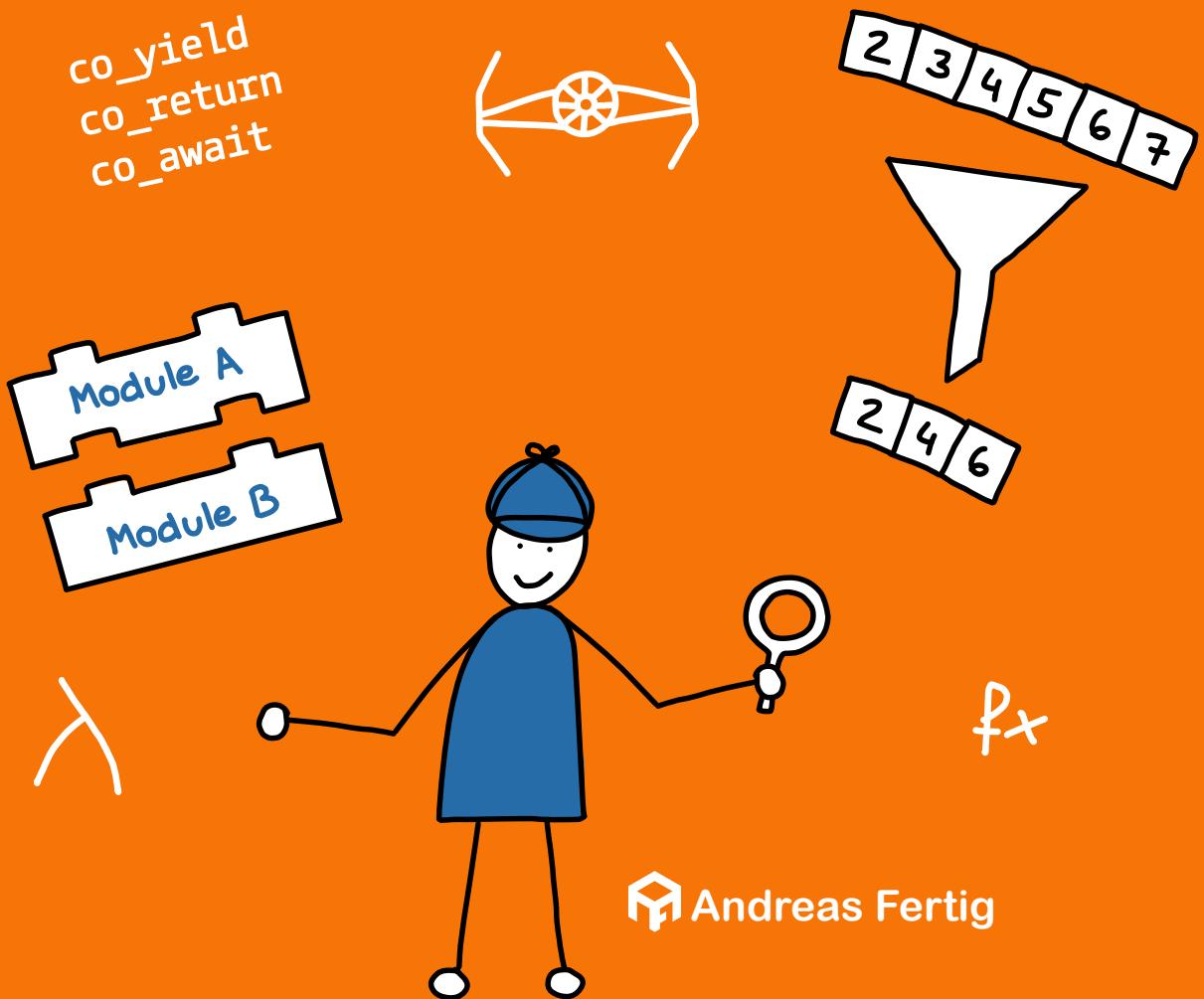


Programming with C++20

Concepts, Coroutines,
Ranges, and more



Andreas Fertig

Programming with C++20

Concepts, Coroutines, Ranges, and more

2. Edition



© 2024 Andreas Fertig
<https://AndreasFertig.com>
All rights reserved

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>.

The work including all its parts is protected by copyright. Any use outside the limits of the copyright law requires the prior consent of the author. This applies in particular to copying, editing, translating and saving and processing in electronic systems.

The reproduction of common names, trade names, trade names, etc. in this work does not justify the assumption that such names are to be regarded as free within the meaning of the trademark and trademark protection legislation and therefore may be used by everyone, even without special identification.

Planning and text:
Andreas Fertig

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

Published by:
Fertig Publications
<https://andreasfertig.com>

ISBN: 978-3-949323-05-8

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

Thank you, Brigitte & Karl! You opened the world for me. You were always there, supporting me, letting me find my own way, making me what I am.

*To Franziska, without her, I would not have accomplished this project. Never tired of reminding me of my talents, driving me when I'm tired, keeping my focus on. A lot of more could be written here. I like to close with:
Thank You!*

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 choosing the MIT license was to avoid uncertainty. It is a well-established open-source license and comes with few restrictions. That should make it easy to use it even in closed-source projects. If you need a dedicated license or have questions about the existing licensing, 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 would like to try out the code with the same compilers and revisions I used, here you go:

- GCC 13.2.0
- Clang 17.0.0

About the Author

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 23.

Andreas is involved in the C++ standardization committee, developing the new standards. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.

With C++ Insights (cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus to understand constructs even better.

Before training and consulting, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.

You can find him online at andreasfertig.com.

About the Book

Programming with C++20 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 features of C++20.

You will start with learning about the so-called big four Concepts, Coroutines, std::ranges, and modules. The big four followed by smaller yet not less important features. You will learn about std::format, the new way to format a string in C++. In chapter 6, you will learn about a new operator, the so-called spaceship operator, which makes you write less code.

You then will look at various improvements of the language, ensuring more consistency and reducing surprises. You will learn how lambdas improved in C++20 and what new elements you can now pass as non-type template parameters. Your next stop is the improvements to the STL.

Of course, you will not end this book without learning about what happened in the constexpr-world.

Style and conventions

The following shows the execution of a program. I used the Linux way here and skipped supplying the desired output name, resulting in a .out as the program name.

```
$ ./a.out  
Hello, Programming with C++20!
```

Output
6

- `<string>` stands for a header file named `string`.
- `[[xyz]]` marks a C++ attribute with the name `xyz`.

From time to time, I use an element from a previous standard after C++11. I explain these elements in dedicated standard boxes such as the following:

0.1 C++14: A sample element

A sample element from C++14.

These boxes carry the standard in which this element was introduced and are numbered such that I can reference them like this: Std-Box 0.1.

All listings are numbered and sometimes come with annotations which I refer to like this **A**.

References carry a page number in case the reference isn't on the same page. For example, Std-Box 0.1 has no page number because it appears on the same page.

Feedback

This book is published on Leanpub (<https://leanpub.com/programming-with-cpp20>) as a digital version. A full-color and a grayscale paperback version are available on Amazon. It helps if you indicate the book type you're referring to.

In any case, I appreciate your feedback. Please report it to me, whether it be a typo, a grammatical error, an issue with naming variables or functions, or another logical concern. You can send your feedback to books@andreasfertig.com.

PDF/Paperback vs. epub

As with most of my material, the book is written in \LaTeX . The epub version is generated by a custom script which first translates \LaTeX into Markdown and then translates Markdown into epub with the help of pandoc. This comes with some limitations. Currently, the bibliography does not use the same style as the PDF, and

the index is missing in the epub.

Another issue I have with the epub is that I do not own a reader device myself. I tested it with Apple's Books. However, please tell me if you have better knowledge of optimizing the output.

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, Jonathan Di Cosmo, and others.

A special thanks goes to Fran Buontempo, editor of ACCU's Overload magazine. She provided extensive feedback from the beginning and was never tired of pointing out some grammar issues along with poking to the base of my code examples, helping me 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)

2021-10-01: Added chapter 3. (Leanpub)

2021-10-15: Added chapter 10. (Leanpub)

2021-10-18: Various spelling and grammar updates due to feedback. Fixed listing numbers. Better styling of boxes (Leanpub)

2021-10-25: Added chapter 4. Various layout improvements. (Leanpub)

2021-10-27: Added chapter 11. Various layout improvements. (Leanpub)

2021-11-10: Added chapter 12, rewrote chapter 1. Various layout improvements. (Leanpub)

2021-11-26: Full release (Leanpub, Amazon)

2023-01-03: Various spelling and grammar updates due to feedback. (Leanpub)

2024-02-06: Prepareing second edition. (Leanpub, Print)

Table of Contents

1 Concepts: Predicates for strongly typed generic code	17
1.1 Programming before Concepts	18
1.2 Start using Concepts	21
1.3 Application areas for Concepts	22
1.4 The requires-expression: The runway for Concepts	23
1.5 Requirement kinds in a requires-expression	25
1.6 Ad hoc constraints	29
1.7 Defining a concept	30
1.8 Testing requirements	32
1.9 Abbreviated function template with auto as a generic parameter	34
1.10 Using a constexpr function in a concept	36
1.11 Concepts and constrained auto types	38
1.12 The power of Concepts: requires instead of enable_if	40
1.13 Concepts ordering	46
1.14 Improved error message	56
1.15 Existing Concepts	62
2 Coroutines: Suspending functions	65
2.1 Regular functions and their control flow	65
2.2 What are Coroutines	67
2.3 The Elements of Coroutines in C++	69
2.4 Writing a byte-stream parser the old way	78
2.5 A byte-stream parser with Coroutines	81
2.6 A different strategy of the Parse generator	91

2.7 Using a coroutine with custom new / delete	98
2.8 Using a coroutine with a custom allocator	100
2.9 Exceptions in coroutines	102
3 Ranges: The next-generation STL	107
3.1 Motivation	107
3.2 The who is who of ranges	114
3.3 A range	114
3.4 A range algorithm	116
3.5 A view into a range	117
3.6 A range adaptor	118
3.7 The new ranges namespaces	121
3.8 Ranges Concepts	122
3.9 Views	125
3.10 Creating a custom range	125
4 Modules: The superior way of includes	131
4.1 Background about the need for modules	131
4.2 Creating modules	133
4.3 Applying modules to an existing code base	136
5 std::format: Modern & type-safe text formatting	145
5.1 Formatting a string before C++20	145
5.2 Formatting a string using std::format	153
5.3 Formatting a custom type	158
5.4 Referring to a format argument	164
5.5 Using a custom buffer	165
5.6 Writing our own logging function	168
6 Three-way comparisons: Simplify your comparisons	175
6.1 Writing a class with equal comparison	176
6.2 Writing a class with ordering comparison, pre C++20	180
6.3 Writing a class with ordering comparison in C++20	183
6.4 The different comparison categories	186
6.5 Converting between comparison categories	190
6.6 New operator abilities: reverse and rewrite	192
6.7 The power of the default spaceship	193

6.8 Applying a custom sort order	196
6.9 Spaceship-operation interaction with existing code	197
7 Lambdas in C++20: New features	201
7.1 [=, this] as a lambda capture	201
7.2 Default-constructible lambdas	205
7.3 Captureless lambdas in unevaluated contexts	207
7.4 Lambdas in generic code	209
7.5 Pack expansions in lambda init-captures	214
7.6 Restricting lambdas with Concepts	217
8 Aggregates: Designated initializers and more	223
8.1 What is an aggregate	223
8.2 Designated initializers	224
8.3 Direct-initialization for aggregates	235
8.4 Class Template Argument Deduction for aggregates	241
9 Class-types as non-type template parameters	247
9.1 What are non-type template parameters again	247
9.2 The requirements for class types as non-type template parameters	248
9.3 Class types as non-type template parameters	250
9.4 Building a format function with specifier count check	253
10 New STL elements	265
10.1 <code>bit_cast</code> : Reinterpreting your objects	265
10.2 <code>endian</code> : Endianness detection at compile time	268
10.3 <code>to_array</code>	270
10.4 <code>span</code> : A view of continuous memory	272
10.5 <code>source_location</code> : The modern way for <code>__FUNCTION__</code>	275
10.6 <code>contains</code> for all associative containers	282
10.7 <code>starts_with</code> and <code>ends_with</code> for <code>std::string</code>	284
11 Language Updates	287
11.1 Range-based for-loops with initializers	287
11.2 New Attributes	291
11.3 <code>using enums</code>	294

11.4 conditional explicit	296
12 Doing (more) things at compile-time	301
12.1 The two worlds: compile- vs. run-time	301
12.2 <code>is_constant_evaluated</code> : Is this a <code>constexpr</code> -context?	305
12.3 Less restrictive <code>constexpr</code> -function requirements	309
12.4 Utilizing the new compile-time world: Sketching a car racing game	311
12.5 <code>consteval</code> : Do things guaranteed at compile-time	315
12.6 <code>constinit</code> : Initialize a non- <code>const</code> object at compile-time . . .	321
Acronyms	327
Bibliography	329
Index	331

Chapter 1

Concepts: Predicates for strongly typed generic code

Templates have been with C++ since the early beginnings. Recent standard updates have added new facilities, such as variadic templates. Templates enable Generic Programming (GP), the idea of abstracting concrete algorithms to get generic algorithms. They can then be combined and used with different types to produce a wide variety of software without providing 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 has always been a bit, well, clumsy. When we write a generic function, we only need to write the function once; then it can be applied to various different types. Sadly, when using the template with an unsupported type, finding any error requires a good understanding of the compiler's error message. All we needed to face such a compiler error message was a missing `operator<` that wasn't defined for the type. The issue was that we had no way of specifying the requirements to prevent the misuse, 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 ourselves, and there is a concept keyword. When I refer to the feature itself, it is spelled with a capital C, Concepts. The lowercase 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 return the result. Much like this:

```
1 const int x = Add(2, 3, 4, 5);
2 const int y = Add(2, 3);
3 const int z = Add(2, 3.0); A This should not compile
```

While the first two calls to Add, `x` and `y`, are fine, the third call should result in a compile error. With A we are looking at implicit conversions, namely a promotion from `int` to `double` because of `3.0`. Implicit conversions can be a good thing, but in this case, I prefer explicitness over the risk of loss of precision. Here Add should only accept an arbitrary number of values of the same data type.

To make the implementation a little more challenging, let's say that we don't want a `static_assert` in Add, which checks that all parameters are of the same type. We would like to have the option of providing an overload to Add that could handle certain cases of integer promotions.

We start with an implementation in C++1 to see the power of Concepts. For the implementation of Add, we obviously need a variadic function template as well as a couple of helpers. The implementation I present here requires two helpers, `are_same_v` and `first_arg_t`. You can see the implementation in Listing 1.1.

```
1 template<typename T, typename... Ts>
2 constexpr inline bool are_same_v =
3     std::conjunction_v<std::is_same<T, Ts>...>;
4
5 template<typename T, typename... Ts>
```

```

6  struct first_arg {
7      using type = T;
8  };
9
10 template<typename... Args>
11 using first_arg_t = typename first_arg<Args...>::type;

```

Listing 1.1

The job of `are_same_v`, which is a C++14 variable template (Std-Box 1.1), is to ensure, with the help of the type-trait `std::is_same` and `std::conjunction_v`, that all types in the parameter pack passed to `are_same_v` are the same. For that, the variable template uses the usual trick of splitting up the first argument from the pack and comparing all other arguments against this first one.

1.1 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 :

```

1 template<typename T>
2 constexpr T pi(3.14);

```

Listing 1.2

One other use case is to make TMP more readable. Whenever we had a type-trait with a value we wanted to access, before C++14, we needed to do this: `std::is_same<T, int>::value`. Admittedly, the `::value` part was not very appealing. Variable templates allow the value of `::value` to be stored in a variable.

```

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

```

Listing 1.3

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.

Our second helper, `first_arg_t`, uses a similar trick. It extracts the first type from a pack and stores it in a using-alias. That way, we have access to the first data type in a parameter pack, and since we later ensure that all types in the pack are the same, this first type is as good as that from any other index choice in the parameter pack.

Great, now that we have our helpers in place, let's implement Add. Listing 1.4 provides an implementation using C++17.

```

1 template<typename... Args>
2 std::enable_if_t<are_same_v<Args...>, first_arg_t<Args...>>
3 Add(const Args&... args) noexcept
4 {
5     return (... + args);
6 }
```

Listing 1.4

In this implementation, as promised, Add is a variadic function template, which we quickly spot looking at the template head. If we go down two lines, we can see the function's name, Add, and that it takes the parameter pack as a `const &`.

The body of Add reveals that I use C++17's fold expressions (Std-Box 1.2) to apply the plus operation to all the parameter packs elements.

1.2 C++17: Fold expressions

Before C++17, whenever we had a parameter pack, we needed to recursively call the function that 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. Assuming that the pack consists of three objects, this example will 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, I hope that's all understandable. The part I heavily object to, despite the fact that it is my own code, is the line with the `enable_if_t`. Yes, it is the state-of-the-art `enable_if` because, with the `_t`, we don't need to say `typename` in front of it. However, this single line is very hard to read and understand. Depending on your knowledge of C++, it can be easy, but remember the days when you started with C++. There is a lot that one has to learn to understand this single line.

The first part, or argument, is the condition. Here, we pass `are_same_v`. Should this condition be true, the next parameter which is `first_arg_t`, gets enabled. This then becomes the return type of Add. Right, did you also miss the return type initially?

Should the condition be false, then this entire expression isn't instantiable. We speak of substitution failure is not an error (SFINAE) as the technique used here, and this version of Add isn't used for lookups by the compilers. The result is that we can end up with page-long error messages where the compiler informs us about each and every overload of Add it tried.

One more subtle thing is that, in this case, `enable_if` does something slightly different than just enabling or disabling things. It tells us the requirements for this function. Yet, the name `enable_if` doesn't give many clues about that.

All these things are reasons why people might find templates tremendously difficult to process. But, yes, I know, those who stayed accommodated to all these shortcomings.

Now it is time to see how things change with C++20.

1.2 Start using Concepts

Sticking with the initial example, we ignore the helpers, as they stay the same. Listing 1.5 presents the C++20 implementation of Add.

```
1 template<typename... Args>
2     A Requires-clause using are_same_v to ensure all Args are of the same
3     type.
4     requires are_same_v<Args...>
5     auto Add(Args&... args) noexcept
6     {
7         return (... + args);
8     }
```

Listing 1.5

Here, we can see that Add remains a variadic function template, probably not the biggest surprise. Let's skip two lines again and go to the definition of Add. What first springs into our eyes is the return type. I chose `auto`. But the important thing is that the return type is there! The rest of the function's signature, as well as the function body, are unchanged. I see this return type as the first win. Before, the `enable_if` obfuscated the return type.

The biggest improvement is the line that says `requires`. Isn't that what's really going on here? This function Add requires that `are_same_v` is true. That's all. I find

that pretty easy to read. The intent is clearly expressed without obfuscating anything or requiring weird tricks. Okay, maybe we must look up what `are_same_v` does, but I can live with that.

We are looking at one of the building blocks of Concepts in Listing 1.5 on page 21, the requires-clause.

1.3 Application areas for Concepts

Before discussing how we can create Concepts, let's first see where we can apply them. Figure 1.1 on page 23 lists all the places in a template declaration where we can apply Concepts.

We see a type-constraint in C1. In this place, we can only use Concepts. We can use a type-constraint instead of either `class` or `typename` in a template-head to state as early as possible that this template takes a type deduced by the compiler, but it must meet some requirements.

The next option is with C2, using a requires-clause. We already applied that in our Add example in Listing 1.5 on page 21. In a requires-clause, we can use either concepts or type-trait. The expression following the `requires` must return a boolean value at compile time. If that value is true, the requirement(s) is (are) fulfilled.

The two places of C3 and C4 are similar. They both apply to placeholder types constraining them. We can also use Concepts to constrain `auto` variables, which we will see later. A constraint placeholder type works only with Concepts. Type-trait are not allowed. In C4, we see something that you might already know from C++14's generic lambdas, Std-Box 7.1 on page 210, `auto` as a parameter type. Since C++20, they are no longer limited to generic lambdas.

At the end, we have the trailing requires-clause. This one is similar to the requires-clause. We can use Concepts or type-trait and can use boolean logic to combine them. Table 1.1 on page 23 gives guidance on when to use which constraint form.

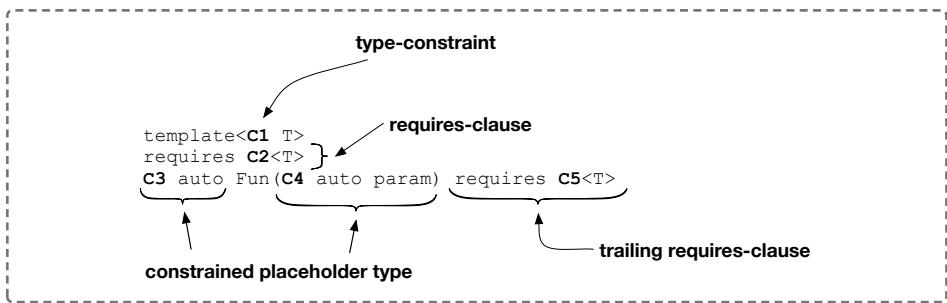


Figure 1.1: The different places where we can constrain a template or template argument.

Table 1.1: When to use which constraint form

	Type	When to use
C1	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 1.1 the type is limited to a floating-point type.
C2	requires-clause	Use this when you need to add constraints for multiple template type or non-type template parameters.
C5	trailing requires-clause	Use this for a method in a class template to constrain it based on the class template parameters.

1.4 The requires-expression: The runway for Concepts

We've already seen the two forms of a requires-clause. It is time to look at our Add example again and see what we can improve with the help of Concepts.

The current implementation of Add only prevents mixed types. Let's call this requirement **A** of Add. By that, the implementation leaves a lot unspecified:

- B** Add can nonsensically be called with only one parameter. The function's name, on the other hand, implies that things are added together. It would make more sense if Add would also require to be called with at least two parameters. Everything else is a performance waste.
- C** The type used in Args must support the `+` operation. This is a very subtle requirement that harshly yells at us once we violate it. It is also a design choice of

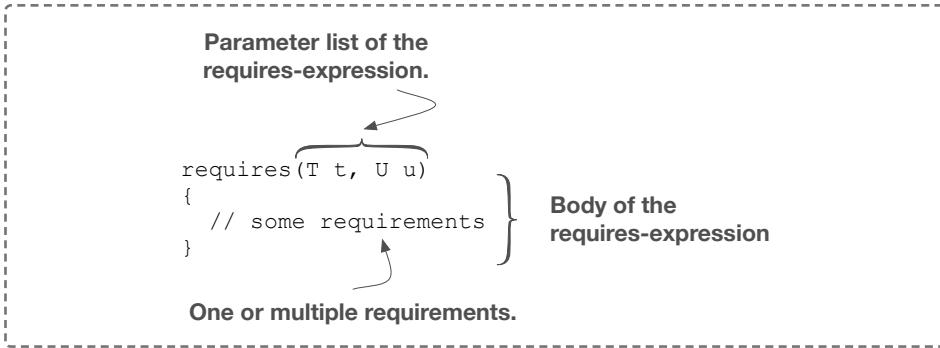


Figure 1.2: Parts of a requires-expression.

the implementor of `Add`. Instead of `operator+`, one could also require that the type comes with a member function `Addition`. That would, of course, rule out built-in types. Should we miss that, we again get these page-long errors that are hard to see through. Only documentation helps at this point, and documentation over time may disagree with the implementation. In such a case, I prefer a check by the compiler over documentation.

- D The operation `+` should be `noexcept`, since `Add` itself is `noexcept`. Did you spot that initially? The implementation of `Add` in Listing 1.5 on page 21 and before was always marked `noexcept`. Why? Because it mainly adds numbers, and I don't want to have a `try-catch-block` around something like `3 + 4`. But since `Add` is a generic function, it also works with, for example, a `std::string`, which can throw an exception. Writing a check for the `noexceptness` of `operator+ pre C++17` is an interesting exercise.
- E The return type of operation `+` should match that of `Args`. Another interesting and often overlooked requirement. It is surprising should `operator+ of type T` return a type `U`. Sometimes, there are good reasons for such a behavior, but it doesn't seem plausible in the case of `Add`. Let's restrict this as well.

Implementing all these requirements above the standard provides us with a requires-expression. Figure 1.2 shows how a requires-expression looks.

I read it like a constructor of type `requires`. You can also see it as a function without a return type. After the name `requires`, which is always the same, we have the optional parameter list. Like in a regular function, this parameter list can be

empty or can have entries. It is important to understand that a requires-expression is only used at compile-time to evaluate the different requirements it brings. It never enters our binary.

The parameters we can define in the parameter list of a requires-expression can consist of any available type, like `int`, `char`, `std::string`, or custom classes like `MyClass`. Because a requires-expression is always used in some kind of template context, we can also refer to template arguments, as Figure 1.2 on page 24 shows with `T` and `U`.

We are totally free when it comes to the qualifiers of these types. We can say that a requires-expression takes a `const T&` or a `const T*`. Well, we can even start entering the East and West `const` debate. Shaping these parameters helps us later when we refer to them in the body of a requires-expression which we will see in §1.12.1 on page 40.

Next in a requires-expression is the body, like with a function. This body comes with a requirement itself. It must contain at least one requirement.

The difference between a requires-clause and a requires-expression

A requires-clause (C2, C5) is a boolean expression. A requires-expression is more complicated. Much like we know it from `noexcept`. We can say that a function is `noexcept`, but we can also query whether a function is `noexcept`.

We can see a requires-clause like an `if` that evaluates a boolean expression, and a requires-expression returns such a boolean value.

1.5 Requirement kinds in a requires-expression

The body of a requires-expression can use four different requirements:

- Simple requirement (SR)
- Nested requirement (NR)
- Compound requirement (CR)
- Type requirement (TR)

The parenthesized letter pairs are for reference in the following code examples.

We will explore the first three requirements using Add, and the requirements we established in §1.4 on page 23. The type requirement doesn't make sense for Add. It is explained without using Add.

1.5.1 The simple requirement

As the name implies, a simple requirement checks for a simple thing, namely whether a certain statement is valid. For the Add example, Listing 1.6 illustrates a simple requirement that checks whether the fold expression used in the body of Add is valid.

```
1 requires(Args... args)
2 {
3     (... + args); C SR: args provides +
4 }
```

Listing 1.6

This check ensures that the type passed to Add provides operator+. We just have to check our first requirement **C**.

1.5.2 The nested requirement

Having successfully checked our requirement **C** brings us to the next requirement, the nested requirement. With this requirement kind, we can implement our requirements **A** and **B**, as Listing 1.7 shows. 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.

```
1 requires(Args... args)
2 {
3     (... + args); C SR: args provides +
4     requires are_same_v<Args...>; A NR: All types are the same
5     requires sizeof...(Args) > 1; B NR: Pack contains at least two
         elements
6 }
```

Listing 1.7

A nested requirement evaluates the return value of an expression. In the case of **B**, the nested requirement ensures that there are at least two elements in the parameter pack. The **requires** in front of each nested requirement is crucial. Without this **requires**, we look at a simple requirement. As great as a simple requirement is, it's

the wrong tool at this point. A simple requirement would, in the case of **B**, always return true regardless of the number of elements in the parameter pack. Only the `requires` in front makes it a nested requirement. This is a trap you should not fall into.

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

1.5.3 The compound requirement

The two last requirements for the function `Add` can be checked with a compound requirement. We can check two things with a compound requirement, the return type of an expression and the noexceptness of that expression. Listing 1.8 shows the compound requirement for the requirements **D** and **E** of `Add`.

```

1  requires(Args... args)
2  {
3      (... + args);           C SR: args provides +
4      requires are_same_v<Args...>; A NR: All types are the same
5      requires sizeof...(Args) > 1; B NR: Pack contains at least two
6          elements
7
D E CR: ...+args is noexcept and the return type is the same as the
8      first argument type
9      // { (... + args) } noexcept;
10     // { (... + args) } -> same_as<first_arg_t<Args...>>;
11     { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
12 }
```

Listing 1.8

Probably no surprise, that a compound requirement uses curly braces to form a scope or compound statement. Here, we again see `Add`'s fold expression. I dislike this repetition, but it comes mainly from the fact that the `Add` example is trivial. Next to the fold expression in the compound statement, we see `noexcept`. This tells the compiler to check whether the expression in the curly braces is `noexcept`. The com-

pound requirement yields false should that be not true. With that, we have a very easy way to check the noexceptness of an expression.

Table 1.2: The four different kinds of requires-expression s

Code	Requires-expression kind	Description
<code>(... + args);</code>	Simple requirement	Asserts that the operation $a+b$ is possible.
<code>requires are_same_v<Args...>;</code>	Nested requirement	Asserts that all types of the pack Args are of the same type.
<code>{ (... + args) } noexcept;</code>	Compound requirement	Asserts that the plus operation is noexcept.
<code>{ (... + args) } -> same_as<U>;</code>	Compound requirement	Asserts that the return type of the plus operation is the same as U.
<code>{ (... + args) } noexcept -> same_as<U>;</code>	Compound requirement	Combination of the former two compound requirements. Asserts that the return type of the plus operation is the same as U and the operation is noexcept.

After `noexcept`, we see a token sequence that looks like a trailing return type, and we can read it as such. This trailing return type-like arrow is followed by a concept. At this point, we must use a concept. Type-trait won't work at this place. The concept you can see is `same_as` from the STL. Its purpose is to compare two types and check whether they are the same. If you look at Listing 1.8 on page 27 closely, you can see that I pass only one argument to `same_as`, the resulting type of `first_arg_t`. So, where is the second parameter? The answer is that the compiler injects as the first parameter the resulting type from the compound statement at the beginning of the line. Pretty handy, right?

Basically, the form of the compound requirement, as presented here, does two checks in one. It checks the `noexcept` state of the expression and the resulting type. We can split this into two steps and check for `noexcept`, simply by omitting everything after `noexcept`. Then, we can do the return type check in the second check by striking `noexcept` from the line as presented. I prefer having both checks in a single statement.

Table 1.2 on page 28 captures all four kinds of `requires`-expression s in a compact manner.

Fine, at this point, we have created a `requires` expression that checks all the requirements of `Add` we established in §1.4 on page 23. Next, we look at how to attach this `requires` expression function `Add`.

1.5.4 The type requirement

The type requirement is the last variant of requirement we can have in a `requires`-expression. This type of requirement asserts that a certain type is valid. Listing 1.9 defines a concept `containerTypes` that checks that a given type `T` provides all the types that allocating containers of the STL in C++ usually provide.

```
1 template<typename T>
2 concept containerTypes = requires(T t)
3 { A Testing for various types in T
4     typename T::value_type;
5     typename T::size_type;
6     typename T::allocator_type;
7     typename T::iterator;
8     typename T::const_iterator;
9 }
```

Listing 1.9

Here you can see that a type requirement always starts with `typename`. Should we leave the `typename` out, we are back at a simple requirement.

1.6 Ad hoc constraints

The easiest form is to attach the `requires` expression built in §1.5 on page 25 to the `requires` clause of `Add`, as Listing 1.10 on page 30 shows.

```

1 template<typename... Args>
2   requires requires(Args... args)
3   {
4     (... + args);
5     requires are_same_v<Args...>;
6     requires sizeof...(Args) > 1;
7     { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
8   }
9   auto Add(Args&&... args) noexcept
10  {
11    return (... + args);
12  }

```

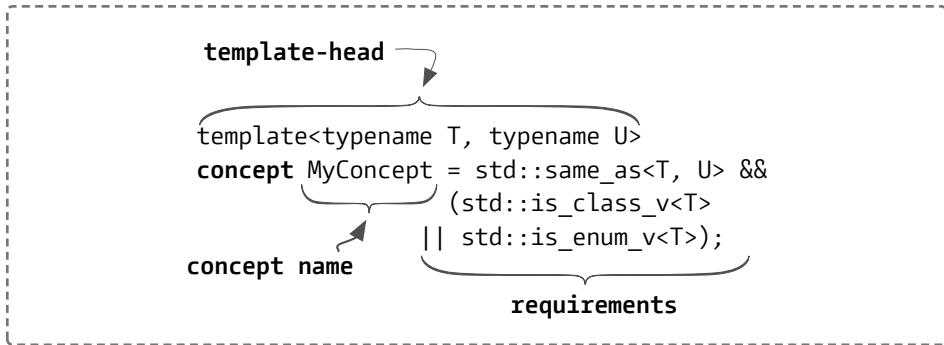
The grey part is the code we developed in §1.5 on page 25, the requires expression. You can see that the first line of the requires expression, starting with **requires**, has a **requires** in front of it. So we have **requires** **requires**. What we are looking at here is a so-called ad hoc constraint. The first **requires** introduced the requires clause, C2, in Figure 1.1 on page 23, while the second starts the requires expression. Instead of C2, we can, of course, also attach the requires expression to the trailing requires clause, which is C5 in Figure 1.1 on page 23.

While an ad hoc constraint is handy, it is also the first sign of a code-smell. We just spent some time developing the requires expression, but all we can do is to use it in one place. Yes, I assume copy and paste is out of the question. The requires expression for Add might be something special that only applies to Add. In that case, using it in an ad hoc constraint is fine, but this isn't true for most of the requirements. Always think twice before using an ad hoc constraint.

How can we do better? Did you notice that I haven't shown you Concepts yet? Only building blocks to concepts and application areas. Now that we have a requires expression, let's start creating a Concept with it.

1.7 Defining a concept

We continue with Add and use the requires expression from §1.5 on page 25 for building our first concept. Figure 1.3 on page 31 illustrates the components of a concept.

**Figure 1.3:** The elements of a concept.

A concept always starts with a template-head. The reason is that concepts are predicates in generic code, which makes them templates. The template-head of a concept comes with the same power and limitations as that of any other function or class template. We can use type and non-type template parameters (NTTPs) parameters, class or typename, or a concept to declare a template type parameter.

After the template-head, Figure 1.3 shows the new keyword `concept`. This starts the concept and tells the compiler that this is the definition of a concept and not, for example, a variable template.

Of course, a concept must have a name. I picked `MyConcept`, yes I know, a great name. After the concept name, we see the equal sign followed by requirements. We assign these requirements to our concept `MyConcept`. As you can see, these requirements can be put together using boolean algebra. Figure 1.3 also shows that we can use either concepts or type-trait as requirements.

With that knowledge, we can look at Listing 1.10 on page 30, which uses the `requires` expression for `Add` and attaches it this time to a concept called `Addable`.

```

1  template<typename... Args>
2  concept Addable = requires(Args... args)
3  {
4      (... + args);
5      requires are_same_v<Args...>;
6      requires sizeof...(Args) > 1;
7      { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
8  };
  
```

```

9
10 template<typename... Args>
11 requires Addable<Args...>
12 auto Add(Args&&... args) noexcept
13 {
14     return (... + args);
15 }
```

Once again, the grey part is the requires expression from §1.5 on page 25. Aside from the concept `Addable`, Listing 1.11 on page 31 shows how `Add` itself changes by using the concept. We use `Addable` now in the requires clause of `Add`. This helps make `Add` more readable, which I find a valuable improvement. The other part is that `Addable` is now reusable. We use it with other functions then `Add` with a similar requirement.

1.8 Testing requirements

So far, we have looked at the different requirement kinds, how to apply them, and how they fit into a requires clause or to a concept. It is time to talk about how to verify our requirements or concepts. As I previously said, we will spend a lot of time developing concepts in the future, so we should also be able to test them like usual code.

The good thing about testing concepts is that we already have all the necessities in place. Remember, concepts only live at compile time. We have a tool to check things at compile time with `static_assert`, so there is no need to check out a testing framework.

We keep using `Add` or, better, the concept we created `Addable`. A type must be valid for `Addable` (or, of course, invalid), to test the various combinations. I prefer creating a stub that mocks the different types. Such a stub is shown in Listing 1.12.

```

1 A Class template stub to create the different needed properties
2 template<bool noexcept, bool operatorPlus, bool validReturnType>
3 struct Stub {
4     B Operator plus with controlled noexcept can be enabled
```

```

5   Stub& operator+(const Stub& rhs) noexcept(nexcept)
6     requires(operatorPlus && validReturnType)
7   { return *this; }
8
9   C Operator plus with invalid return type
10  int operator+(const Stub& rhs) noexcept(nexcept)
11    requires(operatorPlus && not validReturnType)
12  { return {}; }
13 };
14
15 D Create the different stubs from the class template
16 using NoAdd           = Stub<true, false, true>;
17 using ValidClass       = Stub<true, true, true>;
18 using NotNoexcept      = Stub<false, true, true>;
19 using DifferentReturnType = Stub<true, true, false>;

```

Listing 1.12

Here, `Stub` is a class template with three NTTPs **A**. The first one, `nexcept`, controls whether the implementation of `operator+` in `Stub` is `noexcept`. The second parameter, `operatorPlus`, uses a trailing `requires` clause on `operator+` in `Stub` to enable or disable the operator. Last, `validReturnType` decides whether the return type of `operator+` is `Stub`, and by that, valid according to our requirements for `Addable`, or `int`. The choice for `int` is arbitrary. All that's needed is something different than `Stub`.

At the bottom of Listing 1.12 on page 32 at **D**, you can see more meaningful using aliases for the different parameter combinations of `Stub`. For example, I cannot easily recall what `Stub<true, false, true>` does. Still, I understand that `ValidClass` implies that this type should be accepted by `Addable`.

Well, that's it, at least the mocking part. With that we have everything we need to start writing tests, which Listing 1.13 shows.

```

1  A Assert that mixed types are not allowed
2  static_assert(not Addable<int, double>);
3
4  B Assert that Add is used with at least two parameters
5  static_assert(not Addable<int>);
6

```

Listing 1.13

```
7  C Assert that type has operator+
8  static_assert(Addable<int, int>);
9  static_assert(Addable<ValidClass, ValidClass>);
10 static_assert(not Addable<NoAdd, NoAdd>);

11
12 D Assert that operator+ is noexcept
13 static_assert(not Addable<NotNoexcept, NotNoexcept>);

14
15 E Assert that operator+ returns the same type
16 static_assert(
17     not Addable<DifferentReturnType, DifferentReturnType>);
```

Here `static_assert` is used to test all the various combinations of valid and invalid types for `Addable`. Fantastic, isn't it? All in our favorite language, all without macros, all without any external testing framework.

1.9 Abbreviated function template with `auto` as a 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 to look 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`.

1.9.1 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 function's signature indicates that we are looking at a template.

The abbreviated function template syntax can be a little terser. The constraining concept is optional. We can indeed declare a function with only `auto` parameters. This way, C++20 allows us to create a function template very comprehensively.

1.9.2 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 must acquire a lock before performing an operation. An example is a file system operation with multiple processes trying to write data onto the file system. As only one can write at a time because of control structures that 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 that takes a lambda as an argument. In its body, `DoLocked` first acquires a global mutex `global0sMutex`, using a `std::lock_guard` to release the mutex after leaving 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. Deadlocks should no longer happen.

```
1 template<typename T>
2 void DoLocked(T&& f)
3 {
4     std::lock_guard lck{global0sMutex};
5
6     f();
7 }
```

Listing 1.14

I have used this pattern in different variations in many places, but there is one thing that I have always disliked. Can 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 boilerplate code I did not like.

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

```

1 void DoLocked(std::invocable auto&& f)
2 {
3     std::lock_guard lck{global0sMutex};
4
5     f();
6 }
```

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. I often claim that this is much more understandable than the previous version, especially for starters. Thinking of a bigger picture with a more complex example, this syntax is also 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 terser, without `auto`, using just the concept as type. However, some people claim that new features must 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`.

1.10 Using a `constexpr` function in a concept

When it comes to requirements of Concepts, `constexpr` functions come to mind, and the question is, Can we use them, or better yet, their result as a part of a requirement in a Concept? The answer is yes, but only if the function does not use a parameter created in a requires expression or if we play tricks. Sounds complicated, right? It is. Listing 1.16 provides an example.

```

1 A helper function with a default parameter
2 constexpr auto GetSize(const auto& t = {})
3 {
4     return t.size(); B Call the size function of the container
5 }
6
7 C A concept which uses T.size
```

```

8  template<typename T, size_t N>
9  concept SizeCheck = (GetSize<T>() == N);
10
11 D A function that uses SizeCheck
12 void SendOnePing(const SizeCheck<1> auto& cont);

```

Listing 1.16

In **A**, we see a `constexpr` function template or, more precisely, an abbreviated function template. It takes a single parameter and assumes that the type behind it is default constructible. In its body, `GetSize` calls `t.size()` **B**. The default parameter is key here. But let's first see the full picture.

With `SizeCheck` in **C**, we see a concept that uses `GetSize` and compares its return value to `N`. There, we see two interesting things. First, `SizeCheck` is a concept taking a type and a NTTP. The order is also important, as we will see later. In its requirements, `SizeCheck` calls `GetSize`, explicitly instantiating it with `T`, the type parameter of the concept. The result is compared to `N`, the NTTP of `SizeCheck`.

D illustrates a sample usage. The function `SendOnePing` uses `SizeCheck` with 1 as NTTP to ensure that whatever container gets passed has a size of exactly 1. I hope that `SendOnePing`, being an abbreviated function template, is not the most interesting part here, but the way `SizeCheck` is called. The compiler is smart enough to deduce the type of the function's parameter and passes it as the first argument to the concept `SizeCheck`. The second parameter is specified explicitly by us. Cool, right?

Listing 1.17 shows that `SendOnePing` can be called with a `std::array` of `int` with a size of one. No other size is allowed by the concept.

```

1 std::array<int, 1> a{};
2
3 SendOnePing(a);

```

Listing 1.17

The trick we had to play was to make `T` of the concept a part of the `constexpr` function used inside the concept. We cannot use

```
1 requires(T t) { requires t.size() == N; };
```

because `t` is evaluated, and the parameter of a `requires` expression must be unevaluated. Hence, the trampoline jump with a default function parameter.

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

1.11 Concepts and constrained auto types

For some years now, at least some of us have struggled with placeholder variables with a definition of the form `auto x = something;`. One argument I often hear against using `auto` instead of a concrete type is that with this syntax, it 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 does not entirely solve the problem. Think about code review, for example. They often occur in either a browser showing the differences or another diff tool, tending not to provide context. All that said, there is also a good argument for using `auto` variables. They help us get the type right, preventing implicit conversion or loss of precision. Herb Sutter showed years ago [1] that in many cases, we could put the type at the right and, in doing so, leave clues. 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`, 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.

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

Listing 1.18

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 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. These little things matter, especially if your code runs on different targets using different compilers and standard libraries. 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.

```
1 auto v = std::vector<int>{3, 4, 5};  
2  
3 A Using the  
correct type  
4 const std::vector<int>::size_type size = v.size();
```

Listing 1.19

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 regarding what it does, but it is even harder to know the type. Most likely, the deduced type is not a floating-point type, but only with a knowledge of the STL, can you say that.

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

Listing 1.20

This is where Concepts can help us use the best of the two worlds. Think about what we usually need to know in these places. 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.

```
1 auto v = std::vector<int>{3, 4, 5};  
2 const std::integral auto size = v.size(); A Limit the type's  
properties
```

Listing 1.21

Constrained placeholder types allow 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 constraint 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.

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 to write good generic code for years, Concepts enlarge the application areas. We can use Concepts in more places than `enable_if`.

1.12.1 Call method based on requires

One thing that gets pretty easy with Concepts is checking whether an object has a certain function. In combination with `constexpr if`, one can conditionally call this function if it exists. An example is a function template that 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 do 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 to state-of-the-art optimizers. However, for us developers, it means to write and read this nonsense function. We have maintained these functions over the years just to do... nothing. Solutions are using C++11, `decltype` in the trailing-return type, and the comma operator to test a method's existence. The thing is, writing this was a lot of boilerplate code and needed a deeper understanding of all these elements and their combination. With C++20, we can define a concept that has a `requires-expression` containing a simple requirement, with the name `SupportsValidation` and we are done.

```

1  template<typename T>
2  concept SupportsValidation = requires(T t)
3  {
4      t.validate();
5  };

```

Listing 1.22

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, depending on the condition. For example, the condition needs to be a compile-time constant 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.

```
1  template<typename T>
2  void Send(const T& data)
3  {
4      if constexpr(SupportsValidation<T>) { data.validate(); }
5
6      // actual code sending the data
7  }
8
9  class ComplexType {
10 public:
11     void validate() const;
12 };
13
14 class SimpleType {};
```

Listing 1.23

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 (Std-Box 1.4 on page 44), that wrapper should behave as the object wrapped in the `std::optional`. A `wrapper<T>` should behave like `T`. The standard states that `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 constructor if and only if `T` is copy constructible. This task is fairly easy. There are 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 to store the value is using placement new in an aligned buffer. As this should not be a 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 an `enable_if` and 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 must do because of this method's templated version. The resulting code is shorter than the text needed to explain.

```

1  template<typename T>
2  class optional {
3  public:
4      optional() = default;
5
6      template<
7          typename U,
8          typename = std::enable_if_t<std::is_same_v<U, optional> and
9                             std::is_copy_constructible_v<T>
10         >>
11     optional(const U&);
12
13 private:
14     storage_t<T> value;
15 };

```

Listing 1.24

After that, we can try out our shiny, admittedly 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 have 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 the

object of `optional<NotCopyable>` and try to copy construct the second, passing the first as the argument.

```
1  A struct with delete copy operations
2  struct NotCopyable {
3      NotCopyable(const NotCopyable&) = delete;
4      NotCopyable& operator=(const NotCopyable&) = delete;
5  };
6
7  optional<NotCopyable> a{};
8  optional<NotCopyable> b = a;  B This should fail
```

Listing 1.25

That is great! The code compiles! Oh wait, that is not expected, is 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, the intended copy constructor takes a `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 compiler deletes the copy operations of `optional` if a base class has them deleted. Otherwise, they are defaulted.

```
1  struct copyable {};
2
3  struct notCopyable {
4      notCopyable(const notCopyable&) = delete;
5      notCopyable& operator=(const notCopyable&) = delete;
6  };
7
8  template<typename T>
9  class optional
10 : public std::conditional_t<std::is_copy_constructible_v<T>,
11                               copyable,
```

Listing 1.26

```

12     notCopyable> {
13
14     public:
15     optional() = default;
16 };

```

Listing 1.26

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 just write a copy constructor, as we always do. No template is 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 the 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 previous approach. As another plus, this requires zero additional code, which often looks unrelated, can be used by colleagues, and needs maintenance.

```

1 template<typename T>
2 class optional {
3 public:
4     optional() = default;
5
6     optional(
7         const optional&) requires std::is_copy_constructible_v<T>;
8 };

```

Listing 1.27

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

With `std::optional`, we look at a type that 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 to express 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 that may not always return a valid result, for example, the peripheral device is unavailable, `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 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 much 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 that calls the destructor of `T` for the `optional` internal storage.

```
1  template<typename T>
2  class optional {
3  public:
4      optional() = default;
5      // The real constructor is omitted here because it
6      // doesn't matter
7
8      ~optional() requires(not std::is_trivially_destructible_v<T>)
9      {
10         if(has_value) { value.as()->~T(); }
11     }
12
13     ~optional() = default;
14
15     optional(
16         const optional&) requires std::is_copy_constructible_v<T>
17     = default;
```

Listing 1.28

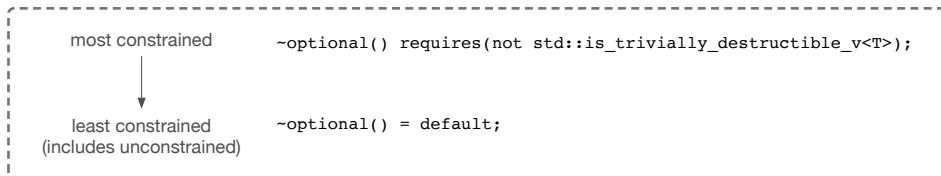


Figure 1.4: The compiler evaluates possible constrained overloaded functions from the most to the least constrained.

```

19  private:
20    storage_t<T> value;
21    bool          has_value{};
22  };

```

Listing 1.28

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.

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-clause s! Apply the least constraint principle.

In the current example of `optional`, the 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 is COM-like objects in Microsoft Windows. A rough sketch of such a class could be this:

```
1 struct COMLike {
2     ~COMLike() {}      A Make it not default destructible
3     void Release();   B Release all data
4
5     // Some data fields
6 };
```

Listing 1.39

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 previous behavior applies. Suppose the type in the `optional` is not trivially destructible. In that case, 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 knowledge, quickly written:

```
1 template<typename T>
2 concept HasRelease = requires(T t)
3 {
4     t.Release();
5 };
```

Listing 1.30

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.

```
1 A Only if not trivially destructible
2 ~optional() requires(not std::is_trivially_destructible_v<T>)
3 {
```

Listing 1.31

```

4     if(has_value) { value.as()->~T(); }
5   }
6
7 B If not trivially destructible and has Release method
8 ~optional() requires(not std::is_trivially_destructible_v<T> and
9                               HasRelease<T>)
10  {
11    if(has_value) {
12      value.as()->Release();
13      value.as()->~T();
14    }
15  }
16
17 ~optional() = default;

```

Sadly, this doesn't compile. The compiler ends up finding two destructors, **A** and **B**. The reason is that both requires-clause s yield true, and both destructors are constrained. Before, we had the case of a constrained and an unconstrained one. One way to handle this is to write mutually exclusive requires-clause s. We add to destructor **A** 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:

$$a \wedge (a \vee b) = a \vee b \quad (1.1)$$

$$a \vee (a \wedge b) = a \quad (1.2)$$

In Eq. (1.1) $a \vee b$ subsumes a , while in Eq. (1.2) a subsumes $a \wedge b$.

Constraint subsumption works only with concepts. This is a strong case for defining named concepts instead of applying requires-clause s. To solve the situation with the destructors, we need to define a second concept `NotDefaultDestructible` and replace the type-trait in the destructors requires-clause s with that. Now we have concepts in the requires-clause, which enables the compiler to do constraint sub-

sumption. If the compiler now approaches the two destructors (**A** and **B** below), it hits the following constraints:

`NotTriviallyDestructible<T>` (1.3)

`NotTriviallyDestructible<T> \wedge HasRelease<T>` (1.4)

Now, the most constrained rule applies because both destructors have only concepts as constraints. The compiler sees that both clauses contain `NotTriviallyDestructible`, but Eq. (1.4) has `HasRelease` in addition. This results in Eq. (1.4) subsuming Eq. (1.3), giving us the final destructor for this type. The one left. The defaulted default destructor is unconstrained, making it the least constrained method.

```

1  A Only if not trivially destructible
2  ~optional() requires NotTriviallyDestructible<T>;
3
4  B If not trivially destructible and has Release method
5  ~optional() requires NotTriviallyDestructible<T> and
6    HasRelease<T>;
7
8  ~optional() = default;

```

Listing 1.32

There is more. This solution is now easily extendable by another destructor. By now, `optional` handles classes with `Release`, but which are trivially wrong. Such a 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 trivially destructible types that 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 that has the `HasRelease<T>` constraint, and we are done.

```

1  A Only if not trivially destructible
2  ~optional() requires NotTriviallyDestructible<T>;
3
4  B If not trivially destructible and has Release method
5  ~optional() requires NotTriviallyDestructible<T> and
6    HasRelease<T>;

```

Listing 1.33

```

7
8 C Trivial and has Release method
9 ~optional() requires HasRelease<T>
10 {
11   if(has_value) { value.as()>Release(); }
12 }
13
14 ~optional() = default;

```

To summarize, prefer named concepts over ad hoc constraints or type-trait. This is especially true for more complex concept-based overloading of methods if you want to benefit from concept subsumption.

1.13.1 Subsumption rule 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 previous add function and limit it to two template arguments. We use this function and create two methods. Both are called `add`, with identical arguments and template-head. Only one is more constrained than the other due to the `AlwaysTrue` concept, while before having the `IsSame` concept as well.

```

1 template<typename V, typename W>
2 concept IsSame = std::is_same_v<V, W>;
3
4 template<typename T>
5 concept AlwaysTrue = true;
6
7 template<typename T, typename U>
8 requires IsSame<T, U>
9 auto add(const T& t, const U& u)

```

```

10  {
11      return t + u;
12  }
13
14  template<typename T, typename U>
15  requires IsSame<T, U> and AlwaysTrue<T>
16  auto add(const T& t, const U& u)
17  {
18      return t + u;
19  }

```

Listing 1.34

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

```

1  int a = 1;
2  int b = 2;
3
4  const auto res = add(a, b);

```

Listing 1.35

The code, as shown, compiles and produces the correct result so far. The second `add` function is selected because this one is the more constrained 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?

```

1  template<typename T, typename U>
2  A The arguments are swapped T, U vs U, T
3  requires IsSame<U, T>
4  auto add(const T& t, const U& u)
5  {
6      return t + u;
7  }
8
9  template<typename T, typename U>
10 B The arguments remain unchanged

```

Listing 1.36

```

11 requires IsSame<T, U> and AlwaysTrue<T>
12 auto add(const T& t, const U& u)
13 {
14     return t + u;
15 }
```

Well, if you try this and your code looks like the one in Listing 1.36 on page 51, 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]
    and AlwaysTrue<T> auto add(const T& t, const U& u)
           ^
1 error generated.
```

Output

Interesting, now the call to `add` is ambiguous. That implies that after this little argument swap, the compiler can no 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, and no types are involved. So far, `IsSame` is seen as different by the compiler. A deep look into the concept definition reveals the type-trait `is_same_v`. This is once again treated with the textual names leading 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` to find the most constrained function. This little textual change lets the compiler think the two con-

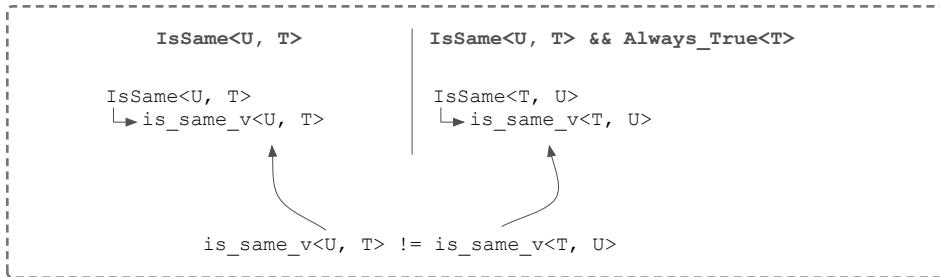


Figure 1.5: Two `is_same_v` differ in their argument order, and with that, are considered different by the compiler.

cepts are different. The compiler looks at ambiguity here. Figure 1.5 shows a more visual explanation.

To make multi-parameter concepts work regardless of the argument order, we need to add a concept that does that swapping. In this case, `IsSameHelper`, which calls `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>`.

```

1 template<typename V, typename W>
2 concept IsSameHelper = std::is_same_v<V, W>;
3
4 template<typename V, typename W>
5 concept IsSame = IsSameHelper<V, W> and IsSameHelper<W, V>;

```

Listing 1.38

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 constrained one and subsumes the add definition without `AlwaysTrue`. No ambiguities any more.

We can conclude here that using an indirection helper concept to make swapped arguments yields the same result for multi-parameter concepts.

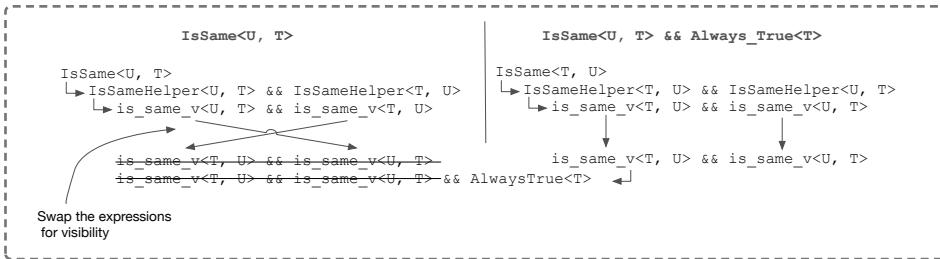


Figure 1.6: IsSame concept with a helper to make swapped arguments work.

1.13.2 One more thing, never say not

In the last part, we learned that the compiler treats concepts during subsumption evaluation differently 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 that 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 the same as before, except that this time, we use `int` and `double` as arguments to add. What I described is shown in the code below.

```

1 template<typename T, typename U>
2 A Inverting IsSame with not
3 requires(not IsSame<T, U>) auto add(const T& t, const U& u)
4 {
5     return t + u;
6 }
7
8 template<typename T, typename U>
9 B Inverting IsSame with not
10 requires((not IsSame<T, U>)and AlwaysTrue<T>) auto add(
11     const T& t,
12     const U& u)
13 {
14     return t + u;
15 }

```

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

```
1 int     a = 1;
2 double b = 2;
3
4 C Call to add is again ambiguous
5 // const auto res = add(a, b);
```

Listing 1.40

With the code as shown in Listing 1.40 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]
    and 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.
```

Output

What did we do now? We used the helper concept before; we did not even swap the arguments, so the helper is unnecessary. 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 constrained, at least from my point of view. However, the compiler disagrees. The compiler disagrees because the `(not IsSame<T, U>)`. The moment we apply `not` or `!` to a concept, the operands become part of the expression. In this context, an expression can be seen as a source location. Two

concepts are only equal if originating 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 compiler's view.

A guiding rule here is to 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 affect using function templates or class templates. With named concepts instead of `typename`, the interface is much more distinguished. Another implicit goal of concepts is improving the error messages you get from your beloved compiler if you supply 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`. Afterward, `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`, that declares a `std::vector<int>` with several values. This function then calls the former function `PrintSorted`. All in all, a few lines of code.

```

1  template<typename T>
2  void PrintSorted(T c)  A By copy to be able to sort it
3  {
4      std::sort(c.begin(), c.end());
5
6      for(const auto& e : c) { std::cout << e << ' ' ; }
7
8      std::cout << '\n';

```

Listing 1.42

```
9     }
10
11 void sortedVector()
12 {
13     std::vector<int> v{30, 4, 22, 5};
14
15     PrintSorted(v);
16 }
```

Listing 1.42

Listing 1.42 on page 56 shows an example implementation. For some of you, this probably seems very easy, and you might think, what does he want now? Bear with me for a moment, please. There are possibly others who have 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 in Listing 1.42 on page 56 trivial, try to think about one of the first times you implemented 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 an 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:

```
1 void sortedList()
2 {
3     std::list<int> l{36, 2, 5};
4
5     PrintSorted(l);
6 }
```

Listing 1.43

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 that fulfills 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` comes 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 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(_RAIIter, _RAIIter) [with
_RAIter = std::_List_iterator<int>]
printSorted0.cpp:4:12:    required from 'void PrintSorted(T)
    [with T = std::__cxx11::list<int>]
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 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> constexp
        decltype ((__y.base() - __x.base()))
        std::operator-(const std::reverse_iterator<_Iterator>&,
```

Output

```
const std::reverse_iterator<_IteratorR>&
```

```
...
```

Output

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 the typename in `PrintSorted`. That alone makes the function clearer to users, but I mentioned that before already.

```
1 template<random_access_iterator T>
2 void PrintSorted(T c)
3 {
4     std::sort(c.begin(), c.end());
5
6     for(const auto& e : c) { std::cout << e << ' ';}
7
8     std::cout << '\n';
9 }
10
11 void sortedVector()
12 {
13     std::vector<int> v{30, 4, 22, 5};
14
15     PrintSorted(v);
16 }
```

Listing 1.45

By doing that and leaving everything else unchanged, the error output is reduced to 26 lines (please note the book formatting is different), stating that there is one error.

```
<source>:22:3: error: no matching function for call to
  'PrintSorted'
  PrintSorted(v);
  ^~~~~~
<source>:9:6: note: candidate template ignored: constraints not
```

Output

```

    satisfied [with T = std::vector<int>
void PrintSorted(T c)
^

<source>:8:15: note: because 'std::vector<int>' does not
    satisfy 'random_access_iterator
template<std::random_access_iterator T>
^

/usr/include/c++/bits/iterator_concepts.h:662:38: note: because
    'std::vector<int>' does not satisfy 'bidirectional_iterator
        concept random_access_iterator =
            bidirectional_iterator<_Iter
^

/usr/include/c++/bits/iterator_concepts.h:652:38: note: because
    'std::vector<int>' does not satisfy 'forward_iterator
        concept bidirectional_iterator = forward_iterator<_Iter>
^

/usr/include/c++/bits/iterator_concepts.h:647:32: note: because
    'std::vector<int>' does not satisfy 'input_iterator
        concept forward_iterator = input_iterator<_Iter>
^

/usr/include/c++/bits/iterator_concepts.h:636:30: note: because
    'std::vector<int>' does not satisfy 'input_or_output_iterator
        concept input_iterator = input_or_output_iterator<_Iter>
^

/usr/include/c++/bits/iterator_concepts.h:615:33: note: because
    '*__i' would be invalid: indirection requires pointer
        operand ('std::vector<int>' invalid
            = requires(_Iter __i) { { *__i } ->
                __detail::__can_reference;
^

1 error generated.
Compiler returned: 1

```

Output

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.

Listing 1.47

```
1  template<typename T>
2  concept HasSortMethod = requires(T t)
3  {
4      t.sort();
5  };
6
7  template<HasSortMethod T>
8  void PrintSorted(T c)
9  {
10     c.sort();
11
12     for(const auto& e : c) { std::cout << e << ' ';}
13
14     std::cout << '\n';
15 }
```

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 with to take away a lot of these minor errors, which blow up page-wise error output. But, if we got the interface wrong at some point, the error messages are way more helpful than ever before. The example Listing 1.47 is probably one I will make again, and again, with concepts, I will be happy about the brief error message. There are other usages where I will no longer have to guess and will get the interface right the first time. Then I will not need the short error messages.

Table 1.3: Existing concepts in C++20

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

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

1.15 Existing Concepts

Now that you've learned how to define your own concepts, I would 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 part of the `<concepts>` header. Most of them are concept definitions for existing type-trait, `same_as` uses `std::is_same_v`. These pre-defined concepts consider subsumption rules, as we discussed them in §1.13.1 on page 50, and have the requires helpers to avoid different results due to parameter swapping. Table 1.3 lists the concepts defined in the standard and available with the STL.

Cliff notes

- `constexpr` function can be called in a nested requirement. Still, the parameters shall not be from the parameter list of a requires-expression.
- A concept always yields a boolean value, never a type.
- The compiler injects missing template arguments if the compiler can deduce them from left to right.
- A `constexpr` function can be called in a nested requirement. Still, 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 get are hard to track to the root cause.
- C++20 allows `auto` as a 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.
- 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: Suspending functions

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?

```
1 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:

```

1 for(int i = 0; i < 5; ++i) { UseCounterValue(i); }
2
3 // other code
4
5 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. It doesn't sound easy knowing that the most common issue in computer science is the off-by-one error.

A more general view of 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 could then 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:

```

1 int counter()
2 {
3     static int i{0};
4     return i++;
5 }
6
7 void UseCounter()
8 {
```

Listing 2.3

```
9   for(int i = 0; i < 5; ++i) { UseCounterValue(counter()); }
10
11 // other code
12
13 for(int i = 0; i < 5; ++i) { UseCounterValue(counter()); }
14 }
```

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 resumable functions are.

Figure 2.1 on page 68 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 can also 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:

```
1 IntGenerator A Returning a coroutine object
2 counter(int start, int end)
3 {
```

Listing 2.4

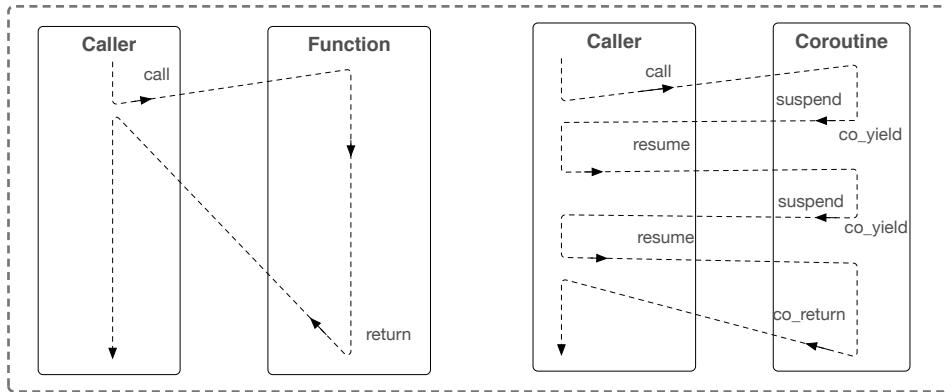


Figure 2.1: The difference in control flow between a regular function and a coroutine.

```

4   while(start < end) {
5     co_yield start; B Yielding a value and giving control back to the
6     caller
7     ++start;
8   }
9
10 void UseCounter()
11 {
12   auto g = counter(1, 5);
13
14   C This sequence runs from 1 to 5
15   for(auto i : g) { UseCounterValue(i); }
16 }
```

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 regular 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: stackful 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 coroutine 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 a stackless 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`

In our previous coroutine example, we saw 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 on page 70.

Table 2.1: Coroutine keywords

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

2.3.3 The generator

When we look at our initial example of a coroutine, we can see therein at A 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 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 Application Programming Interface (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 coroutine state 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`.

```

1  template<typename T>
2  struct generator {
3      using promise_type =
4          promise_type_base<T, generator>; A The PromiseType
5      using PromiseTypeHandle = std::coroutine_handle<promise_type>;
6
7      B Make the generator iterable
8      using iterator = coro_iterator::iterator<promise_type>;
9      iterator begin() { return {mCoroHdl}; }
10     iterator end() { return {}; }
11
12     generator(generator const&) = delete;
13     generator(generator&& rhs)

```

```

14     : mCoroHdl(std::exchange(rhs.mCoroHdl, nullptr))
15     {}
16
17     ~generator()
18     {
19         C We have to maintain the life-time of the coroutine
20         if(mCoroHdl) { mCoroHdl.destroy(); }
21     }
22
23 private:
24     friend promise_type; D As the default ctor is private
25         promise_type needs to be a friend
26
27     explicit generator(promise_type* p)
28     : mCoroHdl{PromiseTypeHandle::from_promise(*p)}
29     {}
30
31     PromiseTypeHandle mCoroHdl; E The coroutine handle
32 };

```

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 [C](#). The memory resource is also the reason why `generator` is move-only.

We left out [D](#) so far, the constructor of the generator, and the friend declaration. Looking 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 [A](#). 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`:

```

1  template<typename T, typename G>
2  struct promise_type_base {
3      T mValue;   A The value yielded or returned from a coroutine
4
5      auto yield_value(T value)    B Invoked by co_yield or co_return
6      { C Store the yielded value for access outside the coroutine
7          mValue = std::move(value);
8
9          return std::suspend_always{};  D Suspend the coroutine here
10     }
11
12     G get_return_object() { return G{this}; };  E Return generator
13
14     std::suspend_always initial_suspend() { return {}; }
15     std::suspend_always final_suspend() noexcept { return {}; }
16     void             return_void() {}
17     void             unhandled_exception()
18                 { std::terminate(); }
19     static auto get_return_object_on_allocation_failure()
20                 { return G{nullptr}; }
21 };

```

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 on page 72. 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 on page 74 using the counter example as a base.

2.3.5 An iterator for generator

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

```
1  namespace coro_iterator {
2      template<typename PT>
3      struct iterator {
4          std::coroutine_handle<PT> mCoroHdl{nullptr};
5
6          void resume()
7          {
8              if(not mCoroHdl.done()) { mCoroHdl.resume(); }
9          }
10 }
```

Listing 2.7

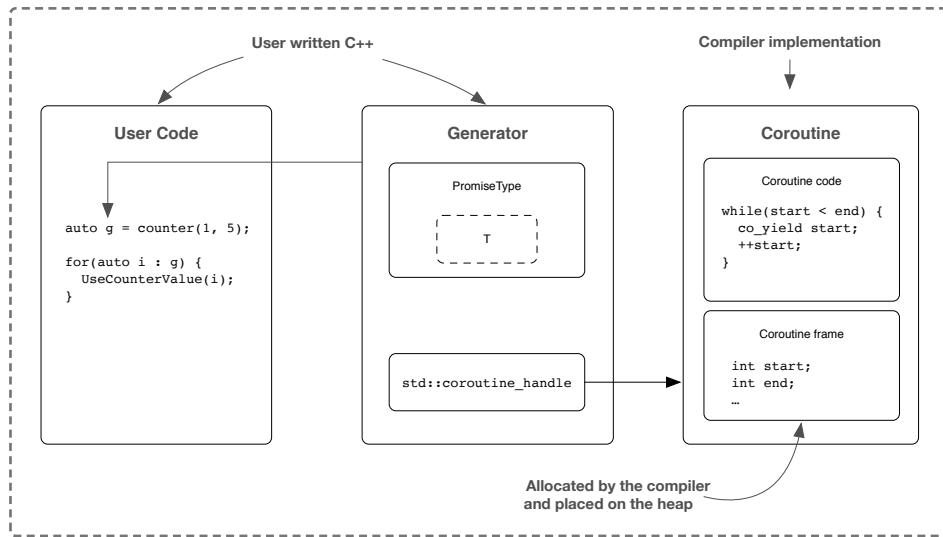


Figure 2.2: How the different coroutine elements interact with each other.

```

11     iterator() = default;
12     iterator(std::coroutine_handle<PT> hco)
13     : mCoroHdl{hco}
14     {
15         resume();
16     }
17
18     void      operator++() { resume(); }
19     bool      operator==(const iterator&) const
20     { return mCoroHdl.done(); }
21
22     const auto& operator*() const
23     { return mCoroHdl.promise().mValue; }
24     };
25 } // 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 lifetime 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 on page 76 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 of a failed allocation.

The second customization point we haven't discussed yet is `unhandled_exception`. We will do so in §2.9 on page 102.

As we have a lot of implementation freedom with all the customization points, Table 2.2 on page 77 summarizes all customization points and their return types.

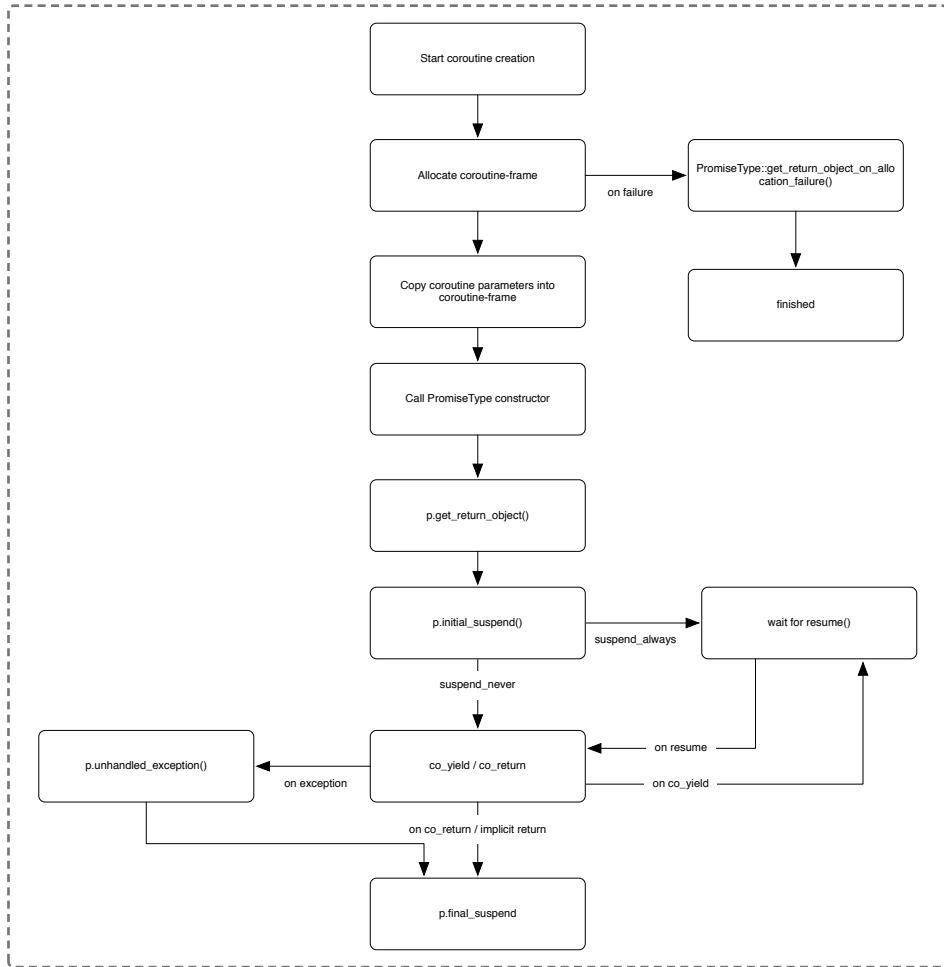


Figure 2.3: The coroutine state machine and the `PromiseType` customization points.

Table 2.2: Coroutine customization points

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

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.

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 also see later.
- 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.
- 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 hand, can be coroutines.

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 as an example, the implementation of a parser.

Suppose we want 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 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 with the same value as ESC is also escaped with ESC. That way, the protocol can contain the full width of a byte and not only printable characters. With

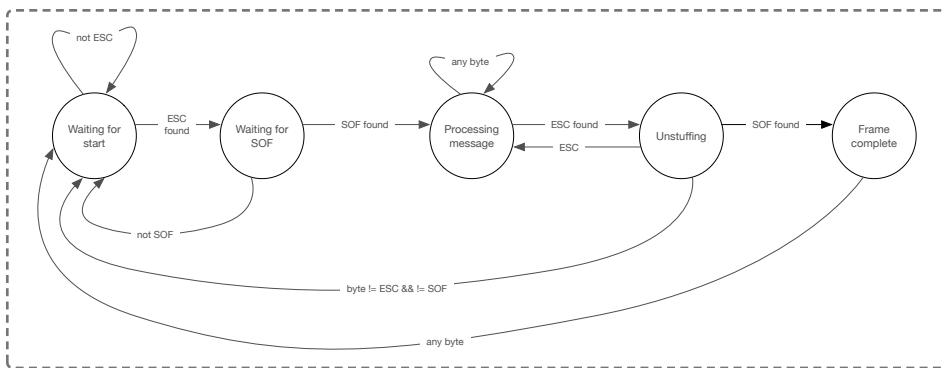


Figure 2.4: State machine diagram of the byte-stream parser protocol.

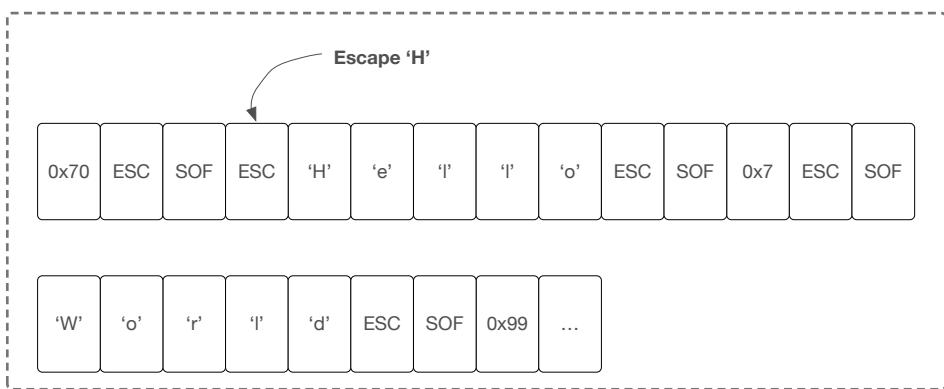


Figure 2.5: Input stream for the parser.

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 state chart and control flow of a parser for this protocol.

The two protocol flags are encoded as follows:

```

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

Listing 2.8

Using 'H' as ESC was a deliberate choice. That way, we must 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.

```

1  template<typename T>
2  void ProcessNextByte(byte b, T&& frameCompleted)
3  {
4      static std::string frame{};
5      static bool      inHeader{false};
6      static bool      wasESC{false};
7      static bool      lookingForSOF{false};
8
9      if(inHeader) {
10         if((ESC == b) && not wasESC) {
11             wasESC = true;
12             return;
13         } else if(wasESC) {
14             wasESC = false;
15
16             if((SOF == b) || (ESC != b)) {
17                 // if b is not SOF discard the frame
18                 if(SOF == b) { frameCompleted(frame); }
19
20                 frame.clear();
21                 inHeader = false;
22                 return;
23             }
24         }
25
26         frame += static_cast<char>(b);
27
28     } else if((ESC == b) && !lookingForSOF) {
29         lookingForSOF = true;
30     } else if((SOF == b) && lookingForSOF) {
31         inHeader      = true;
32         lookingForSOF = false;
33     } else {

```

Listing 2.9

```
34     lookingForSOF = false;  
35 }  
36 }
```

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 simultaneously. 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 into 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 using 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`:

```
1  FSM Parse()  
2  {  
3      while(true) {  
4          byte b = co_await byte{};
```

Listing 2.10

```

5     if(ESC != b) { continue; }

6

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

9

10    std::string frame{};
11    while(true) { B capture the full frame
12        b = co_await byte{};

13

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

17

18            if(SOF == b) {
19                co_yield frame;
20                break;
21            } else if(ESC != b) { D out of sync
22                break;
23            }
24        }

25        frame += static_cast<char>(b);
26    }
27}
28}
29}

```

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 is 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 of 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 is 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 (Listing 2.10 on page 81), we first saw `co_await` to wait in a coroutine for data from outside the coroutine. This data is provided asynchronously. Invoked as shown in the listing, the compiler looks at whether our `PromiseType` also provides the interface for an Awaitable type. A `PromiseType` must provide a method `await_transform` to be an Awaitable type. Alternatively, you can provide another type, which provides operator `co_await`. The method `await_transform` takes a type as a single argument and returns an `Awaiter`. An `Awaiter` works the same as we saw it before with `PromiseType`. The compiler tries to look up a couple of symbols in that type, treating `PromiseType` as an interface. If they are found, the type is an `Awaiter`. If not, we will get a compile error. Table 2.3 on page 84 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 also an `Awaiter`. This type I like to call `async_generator`.

Table 2.3: Coroutine Awaite interface

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

The `async_generator` or, more precisely, the `Awaitable`-part needs to store the awaited data. From the interface, we already saw that an `Awaite` needs to be implemented; we know that there is an `await_ready` method that checks whether the value is already present. This sounds much like a `std::optional` (Std-Box 1.4 on page 44) use-case.

```

1  template<typename T>
2  struct awaitable_promise_type_base {
3      std::optional<T> mRecentSignal;
4
5      struct awaite {
6          std::optional<T>& mRecentSignal;
7
8          bool await_ready() const { return mRecentSignal.has_value(); }
9          void await_suspend(std::coroutine_handle<>) {}
10
11         T await_resume()
12         {

```

Listing 2.11

```

13     assert(mRecentSignal.has_value());
14     auto tmp = *mRecentSignal;
15     mRecentSignal.reset();
16     return tmp;
17 }
18 };
19
20 [[nodiscard]] awaiter await_transform(T)
21 { return awaiter{mRecentSignal}; }
22 };

```

Listing 2.11

Aside from the `Awaiter` interface, the `async_generator` needs to provide `await_transform` as part of the `PromiseType`. Besides 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`.

```

1 template<typename T, typename U>
2 struct [[nodiscard]] async_generator
3 {
4     using promise_type = promise_type_base<T,
5                                     async_generator,
6                                     awaitable.promise_type_base<U>>;
7     using PromiseTypeHandle = std::coroutine_handle<promise_type>;
8
9     T operator()()
10    {
11        A the move also clears the mValue of the promise
12        auto tmp{std::move(mCoroHdl.promise().mValue)};
13
14        B but we have to set it to a defined state
15        mCoroHdl.promise().mValue.clear();
16
17        return tmp;
18    }
19

```

Listing 2.12

```

28     void SendSignal(U signal)
29     {
30         mCoroHdl.promise().mRecentSignal = signal;
31         if(not mCoroHdl.done()) { mCoroHdl.resume(); }
32     }
33
34
35
36     ~async_generator()
37     {
38         if(mCoroHdl) { mCoroHdl.destroy(); }
39     }
40
41
42     private:
43     friend promise_type;   C As the default ctor is private G needs to be
44                           a friend
45     explicit async_generator(promise_type* p)
46     : mCoroHdl(PromiseTypeHandle::from_promise(*p))
47     {}
48
49     PromiseTypeHandle mCoroHdl;
50 };

```

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`

We have a decision to make with the `awaiter_promise_type` that we saw in the previous section. We now need a `PromiseType` in two places, 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 not a 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 also derive from other types.

What we have to do is 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. This parameter pack can also be empty thanks to variadic templates and our code compiles. Here is the upgraded version:

```
1  template<typename T, typename G,
2          typename... Bases> A Allow multiple bases for awaiter
3  struct promise_type_base : public Bases... {
4      T mValue;
5
6      auto yield_value(T value)
7      {
8          mValue = value;
9          return std::suspend_always{};
10     }
11
12     G get_return_object() { return G{this}; };
13
14     std::suspend_always initial_suspend() { return {}; }
15     std::suspend_always final_suspend() noexcept { return {}; }
16     void return_void() {}
17     void unhandled_exception()
18     { std::terminate(); }
19 }
```

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 have 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`.

```
1 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.

```
1 generator<byte> sender(std::vector<byte> fakeBytes)
2 {
3     for(const auto& b : fakeBytes) { co_yield b; }
4 }
```

Listing 2.15

It returns the now well-known generator-type, here of type `std::byte`. As a 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.

There is something very important to notice when looking at these few lines of code. 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, if 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 on page 79 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 that loops over a stream of bytes and feeds them into `Parse`.

```

1 void ProcessStream(generator<byte>& stream, FSM& parse)
2 {
3     for(const auto& b : stream) {
4         A Send the new byte to the waiting Parse coroutine
5         parse.SendSignal(b);
6
7         B Check whether we have a complete frame
8         if(const auto& res = parse(); res.length()) {
9             HandleFrame(res);
10        }
11    }
12 }
```

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.

```

1 std::vector<byte> fakeBytes1{
2     0x70_B, ESC, SOF, ESC,
3     'H'_B, 'e'_B, 'l'_B, 'l'_B, 'o'_B, ESC, SOF,
4     0x71_B, ESC, SOF};
```

Listing 2.17

```

6   A Simulate the first
    network stream
7   auto stream1 = sender(std::move(fakeBytes1));
8
9   B Create the Parse coroutine and store the handle in p
10  auto p = Parse();
11
12 ProcessStream(stream1, p);  C Process the bytes
13
14 D Simulate the reopening of the network stream
15 std::vector<byte> fakeBytes2{
16     'W'_B, 'o'_B, 'r'_B, 'l'_B, 'd'_B, ESC, SOF, 0x99_B};
17
18 E Simulate a second network stream
19 auto stream2 = sender(std::move(fakeBytes2));
20
21 F We still use the former p and feed it with new bytes
22 ProcessStream(stream2, p);

```

With **A**, 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 **B**, 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 I have shown 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 implementing the generator for `Parse`.

One thing that could look odd is the lines `co_await byte` in `Parse`. We know now that these lines work, but telling at 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`:

```
1  FSM Parse(DataStreamReader& stream)  A Pass the stream a parameter
2  {
3      while(true) {
4          byte b = co_await stream;  B Await on the stream
5          if(ESC != b) { continue; }
6
7          b = co_await stream;
8          C not looking at a end or start sequence
9          if(SOF != b) { continue; }
10
11         std::string frame{};
12         D capture the full frame
13         while(true) {
14             b = co_await stream;
15
16             if(ESC == b) {
17                 E skip this byte and look at the next one
18                 b = co_await stream;
19
20                 if(SOF == b) {
21                     co_yield frame;
22                     break;
23                 } else if(ESC != b) {
24                     break; F out of sync
25
26             }
27         }
28     }
29 }
```

Listing 2.18

```

25         }
26     }
27
28     frame += static_cast<char>(b);
29 }
30 }
31 }
```

Listing 2.18

As we can see in **A**, Parse in this implementation takes a parameter `DataStreamReader& stream`. This time by reference, which implies we have to ensure the life-time of `stream` outlives that of `Parse`. In **B**, 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`.

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

Listing 2.19

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

```

1 template<typename T,
2         bool InitialSuspend = true> A New NTTP
3 struct generator {
4     using promise_type = promise_type_base<
5         T,
6         generator,
7         InitialSuspend>; B Forward
8         InitialSuspend
9
10    using PromiseTypeHandle = std::coroutine_handle<promise_type>;
11    using iterator = coro_iterator::iterator<promise_type>;
12
13    iterator begin() { return {mCoroHdl}; }
```

Listing 2.20

```
13     iterator end() { return {}; }
```

```
14
```

```
15     generator(generator const&) = delete;
```

```
16     generator(generator&& rhs)
```

```
17     : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)}
```

```
18     {}
```

```
19
```

```
20     ~generator()
```

```
21     {
```

```
22         if(mCoroHdl) { mCoroHdl.destroy(); }
```

```
23     }
```

```
24
```

```
25     T operator()()
```

```
26     {
```

```
27         T tmp{};
```

```
28         C use swap for a potential move and defined cleared state
```

```
29         std::swap(tmp, mCoroHdl.promise().mValue);
```

```
30
```

```
31         return tmp;
```

```
32     }
```

```
33
```

```
34 private:
```

```
35     D As the default ctor is private we G needs to be a friend
```

```
36     friend promise_type;
```

```
37     explicit generator(promise_type* p)
```

```
38     : mCoroHdl(PromiseTypeHandle::from_promise(*p))
```

```
39     {}
```

```
40
```

```
41 protected:
```

```
42     PromiseTypeHandle mCoroHdl;
```

```
43 };
```

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:

```

1  template<typename T,
2      typename G,
3      bool InitialSuspend> A Control the initial suspend
4  struct promise_type_base {
5      T mValue;
6
7      std::suspend_always yield_value(T value)
8      {
9          mValue = value;
10         return {};
11     }
12     auto initial_suspend()
13     {
14         if constexpr(InitialSuspend) { B Either suspend always or never
15             return std::suspend_always{};
16         } else {
17             return std::suspend_never{};
18         }
19     }
20
21     std::suspend_always final_suspend() noexcept { return {}; }
22     G
23     void get_return_object() { return G{this}; }
24     void unhandled_exception();
25     void return_void() {}
26 };

```

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 we split the generator in this approach. We now have a genera-

tor that only yields values, much like the implementation in our very first example counter. `DataStreamReader` implements the `co_await`-part. Here is this implementation:

```
1  class DataStreamReader { A Awaitable
2  public:
3      DataStreamReader() = default;
4
5      B Using DesDeMovA to disable copy and move operations
6      DataStreamReader&
7      operator=(DataStreamReader&&) noexcept = delete;
8
9      struct Awaiter { C Awaiter implementation
10         Awaiter& operator=(Awaiter&&) noexcept = delete;
11         Awaiter(DataStreamReader& event) noexcept
12             : mEvent{event}
13         {
14             mEvent.mAwaiter = this;
15         }
16
17         bool await_ready() const noexcept
18         {
19             return mEvent mData.has_value();
20         }
21
22         void await_suspend(std::coroutine_handle<> coroHdl) noexcept
23         {
24             mCoroHdl = coroHdl; D Stash the handle of the awaiting
25             coroutine.
26         }
27
28         byte await_resume() noexcept
29         {
30             assert(mEvent mData.has_value());
31             return *std::exchange(mEvent mData, std::nullopt);
32         }
33
34         void resume() { mCoroHdl.resume(); }
```

Listing 2.22

Listing 2.22

```

34
35     private:
36         DataStreamReader&           mEvent;
37         std::coroutine_handle<> mCoroHdl{};

38     };

39
40     E Make DataStreamReader awaitable
41     auto operator co_await() noexcept { return Awaiter{*this}; }

42
43     void set(byte b)
44     {
45         mData.emplace(b);
46         if(mAwaiter) { mAwaiter->resume(); }
47     }

48
49     private:
50         friend struct Awaiter;
51         Awaiter*             mAwaiter{};

52         std::optional<byte> mData{};

53     };

```

Class `DataStreamReader` is the `Awaitable` in this implementation. That means that `DataStreamReader` provides the `Awaiter`-type. We can see in **B** that `DataStreamReader` is neither copy- nor moveable. To save lines, I used Peter Sommerlad's Destructor defined Deleted Move Assignment (DesDeMovA) [4] approach 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, **B** 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 **D**, 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 **E**, 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`.

Table 2.4: `co_await` operator

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

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

```

1  auto stream1 = sender(std::move(fakeBytes1));
2
3  DataStreamReader dr{};  A Create a DataStreamReader Awaitable
4  auto p = Parse(dr);    B Create the Parse coroutine and pass
                         the DataStreamReader
5
6  for(const auto& b : stream1) {
7      dr.set(b);  C Send the new byte to the waiting DataStreamReader
8
9  if(const auto& res = p(); res.length()) { HandleFrame(res); }
10 }
11
12 auto stream2 = sender(std::move(fakeBytes2));  D Simulate a
                                                second network stream
13
14 for(const auto& b : stream2) {
15     dr.set(b);  E We still use the former dr and p and feed it with new
                   bytes
16
17 if(const auto& res = p(); res.length()) { HandleFrame(res); }
18 }
```

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`. In Listing 2.24, you see an implementation of our former `promise_type_base`, which has the two operators.

```

1  template<typename T, typename G, bool InitialSuspend>
2  struct promise_type_base {
3      T mValue;
4
5      std::suspend_always yield_value(T value)
6      {
7          mValue = value;
8          return {};
9      }
10
11     auto initial_suspend()
12     {
13         if constexpr(InitialSuspend) {
14             return std::suspend_always{};
15         } else {

```

Listing 2.24

```
16     return std::suspend_never{};  
17 }  
18 }  
19  
20 std::suspend_always final_suspend() noexcept { return {}; }  
21 G           get_return_object() { return G{this}; };  
22 void        unhandled_exception();  
23 void        return_void() {}  
24  
25 A Custom operator new  
26 void* operator new(size_t size) noexcept  
27 {  
28     return Allocate(size);  
29 }  
30  
31 B Custom operator delete  
32 void operator delete(void* ptr, size_t size)  
33 {  
34     Deallocate(ptr, size);  
35 }  
36  
37 C Allow new to be noexcept  
38 static auto get_return_object_on_allocation_failure()  
39 {  
40     return G{nullptr};  
41 }  
42 };
```

Listing 2.24

Using a coroutine with a custom new is as simple as promised. We provide `operator new` for our `PromiseType` in **A**, 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 stackful coroutines. Okay, maybe we have a little more work to do than we should for real stackful coroutines, but it is doable. Once again we have to update promise_type_base for this task. We can provide an operator new template, which then picks the right allocator. Listing 2.25 shows an implementation of promise_type_base doing that.

```

1  template<typename T, typename G, bool InitialSuspend>
2  struct promise_type_base {
3      T mValue;
4
5      std::suspend_always yield_value(T value)
6      {
7          mValue = value;
8          return {};
9      }
10
11     auto initial_suspend()
12     {
13         if constexpr(InitialSuspend) {
14             return std::suspend_always{};
15         } else {
16             return std::suspend_never{};
17         }
18     }
19
20     std::suspend_always final_suspend() noexcept { return {}; }
21     G                         get_return_object() { return G{this}; };
22     void                      unhandled_exception();
23     void                      return_void() {}
24
25     A Custom operator new
26     template<typename... TheRest>
27     void*
28     operator new(size_t size, arena& a, TheRest&&...) noexcept

```

Listing 2.25

Listing 2.25

```
29  {
30      return a.Allocate(size);
31  }
32
33 B Custom operator delete
34 void operator delete(void* ptr, size_t size)
35  {
36      arena::GetFromPtr(ptr, size)->Deallocate(ptr, size);
37  }
38
39 static auto get_return_object_on_allocation_failure()
40  {
41      return G{nullptr};
42  }
43 };
```

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 the 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. This 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 that has `size` as the 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`.

```

1 arena a1{};
2 arena a2{};

3
4 A Pass the arena to sender
5 auto stream1 = sender(a1, std::move(fakeBytes1));
6

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

9
10 for(const auto& b : stream1) {
11   dr.set(b); D Send the new byte to the waiting DataStreamReader
12
13   if(const auto& res = p(); res.length()) { HandleFrame(res); }
14 }

15
16 auto stream2 =
17   sender(a1, std::move(fakeBytes2)); E Simulate a
     second network stream
18
19 for(const auto& b : stream2) {
20   dr.set(b); F We still use the former dr and p and feed it with new
     bytes
21
22   if(const auto& res = p(); res.length()) { HandleFrame(res); }
23 }
```

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 knows nothing about the arena and can be used as before.

2.9 Exceptions in coroutines

So far, we have looked at the happy path of coroutines, meaning 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 coroutines setup, 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.

```
1  try { ❶ Wrap it in a try-catch block
2      auto stream1 = sender(std::move(fakeBytes1));
3
4      DataStreamReader dr{};
5      auto p = Parse(dr);
6
7      for(const auto& b : stream1) {
8          dr.set(b);
9
10         if(const auto& res = p(); res.length()) {
11             HandleFrame(res);
12         }
13     }
14     ❷ Listen for a runtime error
15 } catch(std::runtime_error& rt) {
16     PrintException(rt);
17 }
```

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.

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 into 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 uncontrollably.
- Using a coroutine with a custom allocator is possible.
- Most likely, C++23 will bring us a coroutine STL with pre-defined generators and `PromiseTypes`.
- Calling `destroy()` on a non-suspended coroutine is UB.

Chapter 3

Ranges: The next-generation STL

In this chapter, we will see the first and largest user of Concepts (see Chapter 1 on page 17): ranges. Ranges are a new element of the STL that makes using algorithms more concise and adds the ability to write code in a functional style.

3.1 Motivation

Probably one of the most powerful parts of C++ is the STL. Although the STL is not used fully in some embedded projects, the library is frequently used, saving us users a lot of time reinventing the wheel. Ranges improve the STL in various ways, as we will see in this section.

3.1.1 Avoid code duplication

The first motivation for using ranges is probably the most obvious one. The `<algorithm>` header gives us nice tools such as `equal` to check whether two containers are equivalent. Here is an example for two strings:

```
1 const std::string firstText{"Hello"};
2 const std::string secondText{"Bello"};
```

Listing 3.1

Listing 3.1

```

3
4 const bool equal = std::equal(firstText.begin(),
5                               firstText.end(),
6                               secondText.begin());

```

This example illustrates two things, the ugliness in terms of repetition and the fact that this code compares the entire container, but we have to spell out begins and ends every time we use an algorithm. The first is there twice, once for `firstText` and once more for `secondText`. The number of times I needed to apply an algorithm only to a portion of a container is vanishing small. Using the entire container is a much more common pattern.

The example Listing 3.1 on page 107 comes with the second downside. The code is hard to read. The intent is not immediately apparent. To be sure of what is happening, we need to parse the entire call to `equal` and check that the `begin` and `end` iterators are not modified. This would mean that we, in fact, do iterate only over a portion of the container as illustrated in Listing 3.2.

Listing 3.2

```

1 const std::string firstText{"Hello"};
2 const std::string secondText{"Bello"};
3
4 const bool equal = std::equal(firstText.begin() + 3,
5                               firstText.end(),
6                               secondText.begin());

```

The benefits ranges give us are expressiveness and a reduction in writing and reading. The code below does the same thing as before. The only difference is that this code uses ranges:

Listing 3.3

```
1 const bool equal = std::ranges::equal(firstText, secondText);
```

As you can see, and as the name implies, ranges always expect to iterate over the entire container. Hence, they take containers as parameters, not the iterators. This drops the need for writing `begin` and `end` repetitively. The bonus is that we can easily see that the operation belongs to the entire range of the two containers. The signal-to-noise ratio is much better with ranges.

We can summarize at this point that ranges, as their name implies, work with the entire range of a container, or better, a collection of items, as we can apply filters and other operations on them. By that, they help us to avoid duplications when we previously needed to explicitly use them with `begin` and `end`.

3.1.2 Consistency

The following motivation is slightly less obvious, corresponding to sometimes confusing and even erroneous situations. Have a look at Listing 3.4:

```

1  struct Container {};
  A Container without begin
2  int* begin(Container);   B Free-function begin for Container
3
4  struct OtherContainer {
  C Container with begin
5    int* begin();
6  };
7
8  void Use(auto& c)
9  {
10    begin(c);           D Call ::begin(Container)
11    std::begin(c);     E Call STL std::begin
12 }
```

Listing 3.4

We have a class modeling a container in **A**. Notice that this container does not have a `begin` member function. Instead, there is a free function `begin` with **B**. But we also have `OtherContainer`, which comes with a `begin` member function in **C**. Now, in `Use`, we don't know which container variant is passed to the function. Hence, we don't know which `begin` actually exists. The question is now, what is the correct way to call `begin` in such a case? The call to `begin`, as shown in **D**, always calls the free function. This is okay for `Container` but not for `OtherContainer`. On the other hand, `std::begin` expects a container to have a `begin` member function, so the right thing for `OtherContainer`, but sadly not for `Container`. The code as presented will not compile for either container type.

This situation is solved using a so-called *using-declaration two-step*, shown in Listing 3.5 on page 110.

```

1 void Use(auto& c)
2 {
3     using std::begin; E Bring std::begin in the namespace
4
5     F Now both functions are in scope
6     begin(c);
7 }
```

There, we see the difference that in **E**, we use `using` to bring `std::begin` into the overload set for `begin`. If we now later, in **F**, place an unqualified call to `begin`, the compiler considers both `begin` functions. In the case as presented, we end up with a call to the free function. The code now behaves correctly. In the case of a `Container` object, `::begin` is called, and for `OtherContainer`, `std::begin` calls `OtherContainer::begin`.

The first issue is you have to know this. The code is not necessarily obvious. The second issue is a bit more subtle. Assume `std::begin` would perform some checks on the type. For example, how about a rule that `begin` should not work with a temporary object? Returning the `begin` iterator to such an object will result in UB once we use that pointer. The STL *could* bring in such a requirement and implement checks accordingly. However, our hand-rolled `::begin` would still not have such a check. That results in different behavior of our code, depending on which `begin` gets called. Such a subtle inconsistency is a very good chance for a wasted week of debugging after realizing that the two `begin` functions have different requirements. That's not good. The optimum would be that all calls are routed via a single `begin` function, which does all the necessary checks before forwarding the call to the proper `begin` version, free or member function.

On top of fewer duplications, ranges also bring us consistency, helping in the described situation. In Listing 3.6 on page 111, we can get rid of the *using-declaration two-step* as well as the call to `begin` with `std::ranges::begin` as a replacement.

```
1 void Use(auto& c)
2 {
3     G Use ranges
4     std::ranges::begin(c);
5 }
```

Listing 3.6

By using `std::ranges::begin`, we get the consistent potential check the STL might perform for a `begin` function, plus this version works with the free and member function and calls the appropriate one.

3.1.3 Safety

Let's move to the next motivation for using ranges: safety. Consider Listing 3.7. Okay, you won't do it in the real world. Let's concentrate on the big picture because sometimes, these happen accidentally. Here, we are passing around a temporary, and accidentally passing this temporary to `std::begin` is possible.

```
1 auto it = std::begin(std::string{"Urg"});
```

Listing 3.7

The issue is that once we start using `it`, we are looking at UB. I described a desire for such a check in the previous section. Errors which occur at run-time, such as this one, are terrible. We need additional tests to catch this error. They should be there anyway, no doubt, but run-time is too late for fast development.

This is what ranges do for us. The code in Listing 3.8 stops compiling because there I used `std::ranges::begin`:

```
1 auto it = std::ranges::begin(std::string{"Urg"});
```

Listing 3.8

This version of `begin` comes with such a check, ensuring we do not accidentally pass a temporary. If we do, we end up with a compile-time error. This is the best error you can get, apart from, of course, no error. Now, you can hopefully see why the former motivation, consistency, is so important. We want the same check, the same level of safety for our hand-rolled `begin` functions, and every other potential check in a future standard.

We are looking here at a so-called customization point. Another example of a customization point is `std::swap`, where we can provide efficient implementations

for our types. Thanks to ranges, we get consistent constraint checking. And yes, this constraint checking is often done by Concepts.

3.1.4 Composability

There is one more motivation for ranges, composability. Consider the code in Listing 3.9. Can you quickly tell what this code does?

```

1 std::vector<int> numbers{2, 3, 4, 5, 6};      A
2
3 std::vector<int> oddNumbers{};                 B
4 std::copy_if(begin(numbers),
5             end(numbers),
6             std::back_inserter(oddNumbers),
7             is_odd);                           C
8
9 std::vector<int> results{};                   D
10 std::transform(begin(oddNumbers),
11               end(oddNumbers),
12               std::back_inserter(results),
13               [] (int n) { return n * 2; });   E
14
15 F
16 for(int n : results) { std::cout << n << ' ';
```

Listing 3.9

We are using the STL in this piece of code. So the code should be great and probably is, but sadly, this code is also hard to read. The start is the vector `numbers` in **A**, which contains a collection of odd and even numbers. Next, in **B**, we declare another vector, `oddNumbers`. This vector is then filled in **C** with all the odd numbers from `numbers`. That requires an algorithm (`copy_if`) from the STL, knowledge about `std::back_inserter`, and, of course, the already addressed verbosity, by passing `begin` and `end` to that algorithm. That is quite some knowledge someone needs to know for just a filter of odd numbers.

The code continues in **D** with another vector, `results`. This vector is then used by the next algorithm, `transform`. We see the `begin / end` repetition once again, as well as the `std::back_inserter` followed by a lambda that multiplies the provided element by two.

The vector, `results`, is then used in a range-based for-loop that prints out all the odd numbers.

Aside from a couple of algorithms, we need to know from the STL, this code creates two intermediate vectors. This makes two names, and we must develop good names. Additionally, these vectors do allocate memory. In real-world code, the odd-filter and the multiplication could be more distributed, making it hard to see the relationship.

While STL algorithms are a great thing, they are not composable, leading to code as in Listing 3.9 on page 112. In general, STL algorithms are hard to compose. Ranges address this, as we see in the following example, Listing 3.10.

```
1 std::vector<int> numbers{2, 3, 4, 5, 6}; A
2
3 B
4 auto results =
5     numbers | std::views::filter(is_odd) |
6     std::views::transform([](int n) { return n * 2; });
7
8 C
9 for(int n : results) { std::cout << n << ' '; }
```

Listing 3.10

We can see that **A** is unchanged. Now, `numbers` is also the only declared vector in this code. After declaring `numbers`, we see in **B** the use of ranges. The syntax might be familiar. It looks like the pipe operation under Unix. First, `numbers` is piped into `filter`, which filters out the odd numbers. The result is then piped into `transform`, which does the multiplication. This code is so much better than the previous version. No intermediate vectors. No names that may distract others. The two operations, `filter` and `transform`, are there in a single statement. Thanks to the other motivation of ranges, fewer code duplications mentioned in §3.1.1 on page 107, the code is immediately very readable, even without deep knowledge about the STL.

To be able to use ranges, we need some terminology, presented below, which will help clarify issues and prevent misunderstandings in what follows.

3.2 The who is who of ranges

With ranges, we get some new terminology, which, at this point, we will look at such that later, we have the same understanding of what I'm referring to.

3.3 A range

A *range* is a type with an iterator pair consisting of a `begin` and `end` function:

```
1 struct Range {
2     T begin();
3     T end();
4 };
```

Listing 3.11

The code in Listing 3.11 is a classical iterator pair. For at least some ranges, we need something slightly different. The above range models a finite range. Here, we usually compare `begin` and `end` for equality and stop the iterations once that condition is true. For an infinite range, which is possible with ranges, the `end` type is usually a different one than that for `begin`. The STL brings us a new type for convenience, `std::default_sentinel_t`:

```
1 struct Range {
2     T begin();
3     std::default_sentinel_t end();
4 };
```

Listing 3.12

You can see `std::default_sentinel_t` as a tag type, which simply enables `end` to have a type and catch the correct overload of `operator==` later on.

3.3.1 What is a common_range?

Ranges, as we established them in §3.3, consist of an `end` member function that provides a sentinel. A `common_range`, on the other hand, is a range where `begin(r)` and `end(r)` return the same type. This implies that a `common_range`'s `end` does not

return a sentinel. All classic iterators are a `common_range`. Most STL types have a `common_range` constructor.

Have a look at Listing 3.13. This example shows a use of C++20's ranges together with `std::accumulate`, a not yet rangified algorithm.

```
1 auto v = std::vector{3, 5, 6, 7, 9};  
2 auto rng =  
3     v | std::views::take_while([](int x) { return x > 5; });  
4 const auto res = std::accumulate(rng.begin(), rng.end(), 0);  
5 std::cout << res << '\n';
```

Listing 3.13

This code, while seeming plausible, does not compile. The reason is that `take_while` returns a C++20 range with a sentinel type for `end`. However, `std::accumulate` expects a `common_range`. In these cases, we need to use `std::views::common` to help us in creating a `common_range` from a C++20 range, as shown in Listing 3.14.

```
1 auto v = std::vector{3, 5, 6, 7, 9};  
2 auto rng =  
3     v |  
4     std::views::take_while([](int x) { return x > 5; })  
5     A View with harmonized iterator types  
6     |  
7     std::views::common ;  
8  
9 auto res = std::accumulate(rng.begin(), rng.end(), 0);  
10 std::cout << res << '\n';
```

Listing 3.14

Here, we can see that all we need to do is to pipe the result of `take_while` into `std::views::common`.

3.3.2 A `sized_range`

A `sized_range` knows its size in constant time, using the `size` member function.

3.4 A range algorithm

A *range algorithm* is an algorithm specialized to operate on a range. In C++20, we have specializations for ranges for many of the algorithms that existed before C++20. However, as of C++20, not all algorithms from, for example, `<algorithm>` are range-ready. Nor are the algorithms in the `<numeric>` rangified in C++20.

We need to specifically use the range version of an algorithm by using the version in the namespace `std::ranges`. We will later see an overview of the different new namespaces.

One reason is that some range algorithms are not drop-in replacements. They come with different complexity or execution parameters.

There is one other important thing about range algorithms. They are executed immediately. This is of significance because views, which we will look at next, aren't.

3.4.1 Projections for range algorithms

A powerful new feature of the rangified algorithms is their projection parameter. You can see them like a transform built into the algorithm.

Suppose we have some data structure like `Book` in Listing 3.15. The data structure has several data members, and a bunch of items of this data structure are stored in a `std::vector`.

```

1 struct Book {
2     std::string title;
3     std::string isbn;
4 };
5
6 std::vector<Book> books{
7     {"Functional Programming in C++", "978-3-20-148410-0"},
8     {"Effective C++", "978-3-16-148410-0"}};

```

Listing 3.15

Our task is to sort this vector using one member as a key. Say `title` in this case. I know this is not complicated, but it is a very dull task. The result tends to be either verbose or hides the intent.

```
1 std::sort(books.begin(),
2           books.end(),
3           [](const auto& a, const auto& b) {
4               return a.title < b.title;
5           });

```

Listing 3.16

Of course, we can move the lambda out of `std::sort` and give that thing a name. That makes the intent clear, but we have now made the lambda accessible, which is not what we want. Plus, we need to come up with a name and not forget, write, and read a bunch of code.

Have a look at the C++20 version where we invoke `std::ranges::sort` using a projection onto the Book title, `&Book::title` without needing to hand-roll a lambda. We can pass this directly to the sorting algorithm, along with an ordering, defaulting to less as follows:

```
1 std::ranges::sort(books, {}, &Book::title);
```

Listing 3.17

Of course, we are free to provide another comparison operation instead of the defaulted `std::less`.

3.5 A view into a range

Thanks to a component called a *view*, ranges are cheap, fast, and can reduce memory allocations. You may already know a view type from C++17. There we have `std::string_view`. The typename already carries the name *view*, yet this datatype has nothing to do with ranges. Despite that, the basic idea is the same, a cheap *view* to some data. While `std::string_view` is specialized for strings, the views from ranges are generic.

3.1 C++17: `std::string_view`

With `std::string_view`, C++17 provides a universal datatype to pass constant strings around. Internally, `std::string_view` stores a pointer to the string and its length. The string must be null-terminated, but the length can point to a position before the null-terminator.



Figure 3.1: The pull model as shown in Listing 3.10 on page 113. First transform pulls a number from filter which itself fetches the number from the variable numbers.

Because `std::string_view` comes with different constructors, it can be constructed from C-style strings, `std::string`, or a pointer and length.

We can access the data with the usual candidates with `data` and the length with `size`. Of course, a `std::string_view` is useable in a range-based for-loop.

A view pulls the data only when needed, making a view a lazy range that is cheap to create. It must have move construction, move assignment, and destruction at a constant time. So all these methods should have $\mathcal{O}(1)$. Optionally, a view can have a copy constructor and copy assignment operator if they satisfy $\mathcal{O}(1)$ as well. The primary goal of views is to be cheap and fast such that all operations have a constant time. Sometimes, people refer to this as the pull model because the data is pulled when needed. Figure 3.1 illustrates the pull with a short pipeline.

The most important thing about views comes with the name. They are only views of some other data. A view doesn't own its elements! We must ensure that the data a view uses lives longer than the view itself. Otherwise, we have UB.

3.6 A range adaptor

The last ingredient ranges need are *range adaptors*. Their purpose is to transform ranges into views and, by that, allow us to create the nice function programming style pipelines we have already seen in Listing 3.10 on page 113. There, we already saw the first range adaptors in the form of `filter` and `transform`.

Although the pipe syntax is the most frequently used, there is another way. We can create range adaptors like objects and pass them to each other. Passing the desired range directly is possible:

```
1 auto filter = std::views::filter(numbers, is_odd); A
2
3 B
4 auto transform =
5     std::views::transform([](int n) { return n * 2; });
6
7 C
8 auto results = transform(filter);
```

In A, we create a `std::views::filter` object `filter` passing the range `numbers` and the filter criteria. Next, we create a `std::views::transform` object in B. As in Listing 3.10 on page 113, the transform multiplies the values by two. We can use these two view objects together, as C shows. You could even use the pipe syntax here.

We can have three different syntaxes for creating a range adaptor. Yep, three. Did I already tell you that we are talking about C++ here? Anyhow, here they are:

- `range | adaptor(args...)`: What we saw in B in Listing 3.10 on page 113.
- `adaptor(range, args...)`: This is what Listing 3.18 A shows.
- `adaptor(args...)(range)`: The variant is used in B of Listing 3.10 on page 113.

Table 3.1 on page 120 provides an overview of existing range adaptors in the STL.

3.6.1 A custom range adaptor

We often aim to reduce duplications. Suppose we have code that uses a particular transformation in several places. For example, say we have a list of prices stored in a `std::vector`. We like to have various filters for them. One is to filter all prices with less than 10 €.

```
1 const std::vector prices{3.95, 6.0, 95.4, 10.95, 12.90, 5.50};
2
3 auto subView =
4     prices |
5         std::views::filter([](auto e) { return e < 10.00; }) |
6         std::views::transform(
7             [suffix](auto i) { return std::to_string(i) + suffix; });
```

Table 3.1: Existing range adaptors.

Adaptor	Description
<code>views::all</code>	Includes all elements of a range.
<code>views::counted</code>	Creates a subrange from an iterator and a count.
<code>ranges::common_view</code>	Converts a view into a <code>common_range</code> (see §3.3.1 on page 114).
<code>ranges::drop_view</code>	Skip the first N elements.
<code>ranges::drop_while_view</code>	Skip all elements as long as the predicate function returns true.
<code>ranges::elements_view</code>	Takes a tuple-like view and a template parameter N to create a view of the N 'th element of each tuple.
<code>ranges::filter_view</code>	Filter using a predicate, only the matching elements are included.
<code>ranges::join_view</code>	Join the elements of multiple ranges into a single view.
<code>ranges::ref_view</code>	Reference the elements of some other range.
<code>ranges::reverse_view</code>	Process a view in reverse order.
<code>ranges::split_view</code>	Split a view based on a delimiter into several subranges.
<code>ranges::take_view</code>	Take only the first N elements and discard the rest.
<code>ranges::take_while_view</code>	Take only the elements from the beginning as long as the predicate function returns true.
<code>ranges::transform_view</code>	Apply a transformation function to all elements in a view.
<code>ranges::keys_view</code>	Get a view on the keys of a pair-like view.
<code>ranges::values_view</code>	Get a view on the values of a pair-like view.

But this is not the only place where we do some filtering on the price values. Yet, all the time, we have to repeat the currency symbol. Maybe we have to deal with different currencies and, therefore, change the symbol, but the logic is the same. In Listing 3.19 on page 119, it is hard to see that the `transform` adds a suffix.

With a custom range adaptor, we can make this clear and let our code speak. We can create such a custom adaptor with the help of a lambda as shown in Listing 3.20 on page 121.

Listing 3.20

```

1 auto subView =
2   prices |
3     std::views::filter([](auto e) { return e < 10.0; })
4   A Use the adaptor as usual
5   | addCurrency(" €");

```

With `addCurrency`, we have a name that carries information. That is very valuable. This function takes a `std::string` with the currency symbol. You probably need a bit more sophisticated formatting because, for some currencies, the symbol goes before the number. Let's ignore this and talk about how we implement `addCurrency`. Listing 3.21 shows an implementation.

Listing 3.21

```

1 B A function returning a callable
2 auto addCurrency(const std::string& suffix)
3 {
4   C Make sure you capture by copy!
5   return std::views::transform(
6     [suffix](auto i) { return std::to_string(i) + suffix; });
7 }

```

We are looking at a function that takes one parameter, the currency symbol. Inside this function `addCurrency`, in **B**, there is the actual range adaptor with the previously seen `std::views::transform`. `addCurrency` does the same job as before, except that the currency symbol is now a capture of the lambda. Be sure to capture by-copy here because this lambda is returned by `addCurrency`. That's it. We just created a custom range adaptor, encapsulating a call to `std::views::transform`, which uses additional parameters to add data to the transformed range element. Of course, we can do the same for the filter if our application requires filtering for less than 10.00 more than once.

3.7 The new ranges namespaces

All the ranges parts are organized in new namespaces within `std`. They are organized as follows:

```

1  namespace std {
2      namespace ranges { /* ... */ // improved algorithms and views
3          namespace views { /* ... */
4              } // namespace views
5          } // namespace ranges
6
7      namespace views = ranges::views; // shortcut for the adaptors
8  } // namespace std

```

The main namespace is `ranges`, in which we find all the new ranges elements as well as the range-compatible algorithms. Within this namespace, there is with `views` another namespace. All the range adaptors are located here. The third namespace, `std::views`, is a shortcut to access the range adaptors. That means that `std::ranges::views` and `std::views` contain the same elements, just spelled slightly differently.

For example, `std::ranges::take_view` becomes `std::views::take`. Postfixing `take` with `view` in a namespace already called `view` is kind of redundant. Hence, the naming pattern is to drop `_view` from an adaptor in `std::ranges`.

Why the new namespace for ranges?

When you see ranges, especially the new namespace, you must now type every time a question like *why a new namespace* often comes up.

The answer is that the behavior and guarantees of some algorithms are different with ranges. Now, C++ is backward-compatible, and we don't want to break existing code. Even if breakage here would mean that the code still compiles but the run-time behavior or the result changes. Why not provide additional overloads, you ask? See §3.10 on page 125 for that, but in short, range algorithms are function objects compared to algorithms being functions. If that's not enough, in some cases, parameters were removed, making overloads difficult even without function objects. Effectively being a source of errors.

3.8 Ranges Concepts

Ranges are the first usage of Concepts (see Chapter 1 on page 17). Table 3.2 on page 123 provides an overview of the various range concepts we have, in addition

to what we saw earlier in Table 1.3 on page 62. Figure 3.2 on page 124 shows the relationship of the iterator categories.

Table 3.2: Range concepts.

Concept	Description
<code>ranges::range</code>	Ensure a type provides a begin iterator and an end sentinel to be a range.
<code>ranges::borrowed_range</code>	A range-type where the iterators are obtained from an expression of it that can be returned safely without danger of dangling.
<code>ranges::sized_range</code>	A range that provides a size function with constant time.
<code>ranges::view</code>	A range that is a view, which has constant time copy/move/assignment.
<code>ranges::input_range</code>	A range that satisfies <code>input_iterator</code> .
<code>ranges::output_range</code>	A range that satisfies <code>output_iterator</code> .
<code>ranges::forward_range</code>	A range that satisfies <code>forward_iterator</code> .
<code>ranges::bidirectional_range</code>	A range that satisfies <code>bidirectional_iterator</code> .
<code>ranges::random_access_range</code>	A range that satisfies <code>random_access_iterator</code> .
<code>ranges::contiguous_range</code>	A range that satisfies <code>contiguous_iterator</code> .
<code>ranges::common_range</code>	A range where begin and end have the same type(see §3.3.1 on page 114).
<code>ranges::viewable_range</code>	Such a range can be safely converted into a view.

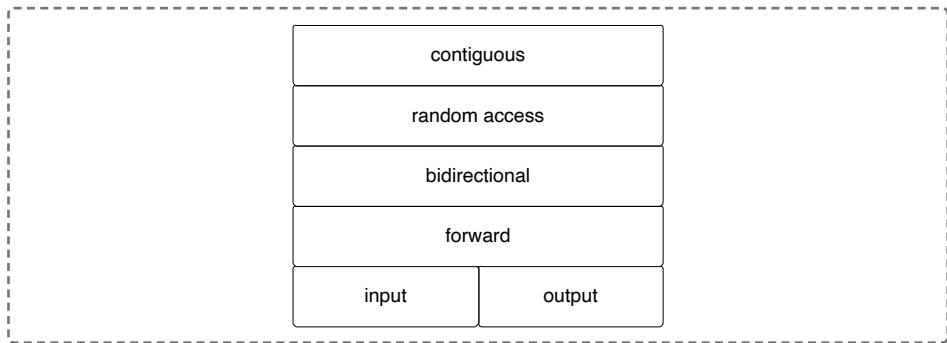


Figure 3.2: The iterator categories and their relationship.

3.9 Views

Views radically change how we can use STL algorithms. They provide on-demand computation. They generate their elements (`iota`) or refer to them from elsewhere.

3.10 Creating a custom range

Let's take a moment and peek behind the curtains of ranges. In this section, we will explore how we can create a custom view, including its corresponding range adaptor. For the sake of simplicity, we will (re)implement `take_view`, which is also present in the STL.

The goal is that the following code compiles and gives the correct result, as it would with `std::views::take`:

```
1 const std::vector<int> n{2, 3, 5, 6, 7, 8, 9};
2 auto v = n | rv::filter(is_even) | views::custom_take(2);
3 std::ranges::copy(v,
4                   std::ostream_iterator<int>(std::cout, " "));
```

Listing 3.23

3.10.1 Implementing the view

The first step is to implement the actual view to a range. In this view, we need to store the range and the number of elements this view should process from the range. While implementing this, we apply the range concepts available in C++20. Listing 3.28 on page 129 shows an implementation, which we will walk through next.

```
1 template<std::ranges::view R> A Using ranges::view concept
2 class custom_take_view
3 : public std::ranges::view_interface<custom_take_view<R>> {
4     B Necessary data members
5     R                                base_{};
6     std::ranges::range_difference_t<R> count_{};
7
8     public:
9         C Default constructible
```

Listing 3.24

```

10    custom_take_view() = default;
11
12    D Constructor for range and count
13    constexpr custom_take_view(
14        R                                     base,
15        std::ranges::range_difference_t<R> count)
16        : base_{std::move(base)}
17        , count_{count}
18    {}
19
20
21    E view_interface members
22    constexpr R base() const& { return base_; }
23    constexpr R base() && { return std::move(base_); }
24
25
26    F Actual begin and end
27    constexpr auto begin() { return std::ranges::begin(base_); }
28    constexpr auto end()
29    {
30        return std::ranges::next(std::ranges::begin(base_), count_);
31    }
32
33    template<std::ranges::range R> G Deduction guide
34    custom_take_view(R&& base, std::ranges::range_difference_t<R>)
35        ->custom_take_view<std::views::all_t<R>>;

```

In the template head of the class template `custom_take_view`, we directly start applying concepts **A**. This class should work only with a view type. This concept checks that `R` is a range that is movable, default initializable, and is a view. The last concept checks whether `R` is derived from `view_base`. Our `custom_take_view` derives from `view_base` with the help of `view_interface`, using Curiously Recurring Template Pattern (CRTP).

After the head, we declare the required data members **B**. In **C**, we ensure that `custom_take_view` is default-constructible. The next constructor in **D** is used when our view is created from a range and a count. This is the second item in the list in §3.6 on page 118.

The `view_interface` requires a couple of members, which are implemented in [E](#). The actual implementation for `begin` and `end` is presented in [F](#). Here, we use the ranges `begin` version. In `end` you can see the *actual* implementation, the only part here that does something other than setting defaults. Using `std::ranges::next`, we retrieve the next iterator element after `begin` with an offset of `count`. With that, we already have a working implementation of `custom_take_view`.

Please note that this is a simplified version. The STL has to deal with different range types, depending on whether `R` is a `simple_view` or a `sized_range`. We leave this out here.

What is also needed is the Class Template Argument Deduction (CTAD) deduction guide in [G](#). Without this deduction guide, we cannot create a `custom_take_view` just by passing a range and a count.

We now have the base for our `custom_take_view`. Let's make our view fit nicely into all the other ranges and add the missing range adaptor.

3.10.2 A range adaptor for `custom_take_view`

The next thing we need for `custom_take_view` is a range adaptor. Let's add this. The code you see below should, in practice, be wrapped in a dedicated namespace like `details`. This is left out here.

```
1 template<std::integral T> A Only integrals
2 struct custom_take_range_adaptor_closure {
3     T count_; B Store the count
4     constexpr custom_take_range_adaptor_closure(T count)
5     : count_{count}
6     {}
7
8     C Allow it to be called with a range
9     template<std::ranges::viewable_range R>
10    constexpr auto operator()(R&& r) const
11    {
12        return custom_take_view(std::forward<R>(r), count_);
13    }
14};
```

Listing 3.25

We start by creating a range-adaptor closure type **A**. The current implementation is a bit verbose. The closure type could be a lambda, but I think seeing the individual pieces is easier for now.

As for `custom_take_view`, `custom_take_range_adaptor_closure` stores the count and returns a `custom_take_view` in the call operator where the range and the internally stored count are passed to the constructor **C**. With that, we already have a bit of glue code that we need for our actual range adaptors, as you see it in Listing 3.26.

```

1  struct custom_take_range_adaptor {
2      template<typename... Args>
3      constexpr auto operator()(Args&&... args)
4      {
5          if constexpr(sizeof... (Args) == 1) {
6              return custom_take_range_adaptor_closure{args...};
7          } else {
8              return custom_take_view{std::forward<Args>(args)...};
9          }
10     }
11 };

```

Listing 3.26

The range adaptor doesn't store any data. The purpose of the adaptor is to create a `custom_take_view` when invoked. This type comes only with a call operator, which is a variadic template. This time, I used a simplification. With the variadic template, `custom_take_range_adaptor` needs only one call operator. Internally, `custom_take_range_adaptor` uses a `constexpr if` to switch between the one- and two-argument case. If only one argument is provided, this is the count. Hence, a closure-type object is returned. A call with two parameters passes a range and the count, allowing us to directly create a `custom_take_view`.

So far, so good. This is the required range adaptor.

3.10.3 Add the pipe-syntax to `custom_take_view`

I see you also like to have the nice pipe syntax for `custom_take_view`. All right, how do we build that? Well, with `operator|`, of course.

Listing 3.27

```

1 template<std::ranges::viewable_range R, typename T>
2 constexpr auto
3 operator|(R&& r, const custom_take_range_adaptor_closure<T>& a)
4 {
5     return a(std::forward<R>(r));
6 }

```

Once again, the function is guarded with concepts, requiring a `viewable_range`. The operator takes a range as the first argument and a `custom_take_range_adaptor_closure` type as the second. Internally, the closure type gets invoked with the range as the parameter. This effectively returns a `custom_take_view`.

With that, our `custom_take_view` supports the pipe syntax. That was easy, right?

What is left to do, for consistency, is that we create our own namespace `views` and put a `custom_take_range_adaptor` object in this namespace, with the abbreviated name `custom_take`.

Listing 3.28

```

1 namespace views {
2     inline details::custom_take_range_adaptor custom_take;
3 }

```

Perfect! Now our `custom_take_view` behaves exactly like `std::views::take`. Well, except that the implementation is simplified, leaving out some special cases.

3.10.4 A more generalized pipe-syntax implementation

Should you implement custom views more often, you probably look for ways to reduce the amount of code you write for each view.

One thing you can do is to generalize the implementation of `operator|`. We already used range concepts extensively. This helps us to provide a generalized `operator|` implementation, as you can see in Listing 3.29.

Listing 3.29

```

1 template<std::ranges::viewable_range R,
2           std::invocable<R> Adaptor>
3 constexpr auto operator|(R&& r, const Adaptor& a)
4 {

```

```
5     return a(std::forward<R>(r));  
6 }
```

We know that the first parameter must be a range. We even had the compiler deduce the second template type, which we then used to instantiate `custom_take_range_adaptor_closure`. Instead of using this explicit type here, we can use the `std::invocable` concept to ensure that the second template parameter is an invocable. That way, we do not need to name the type explicitly while still preserving the requirements.

This is not much, but it saves you implementing `operator|` for each type.

Cliff notes

- Less duplication: `std::ranges::sort(v)`; instead of `std::sort(v.begin(), v.end())`;
- When using `std::ranges::begin` instead of `std::begin`, we always get the constraints check, even if this call boils down to a user-provided `begin`.
- Ranges prevent us from passing temporaries when they will end up as a dangling reference.
- Be aware that passing temporaries to an algorithm becomes possible with ranges. What may look convenient isn't.
- Be sure to capture by-copy if you return a lambda from a custom range adaptor.

Chapter 4

Modules: The superior way of includes

Modules are one of the long-awaited features in C++. We could work with precompiled headers on some platforms, but they were never standardized. C++20 finally changed this by standardizing what modules are. In this chapter, I would like to give you a good idea about how modules work and will also show you how to introduce modules in a legacy system. Before I start, I want to start by explaining the world without modules, which leads naturally to why you want to use modules.

4.1 Background about the need for modules

In my experience, compilation speedup is one point that very often comes up when modules are being discussed. While this *can* be the case, other points make modules desirable.

4.1.1 The include hell

Let's start by considering the following header file `<StrCat.h>`, completely modules-free:

Listing 4.1

```

1  #ifndef STRCAT_H
2  #define STRCAT_H
3
4  #include <string>
5  #include <type_traits>
6
7  namespace details {
8      inline std::string ConvertToString(bool b)
9      {
10         return b ? std::string{"true"} : std::string{"false"};
11     }
12
13 } // namespace details
14
15 template<class T>
16 inline decltype(auto) Normalize(const T& arg)
17 {
18     // Handle bools first, we like their string representation.
19     if constexpr(std::is_same_v<std::remove_cvref_t<T>, bool>) {
20         return details::ConvertToString(arg);
21
22     } else if constexpr(std::is_integral_v<T>) {
23         return std::to_string(arg);
24
25     } else {
26         return (arg);
27     }
28 }
29
30 #endif /* STRCAT_H */

```

Further, assume there is another header `<String>` that includes `<StrCat.h>`.

There are at least two issues that arise from the example. We often have include dependencies, such as `<string>` or `<type_traits>`. Then, with `<String>`, all the headers of `<StrCat.h>` are included once again. Looking at the STL, it often needs to include headers such as `<stdint>` to get `std::size_t` and other basic types. For example, `std::vector` needs `std::size_t`, as well as `std::string`.

The issue that arises from these repeated includes is that they hurt compilation times. Each time we include a header, this header has to be looked up in the include path list of the compiler. Then, the file has to be read from the file system. Next, parsing starts. Although we use include-guards to prevent duplicate symbols and other issues, the compiler must fully parse the header, even with include-guards. Access time might decrease compilation time even more if a header is stored on a network location. Doing this repeatedly then hurts compilation time even more. So, the first thing to avoid in terms of better compilation times is, of course, to speed up this part.

4.1.2 I like to have secrets

There is more. Have a look at `Normalize`. There, we look at a typical case that often comes up. The intention is to provide a function that does something internally. In this case, `Normalize` does two internal things. First, `Normalize` uses type-trait to query some properties of the type, and second, `ConvertToString`, which is hosted in the namespace `details`. Because `Normalize` is a template, we cannot move this private part into an implementation file. We could do that with a non-template function, which we often do for precisely this reason: to hide implementation details.

Sadly, with the code given, we have no other choice than to pollute the header `<String>` and all others, including `<StrCat.h>`, with the `<type_traits>` header. With `<string>` we are looking at a slightly different case, as the purpose here is to convert everything into a string. But we can ask the question of whether `<StrCat.h>` should take care of providing `<string>` to include files or if `<String.h>` is the better one.

This leaves us with the potential performance penalty we discussed before and a design issue. Hiding details and all its contents is impossible. This leads to more code the compiler needs to parse, which can hurt compile times.

Let's see what modules can do for us. We start looking at the different types of modules and the syntax to create them.

4.2 Creating modules

In general, I would say that creating a module is easy. All you have to learn are two new keywords, where one is a context-dependent keyword:

- `module` declares a module or states that a file is part of a module. This is context-sensitive.
- `import` a module or header unit.

The third keyword you already know is `export`. As the name implies, `export` allows us to control which symbols to export from a module.

When creating modules, we can first cheat a little to get a quick, cheap solution by creating a so-called header unit. A header unit is a legacy header file that is imported via `import`. The difference with `include` is that an `import` statement is not preceded by a hash character at the beginning and is followed by a semicolon. The compiler can automatically translate the header file into a module and compile the contents as a module. This gives you the instant speed of modules, along with other pluses, such as hiding `defines`.

The other, much better option is to create a named module. Such a named module is what we refer to as a module. A named module uses the new syntax and all the module features. Consider the header unit as a transition tool.

4.2.1 A header unit

Header files can be turned into a header unit, which will not have all the benefits of modules. With a header unit, for example, we cannot control which symbols are exported. A straightforward view of header units is that they are precompiled headers.

Going back to the example in Listing 4.1 on page 132, to make `<StrCat.h>` a header unit, we use `import` instead of `#include`:

```
1 import "StrCat.h";
```

Listing 4.2

The notation for `" "` or `<>` is the same as for include files. The first syntax looks in the local folders, while the second expects a file to be in the compiler's system include path.

The interesting part when starting with header units is the build system. As I said, they are basically precompiled headers. This implies that we need a step to precompile such a header, and second, the result needs to be stored in a known location. I skip details at this point because they depend heavily on your compiler and toolchain. Also, at the time of writing, modules aren't fully supported at the time of writing in some of the major compilers.

4.2.2 A named module

A much better way to deal with modules that also give you the full power of modules is to create a named module.

Let's look at the skeleton of a module, or more precisely, a module interface.

```
1  module;  A Starts the module, only required, if we have legacy includes
2  // Global module fragment, legacy #includes go here
3
4  [export] module AwesomeModule;  B Export if it is an interface
5  // Module Purview, imports go here
6
7  import x;          C Import something only visible in the module
8  export import y;  D Import something and export it
9
10 export
11 {
12     E Export everything in here
13 }
14
15 export void Fun();  F Export individual symbol
16 void      Bun();  G This symbol stays in the module
```

Listing 4.3

The named module starts in **A** with the global module fragment. This part is optional. We require the global module fragment when we have to include legacy headers.

Next, **B** starts the module interface of a named module `AwesomeModule`. Without `export`, we are dealing with a module implementation. This is an optional step. As before, with header files, we can decide to split the interface and the implementation. However, as modules allow us to keep things module internal, splitting interface and implementation may become more a question of structuring our code. After the start of a named module, the so-called Purview starts. Everything we do in here is private by default. All symbols that should be visible outside of our module require `export` in front.

Is there a new suffix for modules?

Once you start writing modules, you may ask yourself what is the proper file extension for a named module?

There is a desire to have a dedicated file extension for module interface files. One reason is to quickly know whether a project comes with modules or to write a quick search command to find symbols only in module interface files.

The two major compilers, Clang and MSVC, support different extensions for module interface files

- Clang: `.cppm` or `cxxm`
- MSVC: `ixx`

GCC seems to detect whether a file is a module interface without the hint of an extension.

However, no compiler offers a new suffix for the optional module implementation files. The reason is that these files are more or less like regular C++ files, so you can go with your usual file extension for these.

We can see in [E](#) that we can create an entire exporting scope. The alternative is to put `export` before the symbols, as [F](#) illustrates. Without that `export` and outside of an `export` scope, the symbol is visible only inside of the module, which is the case for [G](#).

There is one very crucial difference between a named module and a header unit, the latter preserves macros while a module doesn't.

Modules and dots in the name

Unlike in other languages, you might know where the dot in a name implies some kind of structure of a package, dots in a C++ module name are purely aesthetic. They have no implications on the result for the compiler, only readability for us humans.

Great! With the new knowledge, we can polish Listing 4.1 on page 132 into a shiny named module with all the benefits from modules.

4.3 Applying modules to an existing code base

After we have covered the basics of modules, the time has come to see how they work in code and how you can apply them to an existing codebase.

4.3.1 Down with namespace details

As a first example, I want to consider the `Normalize` function we previously saw in Listing 4.1 on page 132. The code initially came from C++ Insights. Here is the code again for convenience.

```
1  namespace details {
2      inline std::string ConvertToString(bool b)
3      {
4          return b ? std::string{"true"} : std::string{"false"};
5      }
6
7  } // namespace details
8
9  template<class T>
10 inline decltype(auto) Normalize(const T& arg)
11 {
12     // Handle bool's first, we like their string representation.
13     if constexpr(std::is_same_v<std::remove_cvref_t<T>, bool>) {
14         return details::ConvertToString(arg);
15     } else if constexpr(std::is_integral_v<T>) {
16         return std::to_string(arg);
17     } else {
18         return (arg);
19     }
20 }
```

Listing 4-4

There are more conversion functions available, and in `Normalize`, you can see that the function does different things when the type `T` is `bool` or something else. Let's focus on the `bool` case for today.

To convert a `bool` into a `std::string`, I have a function called `ConvertToString`. If you look at the function definition of `ConvertToString`, or better, where this definition is located, you can spot that I put `ConvertToString` into a namespace called `details`. Other people have different names. Sometimes I call that namespace `helper`. I once saw `hands_off`. They all aim to carry the same

intent, saying, *this is part of internal implementation, please don't use the symbols in here somewhere.* Yes, sometimes I have something to hide.

However, this approach is fragile. First of all, nobody knows what `details` means. Within the entire project? Only within this header file? Do you want to tell me this is a detail of the implementation that only I know, but I can use the symbols in this namespace everywhere I want?

There are more interpretations for us humans. For the compiler, this namespace simply means that we must write `details::` to reach `ConvertToString`. That's all. The code will be processed each time this header file is included and compiled if required. And, of course, everyone who can type `details::` before the function name is allowed to use `ConvertToString`, as far as the compiler is concerned.

Looking at the code again, we can say that we failed badly without any better options. We cannot get the compiler to check and obey the meaning of whatever name we have chosen for that namespace.

But that is the past. Let's see how modules improve everything.

4.3.2 Now I can hide my secrets from you...

With C++20's modules, we can shape our API design in a much better, robust way. Have a look at the C++20 version below.

```

1  export module strcat;  A Declare and export a module strcat
2
3  import<type_traits>;
4  import<string>;
5
6  B The namespace is gone
7  std::string ConvertToString(bool b)
8  {
9      return b ? std::string{"true"} : std::string{"false"};
10 }
11
12 C Note that I say export here
13 export template<class T>
14 inline decltype(auto) Normalize(const T& arg)
15 {
```

Listing 4.5

```
16 // Handle bools first, we like their string representation.  
17 if constexpr(std::is_same_v<std::remove_cvref_t<T>, bool>) {  
18     return ConvertToBoolString(arg); C Again, namespace is gone  
19  
20 } else if constexpr(std::is_integral_v<T>) {  
21     return std::to_string(arg);  
22  
23 } else {  
24     return (arg);  
25 }  
26 }
```

Listing 4.5

First, I start declaring that this file is a named module `strcat` in **A**. Next, as in a header file, I import this module's required headers, `<type_traits>` and `<string>`. Here, I use header unit for best performance.

Then, in **B**, we see `ConvertToBoolString` again. The implementation is unchanged, except that the function is no longer in a dedicated namespace. Why? Simply because the namespace is no longer necessary. As I stated before, the name of this namespace wasn't helpful. The intention was to mark the elements in this namespace private. With modules and without saying `export` for a symbol, we get this meaning, which the compiler automatically understands and obeys for free. Not having to come up with a name for a namespace here leaves us this energy for the really important names. In terms of clean code, I think this approach is also better. We managed to reduce this code to its essence.

Moving on to **C**, we see `Normalize`, which like `ConvertToBoolString`, is unchanged, except that the function declaration starts with `export`. By that, we tell the compiler and our fellow developers that `Normalize` is a function that this module exports for use in importing modules or files. Everything that is not marked `export` is unusable outside our module.

4.3.3 What you gain

With the new ability to mark functions as private (or not exported), we have a better way to guarantee a stable Application Binary Interface (ABI) for our customers if we are a library vendor. Why? Because we can now explicitly name the symbols that should be exported and can hide the others. That way, we no longer expose internals that we may want to change later. For example, suppose one day I

can make `ConvertToString constexpr`. That's not something my library customers should see. On the other hand, I might decide to remove `inline`. In the old world, that would risk an ABI break because the compiler has the right to inline this function but isn't required to. Adding or removing `inline` can lead to an ABI break. That is still true for exported functions, but no longer for private ones.

The ability to express the difference between private and public symbols is what I think is by far the most valuable part of modules. For the first time, we have a mechanism to state which symbols we want to be exported - leaving us with the choice to have module-private symbols to shape our code internally without any caveats, as others are no longer allowed to call internal parts. In some sense, modules give us a control similar to what we have always had with classes for ages.

What we just saw applies to other patterns unrelated to modules. For example, the Pointer to implementation (PIMPL) idiom is often used to speed up compile times. With modules, we can write *normal* code without the indirection that PIMPL requires.

4.3.4 Templates in modules

It was always there, but let me point out that with `Normalize`, we looked at a function template the entire time. This answers the question of whether we can have templates in modules. Yes, we can. The definition is preserved because a module is compiled into something like an Abstract Syntax Tree (AST), allowing us to instantiate a template from the compiled module.

4.3.5 Down with DEBUG

Remember that I told you that only header unit preserve macros, but named modules don't? That is a good thing! Yes, there are still places where they are needed. But let's inspect a case where we can use modules to move a define further away.

Consider the following case. We have some code that should only be executed while the binary is compiled in debug mode.

```
1 #ifdef DEBUG
2 std::cout << "Debugging is enabled\n";
3 #endif
```

Listing 4.6

Admittedly, this is a no-brainer. But lots of defines around the codebase make an analysis of our code harder as well as working with that code. Suppose someone accidentally deletes the semicolon. How fast you would detect this error depends on whether you always compile the debug mode or only when needed. In the latter case, weeks can go by until you notice that there is an error, and then tracking down the source of the error becomes much more complicated. While removing the semicolon is rare, a change in a function's API is the more common source of running into trouble here.

A much better way, which has nothing to do with modules yet, is switching to a `constexpr if` (see Std-Box 1.3 on page 41):

```
1 if constexpr(DEBUG) { std::cout << "Debugging is enabled\n"; }
```

Listing 4.7

That way, the faulty code is compiled, even in release mode, but discarded there. The code snippet can be improved a bit further, moving the define further away:

```
1 if constexpr(IsDebugEnabled()) {
2     std::cout << "Debugging is enabled\n";
3 }
```

Listing 4.8

We hide the define behind a `constexpr`, or, if you wish to peek to Chapter 12 on page 301, even a `constexpr` function:

```
1 constexpr bool IsDebugEnabled()
2 {
3     return DEBUG;
4 }
```

Listing 4.9

We ensure that `IsDebugEnabled` returns a `bool`, making our code type-safe at this point. While this is already an excellent approach, it has nothing to do with modules, correct? Well, the thing that has bothered me for a long time is that `DEBUG` is still accessible in the binary. Everybody can still use the old scheme and write code like in the first or second examples. What I want to do is to hide `DEBUG` from everybody and make the symbol accessible only via `IsDebugEnabled`. At this point, we are talking about modules.

Let's move that entire `IsDebugEnabled` function into a newly created module called `config`. We anticipate that, over time, we will have more items like this. For that, we start our module with `export module`, followed by our name `config`. All that's left to do is prefix `IsDebugEnabled` with `export`, and voilà, there you have a perfectly hidden `DEBUG` define.

```
1 export module config;
2
3   export consteval bool IsDebugEnabled()
4   {
5     return DEBUG;
6 }
```

Listing 4.10

We now need to only pass the define `DEBUG` only to the `config` module compilation. The rest of the world doesn't need the macro. Everybody must now use `IsDebugEnabled`.

4.3.6 In-line definitions of class member functions

Let me introduce you to another slight change that modules bring us. Have a look at Listing 4.11.

```
1 struct CppIsGreat {
2   CppIsGreat();
3 };
```

Listing 4.11

There, we are looking at a class called `CppClassIsGreat`, which has a out-of-line defined constructor. The reason usually for doing this is, that inline member functions are implicitly declared `inline`. However, in case a member function has a lot of lines of code, we don't want this code to be inlined. The only way to achieve this is to do an out-of-line definition. Sometimes, you can see the implementation directly beneath the class definition.

Because modules work differently, they are parsed and compiled only once, regardless of how often they are included, this system isn't needed there. As a matter of fact, the system is different for modules.

A class's member functions defined in a module are *not* implicitly `inline`. We have the freedom to implement even a larger member function `inline` and have to specify explicitly should we want that function to be `inline`.

4.3.7 There are some limits

While I'm pleased about the fact that modules allow me to hide more macros, there is a downside to that fact.

Sometimes, and I really regret having to write these lines, macros can be helpful. One popular example is `assert` from `<cassert>`, which is defined by the standard like this:

```
1 #ifdef NDEBUG
2 # define assert(condition) ((void)0)
3 #else
4 # define assert(condition) /*implementation defined*/
5 #endif
```

Listing 4.12

Now, should we import `<cassert>` as a header unit, the macro is preserved. No trouble there. However, in a module-only world, by using a named module, the `define assert` is lost. It looks like we will have some headers in the old-fashioned way, even in the future.

Cliff notes

- You can import a named module or header unit with `import`.
- In a named module, symbols must be exported with `export` if they should be visible outside the named module.
- A simple start with modules is to use header unit.
- Header units do preserve macros, named modules don't.
- Named modules allow us to keep implementation details private inside a module.

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 work 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 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 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%

Output

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

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

The methods automatically update the last points whenever we set the current points via `setPoints`. 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.

```

1  class StockIndex {
2      std::string mName{};
3      double     mLastPoints{};
4      double     mPoints{};
5
6  public:
7      StockIndex(std::string name)
8          : mName{name}

```

Listing 5.2

```
9     {}
10
11    const std::string& name() const { return mName; }
12
13    void setPoints(double points)
14    {
15        mLastPoints = mPoints;
16        mPoints      = points;
17    }
18
19    double points() const { return mPoints; }
20
21    double pointsDiff() const { return mPoints - mLastPoints; }
22
23    double pointsPercent() const
24    {
25        if(0.0 == mLastPoints) { return 0.0; }
26        return (mPoints - mLastPoints) / mLastPoints * 100.0;
27    }
28};
```

Listing 5.2

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

```
1  std::vector<StockIndex> GetIndices()
2  {
3      StockIndex dax{"DAX"};
4      dax.setPoints(13'052.95);
5      dax.setPoints(13'108.50);
6
7      StockIndex dow{"Dow"};
8      dow.setPoints(29'080.17);
9      dow.setPoints(29'290.00);
10
11     StockIndex sp{"S&P 500"};
```

Listing 5.3

```

12     sp.setPoints(3'537.01);
13     sp.setPoints(3'561.50);
14
15     return {dax, dow, sp};
16 }
```

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 does not influence how the compiler sees or treats a number. We are also free on where to place the separator:

```

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

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:

```

1 for(const auto& index : GetIndices()) {
2     std::cout << index.name() << " " << index.points() << " "
3             << index.pointsDiff() << " "
4             << index.pointsPercent() << '%' << '\n';
5 }
```

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 Dow points 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 iostream-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 that does the formatting as desired:

```
1 for(const auto& index : GetIndices()) {  
2     std::cout << std::fixed;  
3     A We need <iomanip> for this  
4     std::cout << std::setprecision(2);  
5  
6     std::cout << std::setw(10) << std::left << index.name()  
7         << " " << std::setw(8) << std::right  
8         << index.points() << " " << std::setw(6)  
9         << index.pointsDiff() << " "  
10        << index.pointsPercent() << '%' << '\n';  
11 }
```

Listing 5.6

As you can see, we have to set `std::fixed` and then `setprecision` to 2 for all that follows. The latter requires an additional header, as A 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` to 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. Seeing what we are formatting here and how the result should look like is very hard for me. 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 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, making debugging and errors even more complicated due to the lack of backup/restore.

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 great. 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<<`. Because 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:

```

1 std::ostream& operator<<(std::ostream& os,
2                               const StockIndex& index)
3 {
4     os << std::fixed;
5     os << std::setprecision(2);
6
7     os << std::setw(10) << std::left << index.name() << " "
8     << std::setw(8) << std::right << index.points() << " "
9     << std::setw(6) << index.pointsDiff() << " "
10    << index.pointsPercent() << '%' << '\n';
11
12    return os;
13 }
14
15 void WithIostreamOperator()
```

```
16  {
17      for(const auto& index : GetIndices()) { std::cout << index; }
18  }
```

Listing 5.7

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:

```
1  for(const auto& index : GetIndices()) {
2      std::cout << std::fixed;
3      std::cout << std::setprecision(2);
4      A Apply the desired locale
5      std::cout.imbue(std::locale("de_DE.UTF-8"));
6
7      std::cout << std::setw(10) << std::left << index.name()
8          << " " << std::setw(8) << std::right
9          << index.points() << " " << std::setw(6)
10         << index.pointsDiff() << " "
11         << index.pointsPercent() << '%' << '\n';
12 }
```

Listing 5.8

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 knowledge about which functions to call. Having operator<< is also great for consistent formatting. However, things get a bit more complicated as soon as localization gets in.

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 what 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`:

```

1 for(const auto& index : GetIndices()) {
2     printf("%-10s %8.2lf %6.2lf %4.2lf%\n",
3            index.name().c_str(),
4            index.points(),
5            index.pointsDiff(),
6            index.pointsPercent());
7 }
```

Listing 5.9

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 signals 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: atomicity. `printf` is a variadic function and takes all its arguments in a single call. That way, we can write them more easily atomically without the risk of interleaved output, as with iostream. The Portable Operating System Interface (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 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 ar-

gument. 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. At the same time, `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 is precisely 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` (Std-Box 3.1 on page 117) for best performance. The format string is a little like `printf` but see for yourself. In Listing 5.10, you see the stock index printing using `std::format`.

```
1 for(const auto& index : GetIndices()) {
2     std::cout << std::format(
3         "{:10}  {:>8.2f}  {:>6.2f}  {:.2f}\n",
4         index.name(),
5         index.points(),
6         index.pointsDiff(),
7         index.pointsPercent());
8 }
```

Listing 5.10

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 to write an open and closing curly brace in the most straightforward case. The library deduces the type and applies the default formatting for the type. The defaults are listed in Table 5.1 on page 155. Automatic type deduction and formatting is the advantage we know from `iostream`. In our case, we

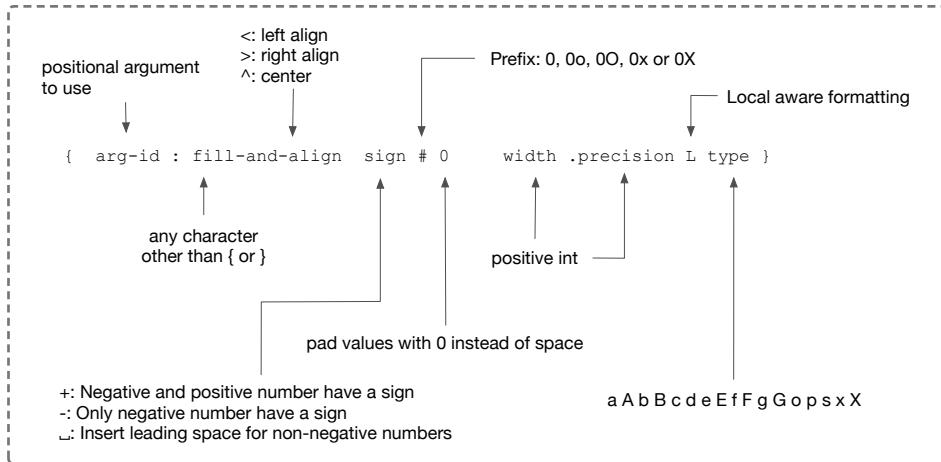


Figure 5.1: Standard format specifiers of `std::format`.

need to specify the string's width, 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. We can specify the type we expect and its format in these braces. Several samples are already shown in the last code example Listing 5.10 on page 153. 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 complete control over the desired output. We now have an option to print the binary representation of an integer. Floating-point numbers allow us various output formats, such as scientific or fixed. A complete list, including the available prefixes, is provided in Table 5.1 on page 155.

Table 5.1: Supported std::format specifiers

Type	Prefix	Meaning	Optional
b	0b	Binary representation	
B	0B	Binary representation	
c		Single character	✓
d		Integer or char	✓
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	✓
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	✓
G		Same as g, just the exponent is indicated with E	
p	0x	Pointer address as hexadecimal representation	✓

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 % by doubling it. This is, of course, an advantage of a system with no format string like `iostream`. As `std::format` uses curly braces to mark a format specifier, we need to escape them as soon as we like to have such a character in the format string. The way this is done is similar to `printf`. The start and end markers are just duplicated.

Listing 5.11

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

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.

If we want a particular argument to be printed in a localized format, we must 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 specific 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.

Listing 5.12

```
1 const double pi = 3.14;
2 const int    i  = 1'024;
3
4 A Create a German locale
5 const auto locDE = std::locale("de_DE.UTF-8"s);
6 B Create a US locale
7 const auto locUS = std::locale("en_US.UTF-8"s);
8
9 std::cout << "double with format(loc, ...)\n";
10 std::cout << std::format(locUS, " in US: {:L}\n", pi);
11 std::cout << std::format(locDE, " in DE: {:L}\n", pi);
12
13 std::cout << "\nint with format(loc, ...)\n";
14 std::cout << std::format(locUS, "1'024 in US: {:L}\n", i);
15 std::cout << std::format(locDE, "1'024 in DE: {:L}\n", i);
16
17 C Simulate a different system locale
18 std::locale::global(locUS);
19 std::cout
20   << "\nint with format(...) after setting global loc\n";
21 std::cout << std::format("1'024 in US: {:L}\n", i);
```

In **A** and **B**, we create two locales, one for German 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 **C**, 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
```

Output

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, it gets cut. However, the following zero is kept if there is only one after the decimal point. Here is some code that illustrates my words:

```
1 const double pi = 3.1400;
2 const double num = 2.0;
3
4 std::string s = std::format("pi {}, num {}", pi, num);
```

Listing 5.13

Below, you see the full output:

```
$ ./a.out
pi 3.14, num 2
```

Output

As we can see, the output of the numbers is then 3.14 and 2.0. That is precisely 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` allows registering custom convert functions in the GNU libc, but that is somewhat clumsy. What does `std::format` provide for us there?

The good news is that `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:

```
1 template<>
2 struct std::formatter<StockIndex> {
3     constexpr auto parse(auto& ctx) { return ctx.begin(); }
4
5     auto format(const StockIndex& index, auto& ctx) const
6     {
```

Listing 5.14

```

7     return std::format_to(ctx.out(),
8                     "{:10}  {:>8.2f}  {:>6.2f}  {:.2f}%", 
9                     index.name(),
10                    index.points(),
11                    index.pointsDiff(),
12                    index.pointsPercent());
13    }
14 };

```

Listing 5.14

Now, we must 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:

```

1 for(const auto& index : GetIndices()) {
2     std::cout << std::format("{}\n", index);
3 }

```

Listing 5.15

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. Suppose we want more than just printing all the information in a stock index for this exercise. 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 before 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.

```

1 template<>
2 struct std::formatter<StockIndex> {

```

Listing 5.16

```

3 enum class IndexFormat { Normal, Short, WithPlus };
4
5 IndexFormat indexFormat{IndexFormat::Normal};
6
7 constexpr auto parse(auto& ctx)
8 {
9     auto it = ctx.begin();
10    auto end = ctx.end();
11
12    if((it != end) && (*it == 's')) {
13        indexFormat = IndexFormat::Short;
14        ++it;
15    } else if((it != end) && (*it == 'p')) {
16        indexFormat = IndexFormat::WithPlus;
17        ++it;
18    }
19
20    A Check if reached the end of the range
21    if(it != end && *it != '}') {
22        throw format_error("invalid format");
23    }
24
25    B Return an iterator past the end of the parsed range
26    return it;
27 }
28
29 auto format(const StockIndex& index, auto& ctx) const
30 {
31     if(IndexFormat::Short == indexFormat) {
32         return std::format_to(ctx.out(),
33                               "{:10} {:>8.2f}"sv,
34                               index.name(),
35                               index.points());
36     } else {
37         const auto fmt{
38             (IndexFormat::WithPlus == indexFormat)
39                 ? "{:10} {:>8.2f} {:>7.2f} {:+.2f}%"sv
40                 : "{:10} {:>8.2f} {:>6.2f} {:.2f}%"sv);

```

```
41
42     return std::vformat_to(
43         ctx.out(),
44         fmt,
45         std::make_format_args(index.name(),
46                               index.points(),
47                               index.pointsDiff(),
48                               index.pointsPercent()));
49     }
50 }
51 };
```

Listing 5.16

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`:

```
1 for(const auto& index : GetIndices()) {
2     std::cout << std::format("{}\n", index);
3 }
4
5 for(const auto& index : GetIndices()) {
6     std::cout << std::format("{:s}\n", index);
7 }
8
9 for(const auto& index : GetIndices()) {
10    std::cout << std::format("{:p}\n", index);
11 }
```

Listing 5.17

In my opinion, this is so much better than tampering via ominous functions with the iostream, risking corrupting 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:

```

1  template<>
2  struct std::formatter<StockIndex> {
3      enum class IndexFormat { Normal, Short, WithPlus };
4      IndexFormat indexFormat{IndexFormat::Normal};
5      A New member to track whether the formatting is localized
6      bool localized = false;
7
8      constexpr auto parse(auto& ctx)
9      {
10         auto it = ctx.begin();
11
12         B Helper to search for a character
13         auto isChar = [&](char c) {
14             if((it != ctx.end()) && (*it == c)) {
15                 ++it;
16                 return true;
17             }
18
19             return false;
20         };
21
22         C Localized formatting
23         if(isChar('L')) { localized = true; }
24
25         if(isChar('s')) {
26             indexFormat = IndexFormat::Short;
27         } else if(isChar('p')) {
28             indexFormat = IndexFormat::WithPlus;
29         }
30
31         if(it != ctx.end() && *it != '}') {
32             throw format_error("invalid format");
33         }
34
35         return it;

```

Listing 5.18

```
36     }
37
38     auto format(const StockIndex& index, auto& ctx) const
39     {
40         D Add localized
41         const auto locFloat{localized ? "L"s : ""s};
42         const auto plus{
43             (IndexFormat::WithPlus == indexFormat) ? "+"s : ""s};
44
45         if(IndexFormat::Short == indexFormat) {
46             const auto fmt =
47                 std::format("{:{10}} {{:>8.2{}f}}"sv, locFloat);
48
49             return std::vformat_to(
50                 ctx.out(),
51                 fmt,
52                 std::make_format_args(index.name(), index.points()));
53
54         } else {
55             const auto fmt{
56                 std::format("{:{10}} {{:>8.2{0}f}} {{:>{1}7.2{0}f}} "
57                     "{{:{1}.2{0}f}}%"sv,
58                     locFloat,
59                     plus)};
60
61             return std::vformat_to(
62                 ctx.out(),
63                 fmt,
64                 std::make_format_args(index.name(),
65                     index.points(),
66                     index.pointsDiff(),
67                     index.pointsPercent()));
68         }
69     }
70 }
```

Listing 5.18

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

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.

```

1 for(const auto& index : GetIndices()) {
2     std::cout << std::format("{:Ls}\n", index);
3 }
4
5 for(const auto& index : GetIndices()) {
6     std::cout << std::format("{:Lp}\n", index);
7 }
```

Listing 5.19

5.4 Referring to a format argument

For the next element of `std::format`, suppose we need to format a couple of shares. 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

Output

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

```

1 for(const auto& share : GetShares()) {
2     std::cout << std::format("{:10} {:>8.2f€} {:>+8.2f€}\n",
3                               share.name(),
4                               share.price(),
5                               share.priceDelta());
6 }
```

Listing 5.21

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:

```
1  for(const auto& share : GetShares()) {  
2      std::cout << std::format(  
3          "{1:10} {2:>8.2f}{0} {3:>+8.2f}{0}\n",  
4          "€",  
5          share.name(),  
6          share.price(),  
7          share.priceDelta());  
8    }
```

Listing 5.22

We can see 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. Special types like `std::stringstream` or `snprintf` 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`:

Listing 5.23

```

1 std::vector<char> buffer{};
2 std::format_to(std::back_inserter(buffer),
3                 "{}, {}",
4                 "Hello",
5                 "World");
6
7 for(const auto& c : buffer) { std::cout << c; }
8
9 std::cout << '\n';

```

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:

Listing 5.24

```

1 constexpr auto fmt("{}{}, {}"sv); A The format string
2
3 B Lookahead the resulting size in bytes
4 const auto size = std::formatted_size(fmt, "Hello"sv, "World"sv)←
5 ;
6 std::vector<char> buffer(size); C Preallocate the required memory
7 std::format_to(buffer.begin(), fmt, "Hello"sv, "World"sv);

```

We first store the format string into a dedicated variable, `fmt`, as shown in A. Then we use `fmt` together with the arguments to invoke `std::formatted_size`, as demonstrated in B. The result is then used to make the `std::vector` pre-allocate the required elements, shown in C. 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` cannot check whether the buffer size 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`:

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

Listing 5.25

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 is valuable, 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`.

```
1 std::array<char, 10> buffer{};  
2 std::format_to_n(buffer.data(),  
3                   buffer.size() - 1,  
4                   "{} {},  
5                   "Hello",  
6                   "World");  
7  
8 std::cout << buffer.data() << '\n';
```

Listing 5.26

The approach using `std::format_to_n` not only gives us safe formatting that stays in bounds but also ensures 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 we need to log some conditions of our financial 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 with a consistent and easy way of formatting the enum `LogLevel`.

```

1  enum class LogLevel { Info, Warning, Error };
2
3  // Part of C++23's STL
4  template<typename T>
5  constexpr std::underlying_type_t<T> to_underlying(T value)
6  {
7      return static_cast<std::underlying_type_t<T>>(value);
8  }
9
10 template<>
11 struct std::formatter<LogLevel>
12 : std::formatter<std::string_view> {
13     inline static std::array levelNames{"Info"sv,
14                                     "Warning"sv,
15                                     "Error"sv};
16
17     auto format(LogLevel c, auto& ctx) const
18     {
19         return std::formatter<std::string_view>::format(
20             levelNames.at(to_underlying(c)), ctx);
21     }
22 };

```

Listing 5.27

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. 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.

Listing 5.28

```
1 void log(LogLevel      level,
2           std::string_view fmt,
3           const auto&... args)
4 {
5     std::clog << std::format("{}: ", sv, level)
6             << std::vformat(fmt, std::make_format_args(args...))
7             << '\n';
8 }
```

This is a working version, easy to create, and thanks to `std::formatter`, the enum is also printed as a string. 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 that `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 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:

Listing 5.29

```
1 void vlog(LogLevel      level,
2           std::string_view   fmt,
3           std::format_args&& args)
4 {
5     std::clog << std::format("{}: ", level)
6             << std::vformat(fmt, args) << '\n';
7 }
8
9 constexpr void
10 log(LogLevel level, std::string_view fmt, const auto&... args)
```

Listing 5.29

```

11  {
12      vlog(level, fmt, std::make_format_args(args...));
13  }
```

With that addition, we help the compiler get the best binary size for us.

Now, using our Log function is similar to using `std::format`:

```

1 void Use()
2 {
3     const std::string share{"Amazon"};
4     const double      price{3'117.02};
5
6     log(LogLevel::Info,
7         "Share price {} very high: {}",
8         share,
9         price);
10
11    errno = 4;
12    log(LogLevel::Error, "Unknown stock, errno: {}", errno);
13 }
```

Listing 5.30

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 case, the question is whether we 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:

```

1 void Use()
2 {
3     const std::string share{"Amazon"};
4     const double      price{3'117.02};
5
6     log(
7         LogLevel::Info, "Share price", share, "very high:", price);
8 }
```

Listing 5.31

```
9     errno = 4;
10    log(LogLevel::Error, "Unknown stock, errno:", errno);
11 }
```

We just pass the arguments in the order we want them 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 it. 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:

```
1 template<size_t Args>
2 constexpr auto makeBraces()
3 {
4     A Define a string with empty braces and a space
5     constexpr std::array<char, 4> c{"{} "};
6     B Calculate the size of c without the string-terminator
7     constexpr auto brace_size = c.size() - 1;
8     C Reserve 2 characters for newline and string-terminator
9     constexpr auto offset{2u};
10    D Create a std::array with the required size for all braces and newline
11    std::array<char, Args * brace_size + offset> braces{};
12
13    E Braces string length is array size minus newline and
14    string-terminator
15    constexpr auto bracesLength = (braces.size() - offset);
16
17    auto i{0u};
18    std::for_each_n(
19        braces.begin(), bracesLength, [&](auto& element) {
20            element = c[i % brace_size];
21            ++i;
22        });
23}
```

Listing 5.32

```

23     braces[bracesLength] = '\n';  F Add the newline
24
25     return braces;
26 }
```

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`:

```

1  void vlog(LogLevel      level,
2           std::string_view   fmt,
3           std::format_args&& args)
4  {
5      std::clog << std::format("{}: ", level)
6          << std::vformat(fmt, args);
7  }
8
9  constexpr void log(LogLevel level, const auto&... args)
10 {
11     A Make the format string
12     constexpr auto braces = makeBraces<sizeof...(args)>();
13
14     vlog(level,
15           std::string_view{braces.data()},
16           std::make_format_args(args...));
17 }
```

Listing 5.33

This allows us to rely on the default formatters for the types. It is excellent for a log function, especially for debugging logs. 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 that adds a timestamp.

```
1 void vlog(std::string_view fmt, std::format_args&& args)
2 {
3     const auto t = GetTime();
4     std::clog << std::format("[{:%Y-%m-%d-%H:%M:%S}] "sv, t)
5             << std::vformat(fmt, args);
6 }
```

Listing 5.34

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

```
$ ./a.out
[2024-02-08-08:19:14.440309] Info Share price Amazon very high: ←
3117.02
[2024-02-08-08:19:14.441875] Error Unknown stock, errno 4
```

Output

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 mainly 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 more manageable and, by default, more correct.

The name spaceship comes from this operator's appearance: `<=>`. The operator looks like one of those star-fighters in a famous space-movie series. The operator is not entirely new; 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 that 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 a class that required comparison operators in the past, 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. How-

ever, consider a basic case where you needed to provide an `operator==` which did something special. For consistency reasons, you usually would have also provided `operator!=`. But what was its implementation? Well, in all cases I can remember, the implementation was this:

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

Isn't that a little sad? That is not special code. It is absolutely trivial but must 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 by imagining we have to write some kind of medical application. You are identified by a unique Medical Record Number (MRN) whenever you enter a hospital. This number is used to identify you during your entire stay 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`, the internal type of the MRN where the value is stored. Implementing a class `MedicalRecordNumber` can be like this:

```
1  class MedicalRecordNumber {
2  public:
3      MedicalRecordNumber() = default;
4      explicit MedicalRecordNumber(uint64_t mrn)
5          : mMRN{mrn}
6      {}
7
8  private:
9      uint64_t mMRN;
10 }
```

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. As defined so far,

objects of the class cannot be compared with anything. The following trivial code, which tests whether two objects represent the same person, fails to compile:

```
1 const MedicalRecordNumber mrn0{};  
2 const MedicalRecordNumber mrn1{3};  
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.

```
1 bool operator==(const MedicalRecordNumber& other) const  
2 {  
3     return other.mMRN == mMRN;  
4 }
```

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:

```
1 bool operator!=(const MedicalRecordNumber& other) const  
2 {  
3     return !(other == *this);  
4 }
```

Listing 6.3

6.1.1 Comparing different types

What we have done so far, adding and implementing `operator==`, is still simple. 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 want 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:

```
1 const bool sameMRNA = mrn0 == 3ul;  
2 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:

```
1  A The initial member functions
2  bool operator==(const MedicalRecordNumber& other) const
3  {
4      return mMRN == other.mMRN;
5  }
6
7  bool operator!=(const MedicalRecordNumber& other) const
8  {
9      return !(*this == other);
10 }
11
12 B The additional overloads for uint64_t
13 friend bool operator==(const MedicalRecordNumber& rec,
14                           const uint64_t&           num)
15 {
16     return rec.mMRN == num;
17 }
18
19 friend bool operator!=(const MedicalRecordNumber& rec,
20                           const uint64_t&           num)
21 {
22     return !(rec == num);
23 }
24
25 C The additional overloads with swapped arguments for uint64_t
26 friend bool operator==(const uint64_t&           num,
27                           const MedicalRecordNumber& rec)
28 {
29     return (rec == num);
30 }
31
32 friend bool operator!=(const uint64_t&           num,
33                           const MedicalRecordNumber& rec)
34 {
35     return !(rec == num);
36 }
```

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 boilerplate 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:

```
1 const bool sameMRN = mRN0 == 3ul;
```

but the other way around wouldn't:

```
1 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, 4 of which 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:

```
1 bool operator==(const MedicalRecordNumber& other) const =
2     default;
3 bool operator==(const uint64_t& other) const
4 {
5     return other == mMRN;
6 }
```

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. We should need four functions instead, which would not be a considerable 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 in Listing 6.5, period. The reasons for it are two new abilities of the compiler operator reverse and rewrite, which are explained in detail in §6.6 on page 192. 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 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 of writing a medical application, consider a class representing 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`, 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 length. Our class is called `String` class with a constructor that 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 that 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`. Currently, `String` does not have a C++20 spaceship operator. Still, 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.

```
1  class String {
2  public:
3      template<size_t N>
4      explicit String(const char (&src)[N])
5          : mData{src}
6          , mLen{N}
7      {}
8
9      A Helper functions which are there for completeness.
10     const char* begin() const { return mData; }
11     const char* end() const { return mData + mLen; }
12
13     B The equality comparisons.
14     friend bool operator==(const String& a, const String& b)
15     {
16         if(a.mLen != b.mLen) {
17             return false; C Early exit for performance
18         }
19
20         return Ordering::Equal == Compare(a, b);
21     }
22
23     friend bool operator!=(const String& a, const String& b)
24     {
25         return !(a == b);
26     }
27
28     D The ordering comparisons.
29     friend bool operator<(const String& a, const String& b)
30     {
31         return Ordering::LessThan == Compare(a, b);
32     }
33
34     friend bool operator>(const String& a, const String& b)
35     {
36         return (b < a);
37     }
38
```

Listing 6.6

```

39     friend bool operator<=(const String& a, const String& b)
40     {
41         return !(b < a);
42     }
43
44     friend bool operator>=(const String& a, const String& b)
45     {
46         return !(a < b);
47     }
48
49 private:
50     const char* mData{};
51     const size_t mLen{};
52
53     E The compare function which does the actual comparison.
54     static Ordering Compare(const String& a, const String& b);
55 };

```

`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 boilerplate code. In fact, for each type we like this class to be comparable with, the number of operators increases by 12. We end up with 30 operators if we want `String` also comparable to a C-style string! All of them simply redirect to the `Compare` function. Okay, 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 be 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 on page 181. 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 §6.4 on page 186. 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 including the `compare` header like this:

```
1 #include <compare>
2
3 auto operator<=>(const String& other) const = default;
```

Listing 6.7

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 an other question.

Whenever the data to which the pointer refers should be compared, `=default` is the wrong solution. In such a case, two pointer addresses can differ, 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.

The member-wise comparison goes through the member variables in declaration order from top to bottom.

This is precisely 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:

```

1  class String {
2  public:
3      template<size_t N>
4      explicit String(const char (&src)[N])
5          : mData{src}
6          , mLen{N}
7      {}
8
9      const char* begin() const { return mData; }
10     const char* end() const { return mData + mLen; }
11
12    auto operator<=>(const String& other) const
13    {
14        return Compare(*this, other); A We already had this
15    }
16
17    bool operator==(const String& other) const
18    {
19        if(mLen != other.mLen) { return false; } B We already had this
20
21        return Compare(*this, other) == 0; C Compare does the work
22    }
23
24    private:
25        const char* mData;
26        const size_t mLen;
27
28        static std::weak_ordering D

```

Listing 6.8

```
29     Compare(const String& a, const String& b);  
30 }
```

Listing 6.8

In the implementation of the spaceship operator, we call the `Compare`-function **A**. 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 perform 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` must also consider the other cases, less than and greater than. These additional cases might be a pessimization for the equality check. Why? Because some optimizations for the equality check are impossible if `Compare` is called. One example is that for the equality check, we can shortcut the check if two objects are of different sizes. This is what **B** 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 that can do this shortcut and return `false` if `a.mLen != b.mLen`. There is no way to do this shortcut in the spaceship operator, and in our case, `Compare`. 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.

One more difference is 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.

6.3.2 Using the STLs comparison function

There is one more thing we can improve thanks to a new library feature in C++20, the `Compare` function. I haven't shown you the implementation and will still hide it. The mere fact that we have to write our own function isn't as good as using something from the STL, namely `lexicographical_compare_three_way`. This new function does exactly what I implemented in `Compare`. By using the STL, we can update our `String` classes spaceship and equal comparison operator as shown in Listing 6.9.

```

1 auto operator<=>(const String& other) const
2 {
3     return std::lexicographical_compare_three_way(
4         begin(), end(), other.begin(), other.end());
5 }
6
7 bool operator==(const String& other) const
8 {
9     return std::strong_ordering::equal == operator<=>(other);
10 }
```

Listing 6.9

6.4 The different comparison categories

The different comparison category types the `<compare>`-header offers establish a system on 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 a type should 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` we saw earlier. A type without comparison operators is another option. Such types are unordered and not equality comparable.

After determining 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 name and perform a case-sensitive comparison, which would lead to another result.

For comparison-strength decisions, 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 a 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 Matryoshka dolls. They are made of wood and usually colorfully painted. They come in an egg-like shape and are nestable. You either start or end up with a single piece that 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`. There is no ordering between the values 0 and -0 of a `float`. 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 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 that returns `std::partial_ordering`. The same applies to `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 on page 189 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 on page 189.

A real-world example of partial ordering is dressing. We have to put on underwear before we can put on pants. That part is orderable. It starts to get tricky 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 unordered parts in the clothing set.

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 §6.3.1 on page 183,

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

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.

This is one way to check the value of, in this case, `std::weak_ordering`. An alternative is 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
<code>is_eq(partial_ordering cmp)</code>	<code>cmp == 0</code>
<code>is_neq(partial_ordering cmp)</code>	<code>cmp != 0</code>
<code>is_lt(partial_ordering cmp)</code>	<code>cmp < 0</code>
<code>is_lteq(partial_ordering cmp)</code>	<code>cmp <= 0</code>
<code>is_gt(partial_ordering cmp)</code>	<code>cmp > 0</code>
<code>is_gteq(partial_ordering cmp)</code>	<code>cmp >= 0</code>

Applied to the `String` example, the implementation of `operator==` uses `is_eq` instead of comparing to zero:

```

1  bool operator==(const String& other) const
2  {
3      if(mLen != other.mLen) { return false; }
4
5      A Using a named comparison function

```

```

6     return std::is_eq(Compare(*this, other));
7 }
```

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 your code base or coding style better.

6.5 Converting between comparison categories

There are conversion rules between the different categories. A closer look at Table 6.2 on page 189 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 illustrates 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 that are equivalent can be translated into

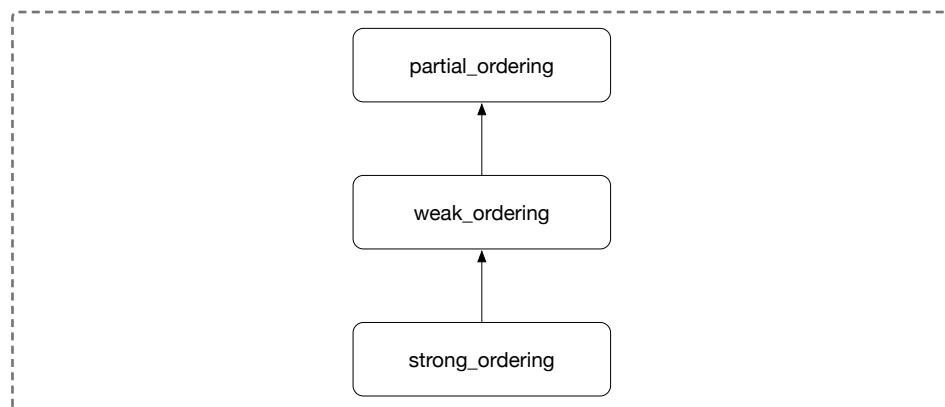


Figure 6.1: The different comparison categories and how they relate to each other. Arrows show an *is-a* implicit conversion ability.

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 particular 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.

```
1  struct Weak {
2      std::weak_ordering operator<=>(const Weak&) const = default;
3  };
4
5  struct Strong {
6      std::strong_ordering
7      operator<=>(const Strong&) const = default;
8  };
9
10 struct Combined {
11     Weak    w;
12     Strong s;
13
14     auto operator<=>(const Combined&) const = default;
15 }
```

Listing 6.11

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. A type-trait can also identify the common comparison category for you: `std::common_comparison_category`.

In case you would, for example, specify `strong_ordering` as the 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. Suppose there is only a user-provided `operator==`. In that case, 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's 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 why 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:

```
1 const bool sameMRN = mrn0 == 3ul;
2 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 §6.1.1 on page 177. In C++20, regardless of which variant we pick, the compiler is allowed to reverse the operands such that both variants end up like this:

```
1 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 on page 193 in the 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 on page 193, 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	<code>==</code>	<code><=></code>
Secondary	<code>!=</code>	<code><, >, <=, >=,</code>

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 the 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 greatly helps for simple classes or structs that 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 imple-

mentation 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 time you write a wrapper-like class that 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:

```

1  class BCD {
2  public:
3      BCD(int v)
4          : mValue{Adjust(v)} {}
5
6
7      A Make BCD convertible to int
8      operator int() const { return mValue; }
9
10
11     B Provide at least equality comparison
12     bool operator==(const BCD& rhs) const
13     {
14         return rhs.mValue == mValue;
15     }
16
17     bool operator!=(const BCD& rhs) const
18     {
19         return not(*this == rhs);
20     }
21
22     private:
23         int mValue;
24
25     static int Adjust(int v);
26 };

```

Listing 6.12

A is just to make BCD convertible back to an `int`. You can also envision 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 easy again; 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:

```
1 class BCD {
2 public:
3     BCD(int v, int significance)
4         : mSignificance(significance)
5         , mValue{Adjust(v)}
6     {}
7
8     operator int() const { return mValue; }
9
10    A Provide at least equality comparison
11    auto operator<=>(const BCD&) const = default;
12
13 private:
14     int mSignificance; B The additional member just works
15     int mValue;
16
17     static int Adjust(int v);
18 }
```

Listing 6.13

We added the new member and defaulted the spaceship operator, done. We got rid of the ridiculous boilerplate 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 paste `constexpr` to all the comparison operators. Here is what you do with the spaceship operator:

```
1 constexpr auto
2 operator<=>(const BCD&) const = default; A Add constexpr
```

Listing 6.14

All that is left to do is add `constexpr` one time A, 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:

```
1 struct Address {
2     std::string city;
3     std::string street;
4     uint32_t    street_no;
5
6     auto operator<=>(const Address&) const = default;
7 };
```

Listing 6.15

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 names. It then sorts the street number in ascending order. Let's assume that street numbers should be sorted in descending order. We can achieve this by providing our own implementation for `operator<=>`:

```
1 struct Address {
2     std::string city;
3     std::string street;
4     uint32_t    street_no;
5
6     auto operator<=>(const Address& rhs) const
7     {
8         A Sort city and street using their <=>
9         if(const auto& cmp = city <=> rhs.city; cmp != 0) {
10             return cmp;
11         } else if(const auto& scmp = street <=> rhs.street;
12                 scmp != 0) {
13             return scmp;
14     }
15 }
```

Listing 6.16

```
14     }
15
16     B Next, sort street_no ascending
17     return rhs.street_no <=> street_no;
18 }
19
20     C The default should be good enough here
21     bool operator==(const Address&) const = default;
22 }
```

Listing 6.16

Thanks to the availability of the spaceship operator in the STL containers, we can invoke them for `std::string` **A**. By combining it with C++17's `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 **B** shows, this is achieved by swapping the arguments to `<=>`. You can further see that the spaceship operator is also defined for built-in types.

This is one case where it is okay to default `operator==` **C**, 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:

```
1 struct Legacy {
2     int a;
3
4     A These define a weak order
5     bool operator==(const Legacy&) const;
6     bool operator<(const Legacy&) const;
7 }
```

Listing 6.17

The two operators ❶ 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. Now, if we have to write a new class `ShinyCppClass`, 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 who does the three-way comparison for each of the struct's fields. This is writing boilerplate code. Thanks to a last-minute update to the standard, we can just default the spaceship operator:

```

1  class ShinyCppClass {
2      Legacy mA;
3      Legacy mB;
4
5  public:
6      ShinyCppClass(int a, int b);
7
8      std::weak_ordering ❶
9      operator<=>(const ShinyCppClass&) const = default;
10 }

```

Listing 6.18

The only special thing is that we must be specific about the return type of `operator<=>`. We cannot just say `auto`. ❶ shows that in this case we apply `weak_ordering` as the return type. This is what `Legacy` allows us. This allows us 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 marvelous thing. Consistent comparisons make the language stronger and reduce 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:

Listing 6.19

```
1 struct A {
2     bool operator==(B&) const { return true; } A
3 };
4
5 struct B {
6     bool operator==(const A&) const { return false; } B
7 };
```

The piece of code in Listing 6.19 is questionable, not because of the silly return of 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? The code stops compiling if there is a const B object. Assuming 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 the equality operator allows the compiler to reverse the arguments. When it does this with the code in Listing 6.19, 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 none is 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. The compiler will point out all these inconsistencies by upgrading your code to C++20. The assumption is that such things happen only very, very rarely.

Cliff notes

- Consistent comparisons are a valuable feature that can take a lot of boilerplate 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 who take the class itself as an argument.

- `constexpr` can be added or removed when using `=default`.
- Remember that `=default` performs 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. This is all you need since they provide the full complement of all comparisons, and the compiler can rewrite them.
- 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 first introduced into C++ with C++11, they have improved with every successive standard since then. C++20 is no exception. There are several changes therein 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. Lambdas are handy helpers whenever we need to create a new type with only a subset of values and a specific action. The correct and/or best capture way is a complex 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 that is 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 of whether we use it in the end. We can end up with unused variables in the lambda, which creates 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 variables used 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 worry. 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 capture-by-copy exactly means. In this simple case, it means that a particular 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. Here is a lambda `twice`, which multiplies the value of `c` by two.

```
1 int c{3};
2
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 class member? What does it actually capture if we write the same `twice` lambda with the same capture? Such a scenario might look like this:

```
1 class SomeClass {
2     int c{3};
3
4 public:
5     void SomeCleverMethod()
6     {
7         auto twice = [=] {
8             A Implicitly capturing this here instead of just c
9             return c * 2;
10        };
11
12        // ...
13        const auto v = twice();
14    }
15};
```

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 instantiations visible are just two among a lot of things it does. There is a command-line version available, and there is 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 of 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 access 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 that 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.

```
1  class SomeClass {
2      int c;
3
4  public:
5      void SomeCleverMethod()
6  {
7      A Lambda internals created by the compiler
8      class __lambda_8_18 {
9          public:
10         int operator()() const { return __this->c * 2; }
11
12     private:
13         B The captured object, a pointer to SomeClass
14         SomeClass* __this;
15
16     public:
```

Listing 7.3

Listing 7.3

```

17     __lambda_8_18(SomeClass* _this)
18     : __this{_this}
19     {}
20 };
21
22 __lambda_8_18 twice = __lambda_8_18{this}; C Passing this
23 const int v           = twice.operator()();
24 }
25 };

```

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 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 clarifies the intention 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. You cannot be sure that the code does what we intend it to. As a programmer, you have no way to express *Yes, I like to copy only the this pointer 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 precise 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 a pointer.

Listing 7.4

```

1 auto twice = [=, this] A Explicit capture this as pointer
2 { return c * 2; };

```

The behavior that `=` implies 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 re-

moved with, say, C++23. Upgrading now not only spares you a break later but 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 with titles and an International Standard Book Number (ISBN). Then we have two prices, a standard price of 34.95 and a reduced price of 24.95. Book and price are two independent structs in our code. We are asked to create a map between books and the related prices. 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`. A user must supply only the value's type 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 wish to have different ways of sorting books, we use a custom compare function. This is essentially a one-liner, so we use 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` its type in the template argument list. We can use `decltype` to get the type of `cmp`. Here is a possible implementation:

```
1  struct Book {
2      std::string title;
3      std::string isbn;
4  };
5
6  struct Price {
7      double amount;
8  };
9
10 auto cmp = [](const Book& a, const Book& b) A Compare lambda
11 { return a.isbn > b.isbn; };
12
```

Listing 7.5

Listing 7.5

```

13 template<typename VALUE>
14 using MapBookSortedByIsbn = std::map<Book,
15                                     VALUE,
16                                     decltype(cmp)           B Typed map for multiple use
17                                     >;                         C Use Book as type for the map
18                                         D The value can be provided
                                         E Type of the custom compare function

```

The idea of the `using`-alias, which creates `MapBookSortedByIsbn` **B** is that a potential user has to type less. We can provide a consistent element that 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, you 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 the construction of the map. Not only that, a user needs to pass `cmp` as a constructor argument, so they also need to know the function's name. The initial concept of having a type that says, "I'm a map of books sorted by ISBN" implies that a user need not 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, enabling the compiler to tell a user about the mistake.

Here is an example that uses two books. A regular and a reduced-price book. In the creation of the map **A**, we add the two books **B**, but then have to know and pass the compare function **C**.

```

1 const Book effectiveCpp{"Effective C++", "978-3-16-148410-0"};
2 const Book fpCpp{"Functional Programming in C++",
3                  "978-3-20-148410-0"};
4
5 const Price normal{34.95};
6 const Price reduced{24.95};
7
8 A Use the map with Price as value
9 MapBookSortedByIsbn<Price> book2Price{
10   {{effectiveCpp, reduced},
11    {fpCpp, normal}},   B Add some items to it

```

Listing 7.6

```
12     cmp  C Sadly we have to know and pass the cmp function
13 };
```

Listing 7.6

The question is, why does this have to be so complicated? Why do we need to pass `cmp` 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 constructor's argument list. With that, users 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.

```
1 MapBookSortedByIsbn<Price> book2Price{
2     {effectiveCpp, reduced},
3     {fpCpp, normal}
4     A cmp is gone
5 };
```

Listing 7.7

We now have a clean map for 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 so. The compare function `cmp`, or more precisely lambda. We created a callable by now, but what did we do with it? We used `decltype` to pass its type to `std::map`. So, we never needed a callable. We only needed a type of lambda to store it in `cmp`. As a result, we have a variable in the scope with the name `cmp`. What can we do about this? Two different solutions come to mind. First, and probably most obvious, instead of creating a variable `cmp`, move the lambda's definition directly into the `decltype`-expression when defining the alias.

Listing 7.8

```

1 template<typename VALUE>
2 using MapBookSortedByIsbn =
3     std::map<Book,
4             VALUE,
5             A Provide the lambda in-place
6             decltype([](const Book& a, const Book& b) {
7                 return a.isbn > b.isbn;
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 before 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`:

```

1 A Define a using alias with the compare lambda which can be reused
2 using cmp = decltype([](const Book& a, const Book& b) {
3     return a.isbn > b.isbn;
4 });
5
6 template<typename KEY, typename VALUE>
7 using MapSortedByIsbn = std::map<KEY,
8                                     VALUE,
9                                     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, which makes lambdas first-class citizens.

Please note that lambdas in a `decltype` expression, or in an unevaluated context in general, only work with captureless lambdas. Lambdas with captures need arguments during construction, the captures of 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 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`:

```
1 struct Magazine {  
2     std::string name;  
3     std::string isbn;  
4 };
```

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.

```
1 const Book effectiveCpp{"Effective C++", "978-3-16-148410-0"};  
2 const Book fpCpp{"Functional Programming in C++",  
3                  "978-3-20-148410-0"};  
4  
5 const Magazine ix{"iX", "978-3-16-148410-0"};  
6 const Magazine overload{"overload", "978-3-20-148410-0"};  
7  
8 const Price normal{34.95};  
9 const Price reduced{24.95};  
10  
11 MapSortedByIsbn<Book, Price> book2Price{{effectiveCpp, reduced},  
12                                         {fpCpp, normal}};  
13 MapSortedByIsbn<Magazine, Price> magazine2Price{  
14     {ix, reduced},  
15     {overload, normal}};
```

Listing 7.11

Making the code in Listing 7.11 possible requires some updates to `MapBookSortedByIsbn` as it currently only works with books. To make `MapBookSortedByIsbn` more flexible, one part requires 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`:

We need to change a second part 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. By making the former `Book` parameters `auto`, the code compiles:

```

1 template<typename KEY, typename VALUE>
2 using MapSortedByIsbn = std::map<
3     KEY,
4     VALUE,
5     decltype([](const auto& a, A Using a generic lambda
6                 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 were the only place where `auto` was valid as a parameter type before C++20. It essentially generates a templated call operator, where each `auto`-parameter becomes a template-type parameter, allowing 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 NTTT 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:

Listing 7.15

```
1 template<typename KEY, typename VALUE>
2 using MapSortedByIsbn =
3     std::map<KEY,
4             VALUE,
5             decltype([]<typename T> { A Lambda with a template-head
6                               (const T& a,      B Use T as in a regular template
7                               const T& b) { return a.isbn > b.isbn; })>;
```

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 that 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 a partial application.

Let's write a `print` function for a vehicle that 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, we want to insert the origin as the first argument to identify the log source, steering, or brake system. The straightforward solution would be to write the origin we need as the first argument of each `print` invocation. That works but hides some risks.

In the simplest solution, we repeatedly duplicate the origin name. 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 so that this variable can then be used. Better, but now we have a global variable with its drawbacks, such as the initialization order. Plus, `print` can still be invoked without an origin. Do you want to take a bet that there was only one place where this happened when you got a trace and needed to identify where it came from? In my experience, chances are high that there is more than one place.

Listing 7.14

```

1  A Passing differently spelled origins
2  print("Steering"s, "angle"s, 90);
3  print("steering"s, "angle"s, 75);
4
5  B Declaring a global variable for the steering origin
6  static const auto originSteering{"Steering"s};
7
8  C Ok, use of the global variable
9  print(originSteering, "angle"s, 90);
10
11 D Passing steering instead of Steering
12 print("steering"s, "angle"s, 75);

```

All these approaches come with drawbacks. Users can still pass a plain string even with a global variable for an origin. The API of `print` is not obvious. This holds even in the 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.

Listing 7.15

```

1  auto getNamedLogger(const std::string origin)
2  {
3      return [=](auto... args) {
4          print(origin,
5              A Forward the arguments using decltype to retrieve the type
6              std::forward<decltype(args)>(args)...);
7      };
8  }

```

Perfect forwarding

Since we have move-operations with C++11, we can perfect forward objects. The term comes down to the fact that copies, and with that, potential allocations and deallocations, are reduced, either for temporary objects or for objects 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 that captures the origin as a copy. That way, a user can create a named logger, pass the origin, and 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 objects when we forward them, saving additional allocations and copies. The example Listing 7.15 on page 212 is a possible C++17 implementation that uses perfect forwarding.

Now, `getNamedLogger` can be used by calling `getNamedLogger` with, for example, "Steering" as a parameter to create a logger for the steering gear. We store the result in an 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 `brakeLogger` without the need to know about `getNamedLogger` and what value to pass as origin for which system.

```
1  A Logger for steering
2  auto steeringLogger = getNamedLogger("Steering"s);
3  B Logger for brakes
4  auto brakeLogger = getNamedLogger("Brake"s);
5
6  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`. Admittedly, 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 regular 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 a parameter type to the chosen `Ts`:

```
1  auto getNamedLogger(const std::string origin)
2  {
3      return [=]<typename... Ts>(Ts... args)
4      {
5          print(origin, std::forward<Ts>(args)...);
6      }
7  }
```

Listing 7.17

Listing 7.17

```

6     } ;
7 }
```

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 many 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 want 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 fulfill the new requirement, `getNamedLogger` needs to itself become a variadic template. All the origins are then captured by the lambda inside `getNamedLogger`, and the pack is expanded to the `print` function that the lambda executes. Here is a possible implementation:

```

1 template<typename... Origins>
2 auto getNamedLogger(Origins&&... origins)
3 {
4     return [=]<typename... Ts>(Ts && ... args)
5     {
6         print(origins..., std::forward<Ts>(args)...);
7     };
8 }
```

Listing 7.18

With just a few tweaks, we managed to add the new requirements to `getNamedLogger`. There is a but. In §7.4.3 on page 213, we enhanced the

former solution of `getNamedLogger` to perfectly forward to the `print`-function the variadic arguments that are passed to the lambda. This was for the arguments to the lambda itself. By making `getNamedLogger` a variadic template and letting the lambda capture the parameter pack of `getNamedLogger`, there is another opportunity to waste resources or to apply perfect forwarding. Assume that either we passed larger `std::string` objects as origins, perfect forwarding will save additional allocations. The same is true for any type that must 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 argument is validly moved.

7.2 C++14: Lambda init-capture

Lambda init-captures were introduced with C++14. They allow the creation of a new variable inside the lambda during its creation using the capture list. The type of such an init-capture is deduced by `auto`. It is unnecessary to choose a name different than the one that is used to initialize the init-capture.

```
1 int x = 3;
2
3 auto l1 = [y = x A A lambda init-capture, creating a new
4           variable y with x as value
5           ] { return y; };
6
7 auto l2 = [x = x B The same name of x inside the lambda
8           ] {
9     C Here x refers to the x of the lambda, not
10    to the x in the outer scope
```

Listing 7.19

Prior to C++20, there was no straightforward solution. Many of us have naturally tried to apply `std::forward` to the parameter pack in the capture list of the lambda using an init-capture, like this:

```
1 return [_origins=std::forward<Origins>(origins...)]
2                           <typename... Ts>(Ts... args)
```

There are other possible variations. 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` **A**. Inside this first lambda, another lambda is required **B**. This second lambda is supplied to `std::apply` inside our original lambda to expand all the values of the wrapping tuple. It receives the tuple elements as a pack in its parameter list `_origins`. In code, this looked like this:

```

1  template<typename... Origins>
2  auto getNamedLogger(Origins&&... origins)
3  {
4      A Create an init-capture of tuple and move origins into it
5      return [tup = std::make_tuple(std::forward<Origins>(
6          origins)...)]<typename... Ts>(Ts && ... args)
7      {
8          std::apply(
9              B A second lambda which is applied to the tuple values
10             [&](const auto&... _origins) {
11                 print(_origins..., std::forward<Ts>(args)...);
12             },
13             tup);
14     };
15 }
```

Listing 7.20

It works, but it is 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.

```

1  template<typename... Origins>
2  auto getNamedLogger(Origins&&... origins)
3  {
4      return [... _origins = std::forward<Origins>(
5          origins)]<typename... Ts>(Ts && ... args)
6      {
7          print(_origins..., std::forward<Ts>(args)...);
8      };
9  }
```

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 with the natural code I showed in Listing 7.21 on page 216 is that we must tell the compiler that `_origins` is a pack. To do that, we must 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.

```
1 auto steeringLogger = getNamedLogger("Steering"s);
2 steeringLogger("angle"s, 90);
3
4 auto brakeLogger = getNamedLogger("Brake"s, "Left"s, "Front"s);
5 brakeLogger("force", 40);
```

Listing 7.22

In Listing 7.21 on page 216, 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 the lifetime of the variables passed to `getNamedLogger` and `getNamedLogger` itself. The arguments we pass to `getNamedLogger` must have a longer lifetime 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` stores 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_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.

```

1  template<typename... Origins>
2  auto getNamedLogger(Origins&&... origins)
3  {
4      return [... _origins = std::forward<Origins>(
5          origins)]<typename... Ts>(Ts && ... args)
6      A Trailing requires with disjunction and is_pointer to limit Origins
7      to no pointers
8      requires(
9          not std::disjunction_v<std::is_pointer<Origins>...>
10         {
11             print(_origins..., std::forward<Ts>(args)...);
12         }
13     }

```

Listing 7.23

Logging usually has to be fast. However, formatting floating-point numbers usually isn't. For this reason, let's limit `getNamedLogger` to accept only arguments that are not floating-point 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.

```

1  return [... _origins = std::forward<Origins>(origins)]
2    <typename... Ts>
3    A Requires-clause with disjunction of is_floating_point to limit Ts
4    requires(not std::disjunction_v<
5      std::is_floating_point<Ts>...>) (
6      Ts&&... args)
7    B Trailing requires with disjunction and is_pointer to limit Origins to
8      no pointers
9    requires(not std::disjunction_v<std::is_pointer<Origins>...>)
10   {
11     print(_origins..., std::forward<Ts>(args)...);
12   };

```

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 the `is_floating_point` we previously used in the requires-clause. However, this time using the concept is simpler. The concept applies to each parameter in the pack automatically. We need not do it ourselves and can drop `std::disjunction`.

```

1  template<typename T>
2  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 the concept, and we are done.

```

1  return [... _origins = std::forward<Origins>(origins)]
2    A Type-constraint NotFloatingPoint to restrict all parameters not to be
     floats
3    <NotFloatingPoint... Ts>(Ts && ... args) requires(
4      not std::disjunction_v<std::is_pointer<Origins>...>)
5    {
6      print(_origins..., std::forward<Ts>(args)...);
7    };

```

Listing 7.26

As you can see, we can apply Concepts to lambdas as to a normal template. With them, we constrain the lambda, disallowing floating-point types and disallowing pointers as arguments as well. Thanks to Concepts, we do not need to commit to one single type or a base class.

Table 7.1: Lambda Captures

Capture	Description	11	14	17	20
[]	Empty lambda	✓	✓	✓	✓
[foo]	Copy foo	✓	✓	✓	✓
[&foo]	foo as reference	✓	✓	✓	✓
[=]	Copy all variables used in the lambda body	✓	✓	✓	✓
[&]	All variables used in the lambda body as references	✓	✓	✓	✓
[this]	Data and members of the surrounding class as references	✓	✓	✓	✓
[*this]	Data and members of the surrounding class as deep copy			✓	✓
[=, *this]	Copy all variables used in the lambda body, this as deep copy			✓	✓
[=, this]	Copy all variables used in the lambda body, this by reference				✓
[y = x]	Create y as new variable initialized by x		✓	✓	✓
[&y = x]	Create y as new reference variable initialized by x		✓	✓	✓
[...y = pack]	Create y as new pack, initialized by a pack				✓

C++20 makes lambdas even more powerful. Some restrictions were lifted, making lambdas appear increasingly as first-class citizens. Captures were 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 into 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 in the same manner as does every other element of C++.

Chapter 8

Aggregates: Designated initializers and more

Initialization is a vast 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`.

The fact 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:

```
1 struct Aggregate {
2     int a = 5;   A Equal in-class initializer
3     int b{7};   B Braced in-class initializer
4 };
```

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? A special member function is user-declared if we just use `=default` to retain a default constructor. Once we also provide the implementation, it is called user-provided.

```
1 struct UserProvided {
2     UserProvided()
3     { /* ... */
4     }
5 };
6
7 struct UserDeclared {
8     UserDeclared() = default;
9 };
```

Listing 8.2

8.2 Designated initializers

Now that we have reiterated what aggregates are, let's talk about one interesting change C++20 made. To some extent, designated initializers are a C compatibility fix in an area where C and C++ were different for as long as C++ existed.

Have a look at the following example:

Listing 8.3

```
1 struct Point { A Point is an aggregate
2     int y;
3     int x;
4     int z;
5 };
6
7 const Point p1{3, 0, 4};
```

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`? Take another look at the code for all those thinking `0`. I tricked you! Instead of declaring the members in the usual order `x,y,z`, I used `y,x,z`! Why? Because I can, and things like this happen in real life. My point is that we cannot tell what gets initialized just by looking at the initialization as it is provided. The same is technically true for classes, but we usually work with constructors. There, we can fix the initialization order of members.

So what can we do? Writing coding guidelines would be a step but not a really helpful one. The example Listing 8.3 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 used to program in C, you know a solution. C has designated initializers for `structs`. There, we can write the following:

Listing 8.4

```
1 struct Point { A Point is an aggregate
2     int y;
3     int x;
4     int z;
5 };
6
7 const Point p1{.y = 3, .x = 0, .z = 4};
```

The part `.x = 0` is called a designated initializer. It gives us the power to explicitly name the member we wish to initialize. For all those thinking about 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. I had used designated initializers more or less all the time when I hit a compile error that took me a while 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 they are 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 Listing 8.4 on page 225 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 often the constructor is called for that designator's 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 vexing parse problem. Below, you see an example of the most vexing parse issue.

```
1 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`.

```
1 struct Point {
2     int y;
3     int x = 2;  A Using in-class member initialization
4     int z;
5 };
6
7 struct NamedPoint {
8     std::string name;
9     Point pt;
10};
11
12 B Initializing with designated initializers
13 const Point p0{3, 0, 4};
14
15 C Initializing all members with designated initializers
16 const Point p1{.y = 3, .x = 0, .z = 4};
17 const Point p2{.y{3}, .x{0}, .z = 4};
```

Listing 8.6

```

18
19 D Initializing a subset with designated initializers
20 const Point p3{.y = 3, .z = 4};
21
22 E Different order as defined in Point will not compile
23 // const Point p4{.x = 0, .y = 3, .z = 4};
24
25 F Designated initializers appears more than once, will not compile
26 // const Point p5{.y = 3, .y = 4};
27
28 G Nested designated initializers
29 const NamedPoint p6{.name = "zero", .pt{.y{0}, .z{0}}};
30
31 H Designated initializers for the outer aggregate
32 const NamedPoint p7{.name = "zero", .pt{0, 0, 0}};

```

First, this time, we use default member initialization for the member `x` in **A**. We remember that this has been 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 also use braced initialization, 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 default member initialization to 2. The designated initializers allow us to omit `x` and only initialize all other members. This is a considerable improvement over the earlier standards. Finally, default-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**, the 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 definitions 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 default 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 default member initialization for its member `x`. In that former example, we saw `p3`, which initializes only `y` and `z`, leaving `x` with the default 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:

```
1 Point p3{};  
2 p3.y = 3;  
3 p3.z = 4;
```

Listing 8.7

We must create a non-`const` version of `p3` and initialize the desired members. By doing that, not only do we lose the `constness`, 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 to keep the `constness` and duplicate the values of the default member initialization by specifying all members:

```
1 const Point p3{3, 0, 4};
```

Listing 8.8

Suppose we initialize all the members of the aggregate. In that case, 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 default member initialization uses 2! What is correct now? Whenever

we approach such a piece of code, we cannot easily tell. In this case, I intentionally made the *mistake*. 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 others use their defaults. By listing the designated initializers, such mistakes are effectively reduced.

8.2.4 Initialize a subset with designated initializers without default member initializers

Defaults are a good next question. What if I had not provided a default for `x` as an default member initializer and left that member out from the designated initializers? Like this:

```

1  struct Point {
2      int y;
3      int x;  A No in-class member initializer
4      int z;
5  };
6
B Initializing a subset with designated initializers
8  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. We face zero initialization here due to the `int` we have. Even with designated initializers, we have no uninitialized members. Whether an unnamed member's value after initialization matches our expectation is a different question.

Return value optimization

The term Return value optimization (RVO) refers to an old optimization technique of compilers. Have a look at the following example:

Listing 8.10

```
1  A Type is neither copy nor movable
2  struct NonCopyableOrMoveable {
3      NonCopyableOrMoveable() = default;
4      NonCopyableOrMoveable(const NonCopyableOrMoveable&) =
5          delete;
6      NonCopyableOrMoveable(NonCopyableOrMoveable&&) = delete;
7      NonCopyableOrMoveable&
8      operator=(const NonCopyableOrMoveable&) = delete;
9      NonCopyableOrMoveable&
10     operator=(NonCopyableOrMoveable&&) = delete;
11     ~NonCopyableOrMoveable()           = default;
12 };
13
14 NonCopyableOrMoveable RVO()
15 {
16     return {};  B This is where RVO happens
17 }
18
19 void Use()
20 {
21     C The return-object is created directly at myValue
22     auto myValue = RVO();
23 }
```

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.

Utilizing Return Value Optimization thanks to designated initializers

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**.

Listing 8.11

```

1 Point GetThePoint() A Returning a Point utilizing RVO
2 {
3     return { .y = 3, .z = 4 };
4 }
```

Another application is pushing elements to a `std::vector` or another STL container.

Listing 8.12

```

1 auto GetVectorOfPoints()
2 {
3     std::vector<Point> points{};
4
5     A Create a Point in-place of a container
6     points.emplace_back(Point{.y = 5, .z = 6});
7
8     return points;
9 }
```

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 an `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 before with `const` that we can create and initialize an aggregate thanks to designated initializers with a single statement has more advantages. Before, I only talked about making the object `const`, but there are more places where creating 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:

Listing 8.13

```

1 void FileAccess(bool open, bool close, bool readonly);
2
3 void Use()
```

```
4  {
5      FileAccess(true, false, true);
6  }
```

Listing 8.13

In Listing 8.13 on page 232, along with the function, we see how calling that function looks with `Use`. I doubt someone can tell which `bool` is for what and if the values make sense here. Think about a code review situation where you must 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:

```
1  FileAccess(/*open*/ true, /*close*/ false, /*readonly*/ true);
```

Listing 8.14

The compiler ignores comments. They may get it wrong over time. Maintaining all the call sites and ensuring that the parameter comments still reflect the current situation is challenging. Reading such inline comments as the previous listing 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:

```
1  struct FileAccessParameters {
2      bool open;
3      bool close;
4      bool readonly;
5  };
6
7  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 function parameter. That way, we have won nothing. However, we can now use designated initializers when creating the object. Here is this version:

Listing 8.16

```
1 FileAccess({.open = true, .close = false, .readonly = true});
```

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 man's solution to named arguments in C++.

An aggregate with designated initializers can be a nice way to model optional arguments for a function. 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 would also like to point out here that using strong types instead of bools 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:

```
1 struct Point2D {
2     int X;  A Note that this is a capital X
3     int y;
4 };
5
6 B Two functions which overload
7 void Add(const Point& p, int v);
8 void Add(const Point2D& p, int v);
9
10 void Use()
11 {
12     Add({.X = 3, .y = 4}, 3);  C Fine, selects Point2D
13     Add({.y = 4, .x = 3}, 3);  D Fine, selects Point
14
15 E Will not compile as solely .y is ambiguous
16 // Add({.y = 4}, 3);
17 }
```

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 resolve which type we refer to. 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 can 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 quickly fill books. Since C++11, we have braced initialization for many 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:

```
1 std::vector<int> v1(35);  A Using parentheses  
2 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 **A** and curly braces in **B**. 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 from 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 too boring for you, stay with me for another minute. Here is the code I expect you all know and have written dozens of times:

```
1 auto pt = std::make_unique<Point>(4, 5);
```

Listing 8.19

Now the question is, not what does it do, 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
```

Output

```
<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
```

Output

It puzzled me many times to recall what I did wrong again. This error is because internally when initializing `Point`, `make_unique` uses parentheses to initialize the freshly created object. You can reduce it to the following line of code:

```
1 const auto* pt = new Point(4, 5);
```

Listing 8.2

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 writing a generic perfect forwarding function impossible.

Generic perfect forwarding

If you desire to write a generic perfect forwarding function pre-C++17, you can use `std::is_constructible` 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 tough 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 supports aggregate initialization from a parenthesized list. It makes the code in Listing 8.21 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 vexing parse issue still remains in C++20.

In Listing 8.22 on page 238, you see an overview of the different initialization forms and their results in C++20.

```

1  struct Point {
2      int y;
3      int x;
4      int z;
5  };
6
7  struct Nested {
8      int i;
9      Point pt;
10 };
11
12 struct LifeTimeExtension {
13     int&& r; A Notice the r-value reference
14 };
15
16 B Initialization of an aggregate
17 Point bPt{2, 3, 5};
18 Point pPt(2, 3, 5); C Since C++20
19
20 D Initialization of an array
21 int bArray[]{2, 3, 5};
22 int pArray[](2, 3, 5); E Since C++20
23
24 Nested bNested{
25     9,
26     {3, 4, 5}}; F Initialization of nested aggregate with nested braces
27 // Nested pNested(9, ( 3,4,5));G Nested parentheses are a different
28     thing
29
30 Point bDesignated{.y = 3}; H Designated initializers works
31 // Point pDesignated(.y=3);I Designated initializers are not supported
32
33 // Point bNarrowing{3.5};J Does not allow narrowing
34 Point pNarrowing(3.5); K Allows narrowing
35
36 Point bValueInit{}; L Default or zero initialization
37 Point pValueInit(); M Still a function declaration
38

```

```
38  N Initialization of a built-in type
39  int bBuiltIn{4};
40  int pBuiltIn(4);
41
42  LifeTimeExtension bTemporary{4};  O Ok
43  LifeTimeExtension pTemporary(4);  P Dangling reference
```

Listing 8.22

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, **structs** or **classes**, as well as **unions**, with user-declared constructors no longer qualify as aggregates. We are looking at a breaking change here. **Classes**, **structs**, or **unions** with user-declared constructors were classified as aggregates in C++17. Consider the following code and remember we are talking about C++17, your existing code-base:

```
1  struct NotDefaultConstructible {
2      int x;
3
4      A Prevent default construction
5      NotDefaultConstructible() = delete;
6  };
```

Listing 8.23

The **struct** **NotDefaultConstructible** yells with its name and the **delete** default constructor in **A** 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 of initializing the object properly later. However, then some value gets forgotten, and you spend a day or more chasing a very weird 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 briefly overstate it here. More important than the name of the type is the deleted default constructor in **A**. 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 **C** does?

```

1 // NotDefaultConstructible ndc1; B This statement does not compile
   as intended
2
3 C What do you think does this statement does?
4 NotDefaultConstructible ndc2{};

```

Hands up, who thinks that ndc2 does rightfully not compile? We did all we could to make it fail at compile-time, so what could possibly go wrong? Well, according to C++17 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?

```

1 struct NotDefaultConstructible {
2     int x;
3
4     private:
5         A Prevent default construction
6         NotDefaultConstructible() = delete;
7     };
8
9 // NotDefaultConstructible ndc1; B This statement does not compile
   as intended
10
11 C What do you think does this statement does?
12 NotDefaultConstructible ndc2{};

```

Well, do you really think that it doesn't compile, or is that more of a wish? It is the latter. Even with a deleted `private` constructor, **C** 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 forces said that everybody should use braced initialization, 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 bizarre case, which is not backward compatible. If you accidentally 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

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 vector type ourselves. The compiler deduces the type from the arguments provided to initialize the `std::vector`:

```
1 A Using std::vector without specifying a type
2 const std::vector a{2, 3, 4};
```

Listing 8.2-6

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 not existing or the process not having 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 would like to know the exact reason. If so, we must 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 want 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 consisting of

an entry for `errno` and one holding the actual return value. Let's call this aggregate `ReturnCode`.

```

1  struct ReturnCode {
2      int returnCode;
3      int err;
4  };
5
6  auto Open(std::string_view fileName)
7  {
8      return ReturnCode{open(fileName.data()), errno};
9 }
```

Listing 8.27

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 we look further, for example, `lseek`, we find another return-type `off_t`. Of course, we can create a new aggregate for each function, 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:

```

1  template<typename T>
2  struct ReturnCode {
3      T    returnCode;  A Make the return code's type generic
4      int err;
5  };
6
7  auto Open(std::string_view fileName)
8  {
9      B We need to specify the type of the template parameter
10     return ReturnCode<int>{open(fileName.data()), errno};
11 }
```

Listing 8.28

While the change itself is easy, you can see an additional change in Listing 8.28 on page 242. We now need 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 provide
struct ReturnCode {
    ^
<source>:10:8: note: candidate function template not viable:
    requires 0 arguments, but 2 were provide
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 provide
struct ReturnCode {
    ^
<source>:10:8: note: candidate function template not viable:
    requires 0 arguments, but 2 were provide
1 error generated.
Execution build compiler returned: 1
```

Output

Specifying the type explicitly is for sure one approach, but if we think in generic code where that type may change with the platform's open equivalent, having to name the type in the wrapper is not what I want. We are looking for a simple solution, so no decltypeing. This is a situation for CTAD. We can write our own so-called *deduction guide*, which tells the compiler how to deduce the type. The error message shows that the compiler already talked about a deduction guide. This helps the compiler do precisely the same as it does when it comes to function templates. There, the compiler can deduce the function template parameters on its own. The compiler can generate a so-called implicit deduction guide for straightforward

cases and classes. 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:

```

1  template<typename T>
2  struct ReturnCode {
3      T    returnCode;
4      int err;
5  };
6
7  A Deduction guide
8  template<typename T>
9  ReturnCode(T, int) -> ReturnCode<T>;
10
11 auto Open(std::string_view fileName)
12 {
13     B Works without specifying a type
14     return ReturnCode{open(fileName.data()), errno};
15 }
```

Listing 8.30

We can now omit the type, as you can see in **A**. 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

```
1  TypeName(Parameters) -> TypeName<Parameters>
```

I omitted the template-head 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, given 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 differs from classes. Let's try out an exercise. Instead of using an aggregate, we use a class with

private members and provide a constructor. For the sake of completeness, we then also provide the now necessary access functions:

```
1 template<typename T>
2 struct ReturnCode {
3     ReturnCode(T t, int e)
4         : returnCode{t}
5         , err{e}
6     {}
7
8     auto GetReturnCode() const { return returnCode; }
9     int GetErrno() const { return err; }
10
11 private:
12     T    returnCode;
13     int err;
14 };
15
16 auto Open(std::string_view fileName)
17 {
18     B Works without specifying a type
19     return ReturnCode{open(fileName.data()), errno};
20 }
```

Listing 8.71

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. Providing an explicit deduction guide seems to be a much better choice than the class version. 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 and the fewer errors we carry in our code. This goes a long way if you consider future maintenance, bug fixing, and so on. Having the compiler generate these for us is a win, which is exactly what we can now get in C++20. Here is the

C++20 version, back with an aggregate.

```
1 template<typename T>
2 struct ReturnCode {
3     T    returnCode;
4     int err;
5 };
6
7 auto Open(std::string_view fileName)
8 {
9     return ReturnCode{open(fileName.data()), errno};
10 }
```

Listing 8.32

It looks exactly like the initial version we saw 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 default member initializers as defaults.
- Designated initializers from a braced initialization, which always gives us default or zero initialization for all unnamed members without default 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 emulate 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 compiler generates implicit deduction guides similarly for both classes and aggregates.

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 template 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 on page 17, we used them at the start in the Add example. We later used them again in §5.6 on page 168 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 §8.4 on page 241 that this is possible for classes since C++17 as well, and C++20 added support for aggregates.

NTTPs is 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`.

```

1  template<typename T,   A A type template parameter
2          size_t N>     B A non-type template parameter
3  struct Array {
4      T data[N];
5  };
6
7  Array<int,    C Passing int as type parameter
8      5>      D Passing 5 as non-type parameter
9  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. As we can see, a value, 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. Pointers to objects or pointers to functions 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. The object must have a `static` storage duration for the first case. `std::nullptr_t`

was allowed as well. Prior to C++20, the missing forms were floating-point numbers and classes.

The reason for the limitations is that the compiler needs to compare two template instantiations and determine 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 determine 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, then 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:

```
1 template<double> x>
2 struct A; A A type which uses double as NTPP
3
```

Listing 9.2

```

4  B Different bit pattern results in different type
5  static_assert(not std::is_same_v<A<+0.6 - 0.2>, A<0.4>>);
6
7  C Different bit pattern results in different type
8  static_assert(not std::is_same_v<A<+0.0>, A<-0.0>>);
9
10 static_assert(+0.0 == -0.0);   D IEEE 754 says they are equal
11
12 E Same bit pattern
13 static_assert(std::is_same_v<A<0.1 + 0.1>, A<0.2>>);

```

At the top in **A**, we declare a template type we use for our comparisons. In **B**, we see the same example 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 applies to 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 $-0.0 == +0.0$, as we can see in **D**. Lastly, in **E**, we see that if two calculations yield the same value, they are considered identical with that bit pattern.

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 doesn'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:

Listing 9.3

```

1 struct fixed_string {
2     char data[5]{};
3
4     constexpr fixed_string(const char* str)
5     {
6         std::copy_n(str, 5, data);
7     }
8 };

```

The type, `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 us to check whether the provided 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:

```

1 template<typename CharT, std::size_t N>
2 struct fixed_string {
3     CharT data[N]{};
4
5     constexpr fixed_string(const CharT (&str)[N])
6     {
7         std::copy_n(str, N, data);
8     }
9 };

```

Listing 9.4

That looks much better. Thanks to the compiler, our `fixed_string` now deduces the type and size of the array we pass in the constructor. A potential initialization of `fixed_string` can be this:

```

1 fixed_string fs{"Hello, C++20"};

```

Listing 9.5

We learned that we do not need more code in §8.4 on page 241 thanks to CTAD. The compiler determines how to instantiate a `fixed_string` object.

Now, we have our baseline. 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:

```

1  template<fixed_string Str> A Here we have a NTTP
2  struct FixStringContainer {
3      void print()
4      {
5          std::cout << Str.data << '\n'; B Use Str
6      }
7  };
8
9  void Use()
10 {
11     C We can instantiate the template with a string
12     FixStringContainer<"Hello, C++"> fc{};
13     fc.print(); D For those who believe it only if they see it
14 }
```

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, 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 briefly about compile-time data.

9.3.2 What compile-time data do we have

I often get the question of 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, set up an entire object at compile-time. The question is not about these details, people ask where to apply `constexpr`. Some say that we never need to run that program if everything is `constexpr` and the compiler evaluates that at compile-time. 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 being 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 on page 145 are examples of data 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, I wanted a type-safe format or print function that checks at compile-time that the number of specifiers provided matches 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 must ensure that our test cases cover every use of a print function. As great as `std::format` is, the current design checks 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`:

```

1 template<fixed_string Str> A Takes the fixed string as NTTP
2 struct FormatString {
3   B Store the string for easy access
4   static constexpr auto fmt = Str;
5
6   C Use ranges to count all the percent signs.
7   static constexpr auto numArgs =
8     std::ranges::count(fmt.data, '%');
9
10  D For usability provide a conversion operator
11  operator const auto *() const { return fmt.data; }
12 };

```

What do we have in these few lines of code? First, we see again `fixed_string` as NTTP, in **A**. The class template `FormatString` then stores an instance of `fixed_string` in the form of a `static constexpr` variable as a static member, as shown in **B**. 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 **C**, `numArgs` counts 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 **D**. 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 the 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 lives easier, we use the abbreviated function syntax from Concepts and declare the first parameter with `auto`. We have seen those in §1.9 on page 34. 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:

Listing 9.8

```
1 void print(auto fmt, const auto&... args)
2 {
3     printf(fmt, args...);
4 }
```

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:

Listing 9.9

```
1 print(FormatString<"%s, %s">{}, "Hello", "C++20");
```

Aside from the 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 creation 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 be 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 don't like seeing and typing 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 in the call of a special operator `operator" _myLiteral`, which converts or constructs an object. They work only with constants like numbers or characters.

Listing 9.10

```

1 template<fixed_string Str>
2 constexpr auto operator""_fs()
3 {
4     return FormatString<Str>{};
5 }
```

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.

```
1 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`, an excellent 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`:

```

1 void print(auto fmt, const auto&... args)
2 {
3     A Use the count of arguments and compare it to the size of the pack
4     static_assert(fmt.numArgs == sizeof...(args));
5
6     printf(fmt, args...);
7 }
```

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 in Listing 9.12 on page 256 compiles. The code compiles because `fmt` only looks like a function parameter, and ok, it is one too, but in the `static_assert`, we do not use the parameter's 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-trait 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. Still, it also contains static data members usable in a `static_assert`. This technique is not new. As I pointed out, it is the same as with type-trait. 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 outstanding, 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. Our `print` function should also have this check.

We take the easy road here 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.

```
1  template<typename T, typename U>
2  constexpr bool plain_same_v = A Helper type-trait to strip the type
3      down
4      std::is_same_v<std::remove_cvref_t<T>,
5                      std::remove_cvref_t<U>>;
6
7  template<typename T>
8  constexpr static bool match(const char c)
9  {
10     switch(c) { B Our specifier to type mapping table
```

Listing 9.13

```

10     case 'd': return plain_same_v<int, T>;
11     case 'c': return plain_same_v<char, T>;
12     case 'f': return plain_same_v<double, T>;
13     case 's': C Character strings are a bit more difficult
14         return (plain_same_v<char,
15                 std::remove_all_extents_t<T>) and
16                 std::is_array_v<T>) or
17                 (plain_same_v<char*>,
18                  std::remove_all_extents_t<T>) and
19                  std::is_pointer_v<T>);
20     default: return false;
21 }
22 }
```

In Listing 9.13 on page 257 there 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 **A**, 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 are 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 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.

Listing 9.14

```

1 template<int I, typename CharT>
2 constexpr auto get(const std::span<const CharT>& str)
3 {
4     auto start = std::begin(str);
5     const auto end = std::end(str);
6
7     for(int i = 0; i <= I; ++i) { A Do it I-times
8         B Find the next percent sign
9         start = std::ranges::find(start, end, '%');
```

```

10     ++start;  C Without this we see the same percent sign
11 }
12
13 return *start;  D Return the format specifier character
14 }
```

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 §7.4.1 on page 210, 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:

```

1 template<typename CharT, typename... Ts>
2 constexpr bool IsMatching(std::span<const CharT> str)
3 {
4     return [&]<size_t... I>(std::index_sequence<I...>)
5     {
6         return ((match<Ts>(get<I>(str))) && ...);
7     }
8     (std::make_index_sequence<sizeof...(Ts)>{});
9 }
```

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. The check effectively ends as soon as the first element, going from the outermost to the innermost, isn't a match. Figure 9.1 on page 260 illustrates the two approaches.

Excellent, we have all the building blocks we need. All that is left for our specifier-matches-type-check is to use the parts in `print`:

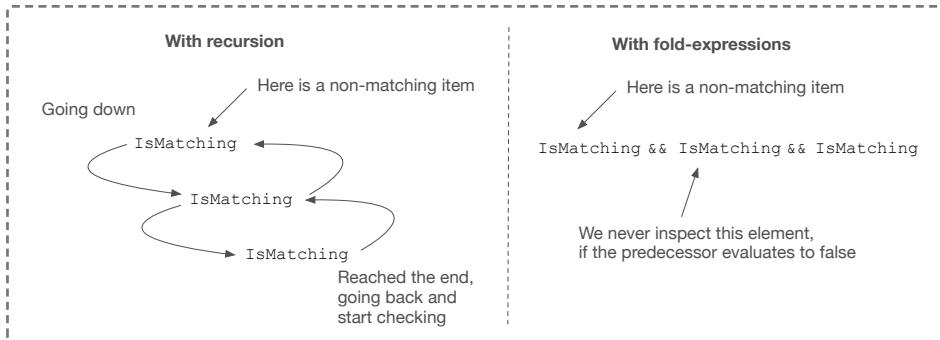


Figure 9.1: Recursion vs. fold expression to find the first non-matching item.

```

1  template<typename... Args>
2  void print(auto fmt, const Args&... ts)
3  {
4      ❶ Use the count of arguments and compare it to the size of the pack
5      static_assert(fmt.numArgs == sizeof...(Args));
6
7      // cannot pass ts... because that is run-time
8      static_assert(
9          IsMatching<std::decay_t<decltype(fmt.fmt.data[0])>,
10         Args...>(fmt.fmt.data));
11
12     printf(fmt, ts...);
13 }
```

Listing 9.16

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 ❶. For example, we can spare all that if we settle for `char` as type. However, I chose to use the generic approach. After we figure that type out, we supply the expanded pack of format arguments in ❷ 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 probably result in a 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.

```
1 void print(char* s, const auto&... ts)
2 {
3     printf(s, ts...);
4 }
```

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.

```

1  A Helper, returns always false
2  template<typename...>
3  constexpr bool always_false_v = false;
4
5  template<typename... Ts>
6  void print(const char* s, const Ts&...)
7  {
8      B Use the helper to trigger the assert whenever this template is
9          instantiated
10     static_assert(always_false_v<Ts...>, "Please use _fs");
11 }
```

In A, 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 checks at compile-time whether a format specifier matches the provided type and that the number of format specifiers matches the 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

In this chapter, we look at how C++20 brings us various smaller library improvements. We will see reliable ways to convert a type bitwise into another without changing the underlying bit representation; a modern way to retrieve source location information instead of the old way most of us are used to from `__FUNCTION__` and `__LINE__`; is a nice improvement to containers when we want to know whether a key exists; and additions to `std::string` finally allow us to check whether a string begins or ends with another string.

10.1 `bit_cast`: Reinterpreting your objects

When we work with low-level code, we come across a need to cast a type bitwise to another type without changing the underlying bit representation. One example is the serialization of network data, where the data comes in as a byte-stream, and we need to convert the data back to an object or vice versa. We sometimes also need to cast from one type to another. One example is showing the bit representation of a `float`. There, we cast a `float` to an `int` to show the underlying representation. The following listing shows the options we had before C++20:

```
1 float pi = 3.14f;  
2
```

Listing 10.1

Listing 10.1

```

3  uint32_t a = static_cast<uint32_t>(pi);  A Does not do what we want
4  //uint32_t b = reinterpret_cast<uint32_t>(pi); B Does not compile
5  C Uses type-punning, can result in UB
6  uint32_t c = *reinterpret_cast<uint32_t*>(&pi);

7
8  union FloatOrInt {
9      float    f;
10     uint32_t i;
11 };
12
13 FloatOrInt u{pi};
14
15 uint32_t d = u.i;  D A lot of code just for a simple cast
16
17 uint32_t f{};
18 memcpy(&f, &pi, sizeof(f));  E Copying the value

```

They all look good at first glance. We start with the safest cast in our toolbox, `static_cast` in **A**. What looks good doesn't do what we want here. The `static_cast` does a proper conversion from `float` to `int`, which in this case means that `a` contains the value 3 after the cast and not the floating-point representation of `pi`.

The next logical choice is **B**, where a `reinterpret_cast` is used to cast one value to another. However, this code doesn't even compile because using `reinterpret_cast` here is invalid.

This brings us to attempt **C**, where we use `reinterpret_cast` on the pointer to `pi` and later dereference the result. The bad thing is that this approach seems to work. However, depending on the compiler and the compiler's optimization level, the result might not be what we desire. With this cast, we look at type-punning, where we are aliasing a pointer of one type through a pointer of another type. The compiler could optimize this away, and we would end up with an unexpected result. Compilers do warn about this if the warning level is sufficient enough.

Yes, getting this task done reliably is hard in C++. So let's look at approach **D**, which requires the most code. We use a `union` that contains a `float` and `int` member. We then set the `float` member with `pi` and read back the `int` member. A lot of code for such a simple task, and we are still in UB-land! What works in C and depending

on the optimization level is not allowed in C++. In C++, we are only allowed to read the *active* member of a union. So, in our case, we can only read back *f*, but not *i*.

This leaves us with something that looks like a non-option, using `memcpy` and copying the `float` into the `int`, as shown in E. This actually works reliably and is well-defined behavior. It's just that the code doesn't look all that appealing. A search on the internet reveals that others have had that situation as well and have written a function template wrapper around that `memcpy` called `bit_cast`. Of course, `bit_cast` then does a couple of additional checks, such as to ensure that the source and destination are the same size.

In C++20, there is no need to implement `bit_cast` ourselves, as the standard now comes with that type in header `<bit>`:

```
1 const float pi = 3.14f;
2 const uint32_t pii = std::bit_cast<uint32_t>(pi);
```

Listing 10.2

Aside from the fact that we do not need to implement and maintain `bit_cast`, this new type ensures that various checks are performed to ensure that the conversion is valid. The two types not only need to have the same size, but they also need to be trivially copyable. What remains the same is that `bit_cast` doesn't do any kind of rounding. In the example, the .14 is cut off, and only the 3 ends up in `pii`.

For all those who have a lot of `constexpr` code, the good news is that the STL `bit_cast` is `constexpr` should the two types and their subobjects have:

- no union
- no a pointer or member-pointer
- no `volatile`
- no non-static data members that are references

C++20 enables us, for the first time with `bit_cast`, to have a standardized way of casting the bit-values of one type to another if the types satisfy the requirements above. As a bonus, `bit_cast` is also `constexpr`.

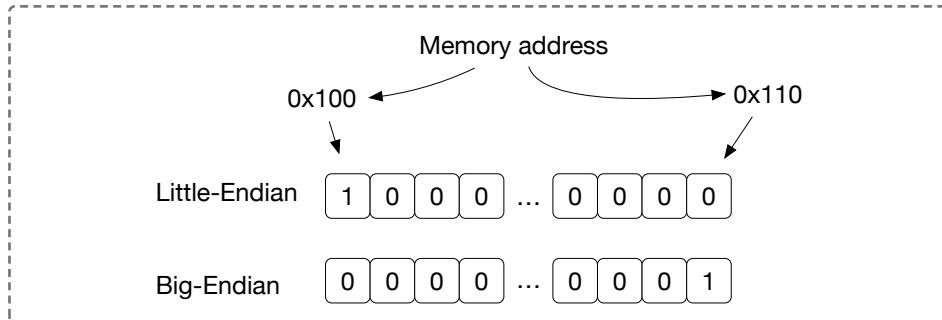


Figure 10.1: Visualization of the layout of the number 1 in little- and big-endian format.

10.2 endian: Endianness detection at compile time

When you work with a network code that transfers data or file formats that need to be saved to disk, you probably have already had to adjust your data's bytes. For example, network traffic is defined to be in network byte order, which is defined as big-endian. However, today, most computers use little-endian as byte order. For example, although Motorola uses big-end chips, Intel and ARM chips are little-endian. To be able to transfer data from a little- to a big-endian host, the little-endian host must swap the data to a big-endian byte order. On Linux, we have the C functions `hton` and `htonl` to swap the byte order of, respectively, a `uint16_t` or `uint32_t` value. It has always bordered me that I had to remember and apply the correct function according to the corresponding type. For example, I can easily use `hton` to swap a `uint32_t`, swapping only half of the `uint32_t`'s bytes. This is especially painful during refactorings or protocol changes where, for us, changing only the type of the data is not sufficient, but also for us to carefully look at all the byte swaps.

hton

The name stands for *host to network*. These functions also exist the other way around and are typically used on Linux or other POSIX-like systems.

For that reason, we often write a C++ wrapper function around these two C functions and let them overload on the type. But there is another shortcoming. The C functions are not `constexpr`. Most data that needs to be swapped is indeed dynamic and not known at compile-time, but some data is known at compile-time. For

example, protocol revisions or similar static things. The ability to swap such data at compile time would be nice.

Writing our own `hton` function is not that hard. The cumbersome part when writing `hton` all the time was finding out whether we needed to swap that on a specific machine. What `hton` does is to swap on little-endian hosts but do nothing on a big-endian host. We needed to use some pre-defined compiler constants to get the information at compile time. Of course, they could differ slightly between compilers.

C++20 adds a class enum `std::endian` to the `<bit>` header. The great thing is that due to the nature of a class enum, `std::endian` is a compile-time value. This now allows us to implement, for example, a `ByteSwap` function, using `if constexpr`, and make the entire function `constexpr`, if we implement the swapping logic by hand:

```
1 constexpr auto ByteSwap(std::integral auto value)
2 {
3     if constexpr((std::endian::native == std::endian::big) ||
4                  (sizeof(value) == 1)) { B chars don't need swapping
5         return value;
6     } else {
7         return ReverseBytes(value);
8     }
9 }
```

Listing 10.3

The implementation assumes that there is a `ReverseBytes` function that actually does the actual swapping. This allows, for example, to provide four different overloads for 8, 16, 32, and 64 bits. We use the `if constexpr` together with `std::endian` to call `ReverseBytes` only if we are on a little-endian machine and the size of the type is greater than one. Otherwise, `ByteSwap` returns the value that was passed to it. `ByteSwap` uses the abbreviated function template syntax we already discussed in §1.9 on page 34. With concept `std::integral`, we ensure that only integral types are swapped. I prefer it this way because we are then able to write generic code around `ByteSwap`. We can pass a `uint8_t` to `ByteSwap` as well, and the implementation ensures that such a type is never swapped.

`std::endian` has three enumerations:

- `little`

- **big**
- **native**

While `std::endian` is a more minor addition, this new type removes the need for macros and, by that, makes our code more robust and cross-platform compatible.

10.3 `to_array`

We already saw `std::array` a couple of times in this book, and you may know `std::array` from your daily work even better. This data type provides us with an efficient way of having an array that does not decay to a pointer when passing such an object to a function. `std::array` further removes the need for `sizeof`, which is error-prone. Prior to C++17, `std::array` had another difference, which can be seen as a con. We always needed to specify the size of the `std::array`:

```
1 const char cArray[]{"Hello, C++20"};   A Compiler deduces the size
2 B We need to specify the size
3 const std::array<char, 13> array{"Hello, C++20"};
```

Listing 10.4

We can see this both ways. Some of you might say that being explicit is the right thing. Others are happy with the implicit way. In this case, I think the implicit way is acceptable. Why? Because I need that exact string no matter how many characters the string has. Having to write out the size explicitly made me count the characters before, and guess what? I forgot the string termination character on my first attempt. In the case presented here, it is not helpful for me to have that size value, the same way as I do not want to specify the size of a `std::string` or `std::vector`.

CTAD from C++17 makes that difference go away. Thanks to CTAD, we are able to write the same code in C++17 this way:

```
1 const char cArray[]{"Hello, C++20"};   A Compiler deduces the size
2 B Compiler deduces the size and type
3 const std::array array{"Hello, C++20"};
```

Listing 10.5

We can omit the type and size and let the compiler deduce both values from the initializer. That can now be seen as superior to a plain C-style array. However, with

CTAD, we can now run into an issue: `array` in **B** does not do what we want the code to do, at least in the typical case. The element type of the resulting `std::array` is `const char*`, and the size is one, which is clearly not like the C-style array version. This is an error that is likely to happen for a string. For other types like `int`, we pass the array elements comma-separated and can use CTAD properly. This issue mainly comes from the fact that, at this point, we cannot express what we want. For example, we could also want an array of pointers to some strings.

Even with CTAD, there is still no way to provide only the type and let the compiler deduce only the size. This is sometimes required if we want to initialize a `std::array` with a smaller type than the `std::array`'s element type.

With C++20, we have a new function `std::to_array`, which creates a `std::array` for us. The difference is that in this case, `std::to_array` deduces a C-style array for an array, and that we can specify the type and let the compiler figure out the size:

```
1  A Compiler deduces the size and type
2  const auto array = std::to_array("Hello, C++20");
3
4  B Compiler deduces the size and specify the type
5  const auto array2 = std::to_array<const char>("Hello, C++20");
6
7  C Create the array inline
8  const auto arrayFromList = std::to_array({3, 4, 5});
9
10 int intArray[]{3, 4, 5};
11 D Move the values
12 const auto movedArray = std::to_array(std::move(intArray));
```

Listing 10.6

As you can see, we can use `std::to_array` in auto-mode, letting it deduce the type and the size, as shown in **A**, or specify only the type, as shown in **B**, while the compiler handles the rest. We can also create the array inline, as shown in **C**. For move-only types, if we want to move the values of an existing array into a `std::array`, we can also use `std::to_array`.

Especially when dealing with C-style arrays, remembering `std::to_array` is worth it the next time you encounter such a code fragment.

10.4 span: A view of continuous memory

For this section, we suppose that we are writing a part software that sends and reads data. For this example, it doesn't matter whether the software is a network protocol with which we communicate. Common functions on a Linux host are `read` and `write` from C. They both come with a value length API, which often looks like this:

```

1  bool Send(const char* data, size_t size);
2  void Read(char* data, size_t size);

3
4  void Use()
5  {
6      char buffer[1'024]{};

7
8      Read(buffer, sizeof(buffer));
9      Send(buffer, sizeof(buffer));

10     // some code in-between

11
12     char buffer2[2'048]{};

13     Read(buffer, sizeof(buffer2));
14     Send(buffer, sizeof(buffer2));
15 }

```

Listing 10.7

At the top, we see the two functions `Send` and `Read`. As I said, their actual implementation doesn't matter for this example. Below them, we see how they are used, with two different arrays of type `char`. Imagine there is some code between the two example calls to `Read` and `Send`. Pause a moment and ask yourself the question of whether this code contains a bug.

The answer is yes. The code does contain a bug. In the second `Read/Send` pair, `buffer` is passed as the first argument and `sizeof(buffer2)` as the length. Size of buffer two! But `buffer2`'s size is twice that of `buffer`. Yes, better names could avoid this bug, but we have no guarantee for that. I personally dislike the call to `sizeof` very much. We have more to read, write, and the code asks for inconsistencies. We can also say that such code is not C++, so let's make this example more C++ish by using `std::array`:

```
1 template<size_t SIZE>
2 bool Send(const std::array<char, SIZE>& data);
3
4 template<size_t SIZE>
5 void Read(std::array<char, SIZE>& data);
6
7 void Use()
8 {
9     std::array<char, 1'024> buffer{};
10
11    Read(buffer);
12    Send(buffer);
13
14    // some code in-between
15
16    std::array<char, 2'048> buffer2{};
17
18    Read(buffer2);
19    Send(buffer2);
20 }
```

Listing 10.8

As you can see, both `Send` and `Read` now take a `std::array` and are templates. Each needs to be a template size of the `std::array` to be deduced, as we use two different-sized arrays. Our using side becomes cleaner and more robust. The calls to `sizeof` are completely gone. All we can mess up now is to pass the wrong array, but that would not lead to a buffer overflow as it did before. This code is a good approach in C++17 and before. The only thing that can be an issue is that `Send` and `Read` are now templates. There is a non-zero chance that the instantiations for each different-sized buffer created an overhead in the binary size of this code, which we do not want. This starts to become a problem if the functions are complex or longer. Optimizers are great, but they have their limits. Another disadvantage of a large code base is compile time. The instantiation of the two functions for each size does cost some time. Whether that is acceptable depends on various factors of your project. Not having to worry or think about these disadvantages would be much better.

This is where a new type in C++20 comes into play: `std::span`. You find `std::span` in a new header with the same name. Let's first look at how the code looks when we apply `std::span`:

Listing 10.9

```

1  bool Send(std::span<const char> data);
2  void Read(std::span<char> data);
3
4  void Use()
5  {
6      std::array<char, 1'024> buffer{};
7
8      Read(buffer);
9      Send(buffer);
10
11     // some code in-between
12
13     char buffer2[2'048]{};
14
15     Read(buffer2);
16     Send(buffer2);
17 }
```

We can see that in this version, we substituted `std::span` into the function signature of `Read` and `Send`. The template-head is also gone. In `Use`, we can see that a `span` is constructible from a `std::array` as well as from a `char` array.

A `std::span` provides a member `data` to access its data and `size` to obtain the size of the data. We can ask an object of type `span` whether the object is `empty`, access the first (`front`) or last (`back`) element, and can access elements via the subscript-operator. `std::span` very much looks and feels like any other container of the STL, yet `std::span` is different. A `std::span` is, like a `std::string_view` (Std-Box 3.1 on page 117), only a view on the data. And this has nothing in common with `std::ranges::view`. A `std::span` stores only a pointer to the data and the length of the data. That implies that the data the `std::span` refers to must live longer than the `span` itself. Otherwise, we have UB because the memory the `span` points to is no longer valid. It is very important to notice that `std::span` is a non-owning container!

Below we see how a `std::span` looks internally:

```

1  template<typename T>
2  class Span {
```

Listing 10.10

```
3     T*      mData;
4     size_t  mSize;
5
6 public:
7     // constructors
8 }
```

As already said, `std::span` stores only a pointer to the data. The trick over `std::array` is that the size is determined in the constructors at run-time. I left the constructors out here because there are many of them. But the important basic is that `std::span` stores only a pointer to memory provided from elsewhere. As long as you keep that in mind and respect it, `std::span` is an excellent type for the kind of API we looked at with `Send` and `Read`.

10.5 `source_location`: The modern way for `_FUNCTION_`

Often, when we are writing some kind of debug or log function, there is a desire to also have information about the filename, line number, and function name from which the debug or log function was called. To simplify the example here, I will use a log function as a synonym for both debug and log function. We've already seen an implementation in §5.6 on page 168, but this implementation was without additional source information. Probably the most famous example of such a function is `assert`. Whenever the condition is not met, `assert` announces this violation and includes the function name and line number where the error occurred. Those of you who already have had to implement such a log function know that getting the additional information, line number, etc., requires a trick.

10.5.1 Writing a custom assert function

Suppose we are about to write a custom `Assert` function. This `assert` should show the function name and line number where the `assert` was triggered, as well as a message to leave a clue as to what went wrong. `Assert` also has a condition such that the `assert` message is only sent if that condition is not true. Below, you see a possible implementation.

Listing 10.11

```

1  A Assert function taking function, line, condition, and message
2  void Assert(bool condition,
3           std::string_view msg,
4           std::string_view function,
5           int line)
6  {
7      if(not condition) {
8          std::clog << function << ':'
9              << line B followed by function and line
10             << ":" << msg C and the message
11             << '\n';
12     }
13 }
```

This implementation does the job. From a usability aspect, the code is not that great. Users must repeat static information for each use to get the function name and line number. We can obtain them via the pre-defined compiler macros `__FUNCTION__` and `__LINE__`. This information needs to be repeated for each call to assert. The implementation also begs the risk of supplying false information by accident. For example, the hard-coded current line number could be passed to `Assert`. After a refactoring, this would lead to the wrong place. We don't see these two parameters here when we look at the `assert`, the STL, and the C-standard provide. The reason is that `assert` is a macro, which adds this information for us. We can rebuild this functionality for our own `Assert`:

Listing 10.12

```

1  D Macro wrapper to call Assert
2  #define ASSERT(condition, msg)
3      Assert(condition,
4              msg,
5              __FUNCTION__,
6              __LINE__) E Get function and line information from caller
```

The reason why a macro is required here is that a macro is the only way we can use `__FUNCTION__` and `__LINE__` to get the values of the caller. The preprocessor replaces the macro before the compiler sees the source code. After this preprocessing step, the `ASSERT` macro is replaced with a call to `Assert` as we would write it. Quite

a helpful technique. However, this technique comes with a few drawbacks. For instance, users have to know about the `ASSERT` macro. They can still inadvertently use the `Assert` function instead. Macros are not type-safe, so various errors that are hard to detect are possible. The `assert` macro from the STL takes advantage of the fact that a macro is preprocessed. The STL `assert` can be disabled at compile-time if the define `NDEBUG` is not set. That way, the entire expression disappears. This is great if you like to get your assert statements removed in release mode. However, eliminating code after tests have run is not always acceptable. For example, the removal changes the timing of the software. In some projects, asserts stay there forever to ensure that the timing remains the same and that potential bugs result in consistent behavior. In such a case, the benefit of a macro, to vanish, becomes zero. What we would like there is just the `Assert` function but with the automatic insertion of the function name and line number.

Using default arguments comes to mind to solve the automatic insertion of the parameters function name and line number. There are two obstacles to this approach. First, `__FUNCTION__` is only valid in a function context, not as a default parameter. Second, putting `__LINE__` as a default argument in a function signature will use the function's definition line number, not the one from which the function is called. So, this approach is a dead-end. This is the reason why we had to use the macro approach all these years.

10.5.2 Writing a custom assert function with C++20

Now we have C++20, which brings us a new include `<source_location>`, which defines a type with the same name.

```
1  A Assert function taking condition, message, and source location
2  void Assert(bool                  condition,
3              std::string_view    msg,
4              std::source_location location =
5                  B current() is special
6                  std::source_location::current())
7  {
8      if(not condition) {
9          std::clog << location.function_name() << ':'
10             << location.line() << ":" << msg << '\n';
11 }
```

Listing 10.13

```

11     }
12 }
13
14 void Use()
15 {
16     C A call to Assert with information of Use
17     Assert(1 != 2, "Not met");
18 }
```

What we can see at first glance is that instead of passing `__FUNCTION__` and `__LINE__` with two parameters in C++20, we can use `source_location`, which provides information for both. We can either default construct a then empty `source_location` object, or we can use the construction method `current`. This method is the first of its kind, as it is very special in the world of C++. The `std::source_location::current` does what we previously needed a macro for. This function obtains the information from the caller! Say that we call `Assert` from `Use`, as shown in Listing 10.13 on page 277. The information that `location` carries is the function name `Use` and the line number of the call to `Assert` in `Use`. This new element enables us to build the `Assert` function without a macro. The additional plus is that such an `Assert` is type-safe, containing the information in a single object. Besides the function name and line number, `source_location` also provides the file name. Table 10.1 on page 279 shows all members and their relation to the macros.

10.5.3 Writing a custom log function with C++20

Whenever we need file and line number information, `std::source_location` is a great tool, as we already saw in the previous section. How can we use `std::source_location` in a custom log function? The goal is the same as before. We want to write this log function without the need for a macro. A pre-C++20 attempt could look like the following:

```

1 A Log function taking function, line, and variadic arguments
2 void Log(LogLevel      level,
3           std::string_view functionName,
4           int          line,
```

Table 10.1: Member functions of std::source_location

Method	Functionality	Macro
current()	A static member that creates a new <i>source_location</i> object with the information from where it is called.	N/A
column() ^a	The column number where the object was created.	N/A
file_name()	The file name where the object was created.	__FILE_NAME__
function_name()	The function name where the object was created.	__FUNCTION__
line()	The line number where the object was created.	__LINE__

^a This is an implementation-defined value and can be zero (0) regardless of the real column number.

```

5         std::string_view fmt,
6         const auto&... args)
7     {
8         std::clog << std::format("{}:{}:{}: ",
9                               static_cast<unsigned int>(level),
10                              functionName,
11                              line)
12         << std::vformat(fmt, std::make_format_args(args...))
13         << '\n';
14     }
15
16 B Macro wrapper to call Log
17 #define LOG(level, fmt, ...) \
18     Log(level, __FUNCTION__, __LINE__, fmt, __VA_ARGS__)

```

Listing 10.14

We can see that `Log` takes a log-level and the source information in the form of function name and line number. These parameters are followed by a format string and the format arguments. `std::format` is used to format the output and `std::clog` to manage the output. We are still in the same situation as before with `Assert`. A macro is needed to obtain the source information, or users have to fill in this infor-

mation manually. However, this time, we are looking at a slightly different pattern. Have a look at a possible implementation:

```

1  A Log function taking function, line, and variadic arguments
2  void Log(LogLevel           level,
3          std::source_location location,
4          std::string_view      fmt,
5          const auto&... args)
6  {
7      std::clog << std::format("{}:{}:{}: ",
8                               static_cast<unsigned int>(level),
9                               location.function_name(),
10                              location.line())
11     << std::vformat(fmt, std::make_format_args(args...))
12     << '\n';
13 }
14
15 B Macro wrapper to call Log
16 #define LOG(level, fmt, ...) \
17     Log(level, std::source_location::current(), fmt, __VA_ARGS__)

```

Listing 10.15

The difference with `Assert` is that `Log` expects a variadic number of arguments. Because of that, we cannot place `std::source_location` as the last parameter and default this parameter. As you can see, there is still a `LOG` macro forwarding the `std::source_location` to the `Log` function. This approach is likely still a little bit better than in C++17 but with the macro in place, not what is desired here.

Let's think about the situation a bit. We know that `std::source_location::current()` retrieves the source information from where the function is called. We solved this in the `Assert` with a default parameter, which takes place at the point where `Assert` is called. For `Log`, we need something similar, but the variadic arguments block our former solution. We can use or misuse another feature of C++ to achieve what we want. Implicit conversion allows us to create temporary objects in place, and we can have default arguments in constructors. Let me show you the code for this before going into detail:

Listing 10.16

```
1 struct Format {
2     std::string_view      fmt;
3     std::source_location loc;
4
5     A Constructor allowing implicit conversion
6     Format(
7         const char*           _fmt,
8         std::source_location _loc = std::source_location::current())
9     : fmt{_fmt}
10    , loc{_loc}
11    {}
12 };
13
14 B Log now takes Format which implicitly adds source location information
15 void Log(LogLevel level, Format fmt, const auto&... args)
16 {
17     std::clog << std::format("{}:{:{}:} ",
18                               static_cast<unsigned int>(level),
19                               fmt.loc.function_name(),
20                               fmt.loc.line())
21     << std::vformat(fmt.fmt,
22                      std::make_format_args(args...))
23     << '\n';
24 }
```

There is a new type `Format`, which contains a `std::string_view` and a `std::source_location`. The constructor of `Format` **A** allows implicit conversion and takes `std::source_location` as the second argument, defaulted to `source_location::current`. Whenever the compiler creates a temporary object of `Format` with only a `const char*`, this object also contains the source information from where it was created.

`Log` now takes `Format` as the second argument, and `std::source_location` is gone from the parameter list. The inner of `Log` comes with a slight modification as well. We now need to use `fmt`, which contains the format and source information. Other than that, the code is unchanged, except that the macro is gone.

10.6 contains for all associative containers

The STL comes with various containers. Containers that organize their data as key-value pairs are called associative containers. We can further distinguish between ordered and unordered associative containers. In C++, we have the following associative containers:

- `std::set`
- `std::map`
- `std::multiset`
- `std::multimap`
- `std::unordered_set`
- `std::unordered_multiset`
- `std::unordered_map`
- `std::unordered_multimap`

For this section, I will stick to `std::map` as an example for associative containers. When we use such a container, one task that comes up is to check whether a particular key already exists in the container. I'm not talking about checking whether the operation was successful after an insertion. I'm talking about the case where we only want to know whether a particular key exists. Based on that information, we, for example, prompt the user for a value because the key was not present. In C++17, checking whether a key exists is often done via `find`:

```

1 const std::map<int, std::string> cppStds = {{11, "C++11"},  

2                                         {14, "C++14"},  

3                                         {17, "C++17"},  

4                                         {20, "C++20"}};  

5  

6 const int key = 20;  

7  

8 if(cppStds.find(key) != cppStds.end()) {  

9     std::cout << "Found\n";

```

Listing 10.17

Listing 10.17

```
10 } else {
11     std::cout << "Not found\n";
12 }
```

Sometimes, people use the `count` method to solve the same task. However, this works only well for `std::map` and `std::set`. There, both `find` and `count` have logarithmic complexity in the size of the container. For `std::multimap` and `std::multiset`, this is different. `count` of one of the multi containers has logarithmic complexity plus a linear one in the number of elements found. The good part about `count` is that it returns an integer that is convertible to a boolean. We can use the result of `count` directly in an `if`, without writing the negation we see in Listing 10.17 on page 282 with `find`, where `find` returns an iterator. From a performance aspect, `find` was the more generic approach, with the same performance for all containers. From a teaching perspective, `find` is a nightmare to teach. From a programmer's perspective, the fact that we cannot express our intent in code is a nightmare. C++20 puts us out of our misery. We finally have a `contains` method for all associative containers.

Listing 10.18

```
1 const std::map<int, std::string> cppStds = {{11, "C++11"},  
2                                         {14, "C++14"},  
3                                         {17, "C++17"},  
4                                         {20, "C++20"}};  
5  
6 const int key = 20;  
7  
8 if(cppStds.contains(key)) {  
9     std::cout << "Found\n";  
10 } else {  
11     std::cout << "Not found\n";  
12 }
```

The code in C++20 is short and clean and captures our intent. By using `contains`, we also ensure that we always get the best performance when testing whether a container contains a key.

10.7 starts_with and ends_with for std::string

One frequently used type of the STL is `std::string`. This lovely datatype has plenty of useful methods to modify and search in a string. However, one very often executed task is to check whether a string starts or ends with a certain string. Here is an example where we check whether the string "Hello, C++20" starts with "Hello":

```
1 const std::string s{"Hello, C++20"};
2
3 if(s.find("Hello") != std::string::npos) { puts("Found!"); }
```

Listing 10.19

I vaguely remember the first time I needed to solve this, well, task. I was surprised that there is no `starts_with` function in C++ for `std::string`. Next, I realized, after a while, that `string::npos` exists. There is another suboptimal thing. In the positive case, we have found the string, but the conditions reads not equal (`!=`). We saw this pattern before in §10.6 on page 282. The not equal part is confusing when reading code. I sometimes got feedback that my code was wrong because I accidentally must have switched the positive and negative cases. Feedback I understand entirely. I have to look twice myself. I have written a couple of implementations that wrap that code and are called `starts_with`. A quick search on the internet reveals that I'm not alone. There is a chance that `starts_with` is one of the most implemented functions.

C++20 now finally solves this by adding the so badly missed two member functions `starts_with` and `ends_with` to `std::string`. The former code becomes this:

```
1 const std::string s{"Hello, C++20"};
2
3 if(s.starts_with("Hello")) { puts("Found!"); }
4
5 if(s.ends_with("C++20")) { puts("Found!"); }
```

Listing 10.20

This code is, to me, now much more apparent. We can express our intentions in code. No need to say `find` and compare the result with not equal to another, maybe less frequently used constant. We can now say what we mean without any negation. Sometimes, if not always, the little things matter so much. Sadly, the also missed function `contains` did miss the C++20 release time frame.

Cliff notes

- When you need to convert a type bitwise from one type to another with well-defined behavior, use `std::bit_cast`.
- We can use `std::endian` to detect the endianness at compile-time.
- Use `std::to_array` when you want to create a `std::array` from a C-style string.
- `source_location::current` is a special new function that gives the source location information of the caller.
- Remember that `std::source_location::column` can return zero because the value is implementation-defined.
- For best performance, always prefer `contains` for all associative containers when testing whether the container contains a key.
- Use `starts_with` or `ends_with` for `std::string` to express your intent.

Chapter 11

Language Updates

Now that we have seen the big four and a handful of smaller but still valuable features, namely the spaceship operator, the new ways of formatting a string in C++, the updates to lambdas, and, of course, the improvements in the STL, it is time to look at some smaller languages updates that have the potential to influence your daily coding, maybe faster than adjusting to a larger feature.

11.1 Range-based for-loops with initializers

We start with something, I think, every program consists of, loops. The most important thing about loops is that they must be carefully written to avoid an unwanted infinite loop. Another good practice is to use narrow scopes for all variables. This has always been a reason for choosing for loops over while loops. A missing piece until C++11 was a simple way to iterate over, for example, a `std::vector`. We have had this ability with range-based for-loops since C++11. Such loops make our code compact and easy to read and prevent errors in many cases. That all is probably no news.

11.1.1 Using a counter-variable in a range-based for-loop

One piece was still missing. Have a look at the following code:

Listing 11.1

```

1 std::vector<int> v{2, 3, 4, 5, 6}; A
2
3 size_t idx{0}; B Belongs to the for but is in outer scope
4 for(const auto& e : v) {
5   C Show the position using idx
6   printf("[%zd] %d\n", idx++, e);
7 }
```

We declare a `std::vector` `v`, in **A**, and fill it with some numbers. Then we declare, in **B**, the variable `idx` and initialize it with zero. Please note that `idx` is declared outside of the range-based for-loop that directly follows the declaration of `idx`. In this range-based for-loop, we iterate over the entire vector `v`, printing the value and its position. The position is where we need `idx`. As you can see in **C**, `idx` is also post-incremented to reflect the iteration step.

Now, this code is kind of okay. One issue is that `idx` belongs to the range-based for-loop, but this relationship is not evident. Imagine this code snippet being part of a large code base. Chances are high that, over time, `idx` will cease being directly before the range-based for-loop. Then, the relationship would be more obfuscated. Things would become even more problematic if some in-between code were to set `idx` to something other than zero.

A pragmatic solution we can apply here is to put both `idx` and the range-based for-loop in a scope, as you can see below in **A**:

```

1 std::vector<int> v{2, 3, 4, 5, 6};
2
3 { A The scope ties idx and the loop together
4   size_t idx{0};
5   for(const auto& e : v) { printf("[%zd] %d\n", idx++, e); }
6 }
```

Listing 11.2

The additional scope makes the intent of the code clear. It should prevent us from getting into trouble should the two disconnect over time. Yet, we have to write the scope and, as usual, indent the statements in the scope one more level. Our code then becomes a bit harder to read because the natural flow is broken. Depending on the maximal line length and indentation rules, we can end up with more lines than we had before.

This is where C++20, once again, assists us in keeping simple things simple. In a traditional `for`-loop, we always had the option of declaring a variable in the head. In range-based `for`-loops, this ability was lost, but it is now available in C++20:

```
1 std::vector<int> v{2, 3, 4, 5, 6};  
2  
3 for(size_t idx{0}; const auto& e : v) {  
4     printf("[%zd] %d\n", idx++, e);  
5 }
```

Listing 11.3

This is great! Not only do we no longer have to add the additional scope, range-based `for`-loops are now a bit closer to regular `for`-loops, which drops the need to remember differences between them.

One part could still be improved. What I like about `for`-loops is that in the head of a `for`-loop, we usually can see all that is going on: the loop variable that is declared, the condition when it terminates, and the post-action (iteration expression). Is the counter incremented by one, two, or ten? I think it would be nice if we could also have a post-action in range-based `for`-loops as well.

11.1.2 A workaround for temporaries

The new ability of range-based `for`-loops helps us out in another situation. However, it is a workaround and not an actual fix. Suppose we have the following code:

```
1 class Keeper { A  
2     std::vector<int> data{2, 3, 4};  
3  
4 public:  
5     ~Keeper() { std::cout << "dtor\n"; }  
6  
7     B Returns by reference  
8     auto& items() { return data; }  
9 };  
10  
11 Keeper GetKeeper() C Returns by value  
12 {  
13     return {};
```

Listing 11.4

```

14 }
15
16 void Use()
17 {
18     D Use the result of GetKeeper and return over items
19     for(auto& item : GetKeeper().items()) {
20         std::cout << item << '\n';
21     }
22 }
```

Listing 11.4

The `Keeper` class stores a `std::vector` and provides, with `items`, an access function. Noticeable here is that `items` returns a reference to its internal data. Then we have `GetKeeper`, a function that returns a `Keeper` object. Note that it is an object, not a reference or a pointer to it. These are the pieces we then use in D. Our use is to iterate over all elements in the vector in `Keeper`. We do so by calling `GetKeeper().items()` in the head of the range-based `for`-loop. Totally sensible code that compiles. The good news is it works as well. No, actually, that is not true. It is bad news if it works because, just right now, we are looking at UB. If this code works, it does so for the wrong reasons and can break anytime.

What is the UB? It is in how range-based `for`-loops bind to their range-initializer. In our case `GetKeeper().items()` (you can use C++ Insights to peak behind it):

```
1 std::vector<int> & __range1 = GetKeeper().items();
```

We end up with a reference to `std::vector`, returned by the temporary object `Keeper`. However, it is a reference to a temporary object on which we invoke another function `items()`. The temporary object `GetKeeper` is destroyed after the semi-colon and with that before we reach the body of the range-based `for`-loop. Chances are that this bug would go unnoticed because the contents would remain on the stack, but it remains a bug.

With C++20's range-based `for` with initializer, we can workaround this and split the calls `GetKeeper` and `items` like this:

```

1 for(auto&& items = GetKeeper(); auto& item : items.items()) {
2     std::cout << item << '\n';
3 }
```

Listing 11.5

By this split, `GetKeeper` gets the lifetime extension. Hence, the binding `items`. `items()` uses a valid object.

As I said, consider it a workaround. Without wanting to create too much hope, there is a chance that this will get fixed in C++23.

Use ref-qualifiers to prevent lifetime issues

With the help of ref-qualifiers, you can add another layer of safety without requiring C++20. By allowing `items` to only be called on an lvalue object, the broken code would not compile.

11.2 New Attributes

C++20 continues to unify compiler-specific attributes for us. We have three new attributes, `[[likely]]`, `[[unlikely]]`, and `[[no_unique_address]]`, in our basket for writing compiler-independent code without requiring macros.

11.2.1 likely / unlikely

We are looking at a way to give the compiler optimization hints with the two attributes `likely` and `unlikely`. Various compilers already had such a way. However, the syntax was different. Consider Listing 11.6.

```
1 switch(value) {  
2     case 0: puts("Hello"); break;  
3     case 1: puts("World"); break;  
4     case 2: puts("C++"); break;  
5 }
```

Listing 11.6

Here, we're looking at a `switch` with three different `case` labels. They are numbers, as you can see, and are sorted. Sometimes, it makes our code more readable if we sort numbers or elements in a certain way instead of using a random system. For example, missing cases are easy to spot that way.

While the code in Listing 11.6 is perfectly fine, what if we know that 1 is the most likely value, used 90% of the time? Then it should be the first label because the compiler then checks for it first, right? Well, then our sort system becomes random.

Listing 11.7

```

1  switch(value) {
2      case 0: puts("Hello"); break;
3      [[likely]] case 1: puts("World"); break;
4      case 2: puts("C++"); break;
5  }

```

There is no reason to mark the other cases as unlikely. The attributes are like a boolean. They are set or not. There is no way to specify an order between two or more `[[likely]]` or `[[unlikely]]` attributes. So, only mark the dominant statement in the control flow path with the attribute.

As a general rule, try to mark the *path*, which leads to the most likely statement or statements with the attributes.

11.2.2 no_unique_address

At this point, I want you to remember what we discussed in §8.3 on page 235, that aggregates can no longer have user-declared constructors. Otherwise, they are no longer aggregates. What if we want a aggregate that is default-constructible but not copyable?

Listing 11.8

```

1  struct Resources {
2      int          fileDescriptor;
3      std::string messageID;
4
5      A Ensure not copyable, but move works
6      Resources()           = default;
7      Resources(const Resources&) = delete;
8      Resources(Resources&&)     = default;
9  };
10
11 int main()
12 {
13     Resources r1{3, "buffer"};
14

```

```
15     //Resources r2{r1}; B Does not compile  
16 }
```

Listing 11.8

In Listing 11.8 on page 292, we have the aggregate `Resources`, well in C++17 `Resources` is an aggregate, with the new C++20 rules, it isn't anymore. But let's first go through what the idea behind `Resources` is. As the name of the type indicates, `Resources` holds some unique resources. Therefore, `Resources` should not be copied. What would a copy of a file descriptor mean? Definitely not a good idea to allow copying in such a case. In **A**, we see a typical pattern from C++17 and before, in which we delete the copy operation and default the default constructor as well as the move constructor. With that, we keep move alive. A reasonable approach. I have often seen macros to achieve this result and to reduce repetition. We cannot create a base class `NotCopyable` and let `Resources` derive from it because then we need to list the initializer of `NotCopyable` during the object creation.

Empty Base Optimization (EBO)

The term EBO refers to a technique in which a class A derives from another class B. If B does not contain any data members and those do not need memory, the compiler can optimize the size of B away. Usually, in C++, every object has a size of at least one. EBO is a special case and often used in generic code when a class should be instantiable by a user-provided type to reflect properties of that type without occupying additional space. Passing an allocator is such an example.

With the C++20 rules, `Resources` is no longer an aggregate. It contains a user-declared copy constructor, which, by the C++20 rules, no longer makes it an aggregate. Does this mean we can no longer create a non-copyable aggregate in C++20?

Well, not the same way as before, but there is a way. There is an interesting new kid on the block in the form of a new attribute `no_unique_address`. It was designed for situations like this. Its purpose is that we can have a type in a class that does not occupy memory. Hence no unique address. This means that a type marked with `no_unique_address` can have a zero size. There is a technique called EBO, which has been there since the beginning of C++, which has its downsides compared to `no_unique_address`.

```
1 struct NotCopyable {  
2     NotCopyable() = default;
```

Listing 11.9

```

3     NotCopyable(const NotCopyable&) = delete;
4     NotCopyable(NotCopyable&&)      = default;
5 };
6
7 struct Resources {
8     int          fileDescriptor;
9     std::string  messageID;
10
11    [[no_unique_address]] NotCopyable
12    nc;  A Ensure not copyable, but move works
13 };
14
15 int main()
16 {
17     Resources r1{3, "buffer"};
18
19     //Resources r2{r1}; B Does not compile
20 }

```

For EBO, we must derive from the empty base class. In a lot of cases, this makes no sense. In Listing 11.9 on page 293 `Ressources` could derive from `NotCopyable` instead of having a data member `nc` of type `NotCopyable`. In that case, EBO is possible. However, now we say that `NotCopyable` is a base of `Ressources`, which it isn't. We made the inheritance decision for efficiency reasons and not because an actual inheritance is modeled. So we misused EBO, which we no longer have to do with `[[no_unique_address]]`.

11.3 using enums

Class enums have been a valuable thing since C++11. As opposed to regular enums, their values are defined in the namespace of the enum. This helps us to keep our global namespace clean. One requirement that comes with class enums is that the enumerators need to be prefixed with the enum's name, which builds the namespace. This is the right thing in a lot of cases. However, imagine that you have to write a piece of code where, in a narrow scope, you need to access multiple if not all, enumerators

of such a class enum. Say you have a class enum `Permission` that contains the values `Read`, `Write`, and `Execute`. You must repeat the enum name three times for a simple `to_string` function. Listing 11.10 shows such an implementation.

```
1 constexpr std::string_view to_string(Permission permission)
2 {
3     switch(permission) {
4         case Permission::Read: return "Read";
5         case Permission::Write: return "Write";
6         case Permission::Execute: return "Execute";
7     }
8
9     return "unknown";
10 }
```

Listing 11.10

With just three values, there is already nothing to cheer for. With even more enumerants, it gets really annoying. The main difference with other uses of class enums is that this time, the scope is very narrow, only within the `switch`. Usually, in C++, this is where you can use `using` to make all elements of a namespace visible in that scope. Unfortunately, up to C++17, this wasn't allowed for class enums. It is now in C++20, which will enable you to refactor the code shown in Listing 11.10 to one that uses the new facility by pulling all enumerations into the scope of the `switch`, as Listing 11.11 shows.

```
1 constexpr std::string_view to_string(Permission permission)
2 {
3     switch(permission) {
4         using enum Permission; A New use of using
5         case Read: return "Read";
6         case Write: return "Write";
7         case Execute: return "Execute";
8     }
9
10    return "unknown";
11 }
```

Listing 11.11

As always, be careful with `using` and be sure to pull only what is really needed. The great advantage of class enums is that the enumerators are in a dedicated namespace.

11.4 conditional explicit

Another imbalance C++20 clears is the use of `explicit`. Before C++20, we could only state that a specific constructor or conversion operator is `explicit`. Recall that `noexcept` takes a condition to express whether or not a function is `noexcept`. While this sounds a bit weird, it is pretty helpful in generic code, where a function's `noexcept` behavior depends on a parameter type, for example. Let's explore what we can do with `explicit` in C++20.

11.4.1 Writing a well-behaved wrapper

Have a look at the code in Listing 11.12. It shows two classes `A` and `B`. In `A`, you can see that it comes with an `explicit` conversion constructor and a non-explicit conversion operator, both using `B`. Now, if there is a function `Fun` accepting type `A`, this function can be called with an `A` object but not with a `B` object.

```

1  struct B {};
2
3  struct A {
4      A() = default;
5      explicit A(const B&);  A Marked explicit
6      operator B() const;
7  };
8
9  void Fun(A a);
10
11 void Use()
12 {
13     Fun(A{});
14     // Fun(B{}); B Will not compile due to explicit ctor
15 }
```

Listing 11.12

Keeping the example of generic code, imagine you are writing a `Wrapper` that should wrap any type and model the wrapped type's behavior. Listing 11.13 shows such an approach.

```
1 template<typename T>
2 struct Wrapper {
3     template<typename U>
4     Wrapper(const U&);
5 }
6
7 void Fun(Wrapper<A> a); A Takes Wrapper<A> now
8
9 void Use()
10 {
11     Fun(A{});
12     Fun(B{}); B Does compile!
13 }
```

Listing 11.13

First, `Wrapper` is a class template that comes with a conversion constructor to allow `Wrapper` to be created with types convertible to the wrapped type. To test this solution, `Fun` now takes a `Wrapper<A>` instead of plain `A`, as you can see in Listing 11.14.

Although it might not look like it passes the tests at first glance, `Fun` is now callable with a `B` object. The issue is that even with Concepts, we need to write two conversion constructors, one that is marked `explicit` and the other that isn't. By using `std::is_convertible_v`, we can then direct to the matching conversion constructor. However, this requires nasty code duplication, which is required for each of these cases, meaning the conversion operator as well. This is what C++20 makes clear and easy to write, as seen in Listing 11.14.

```
1 template<typename T>
2 struct Wrapper {
3     template<typename U>
4     explicit(not std::is_convertible_v<U, T>) Wrapper(const U&);
5 }
```

Listing 11.14

With the power of a conditional `explicit`, all that is needed is `std::is_convertible_v`. No Concepts, no code duplication. The very same is applicable for a conversion constructor as well.

11.4.2 Communicate your intention, explicitly

Aside from what you just saw, the ability to easily create a well-behaved wrapper, there is an additional angle to the new conditional `explicit`. Consider the code in Listing 11.15. Think about what you would say about this code in a code review.

```

1  struct A {
2      A() = default;
3      explicit A(int);
4  };
5
6  struct B {
7      B() = default;
8      B(int);
9  };

```

Listing 11.15

My response is that I have to ask whether the constructor of `B` taking `int` is intentionally *not* `explicit`. With the code as presented, this isn't clear. Maybe there is a comment somewhere in the source code or the commit message. Anyway, it is not where it is needed. This is where we can use conditional `explicit` without any conditional part, just as a static `false`.

```

1  struct B {
2      B() = default;
3      explicit(false) B(int);
4  };

```

Listing 11.16

The code in Listing 11.16 now carries the author's intention. You can still question whether it is the right decision in a code review, but the decision is explicitly visible.

Cliff notes

- Remember that you can use conditional `explicit` stating your intention that a certain constructor should not be `explicit`.
- Mark only the dominant control flow path with either `[[likely]]` or `[[unlikely]]`.

Chapter 12

Doing (more) things at compile-time

C++ got a very interesting feature with C++11: `constexpr` functions and variables. `constexpr` allows us to execute code at compile-time with `constexpr` functions and to have the value of variables computed at compile-time with `constexpr` variables.

The functionality of `constexpr` functions has improved with each successive standard. C++20 is no exception. More restrictions have been lifted, and with `consteval` and `constinit`, two new kids are in town.

Before we dive into all the exciting new things we can do at compile-time with C++20, let's first recall what `constexpr` means and what limitations we had for `constexpr` functions before C++20.

12.1 The two worlds: compile- vs. run-time

The ability of the compiler to request a result of a function call at compile-time is an interesting twist to the language. The `constexpr` functions we could write with C++11 were limited to a single `return` statement. This often ended in recursive function calls when doing compile-time calculations.

Listing 12.1

```

1  A Define a constexpr function
2  constexpr size_t StrLen(const char* str)
3  {
4      B Use a single return and recursion with the ternary operator
5      return *str ? 1 + StrLen(str + 1) : 0;
6  }
7
8  int main()
9  {
10     constexpr char name[] {"Scott"};
11     auto          len = StrLen(name);
12
13     printf("%zd\n", len);
14 }
```

In this example, we see the implementation of a function that obtains the length of a C-string at compile-time. Therefore, `StrLen` is marked `constexpr`. Now, knowing what the rules concerning `constexpr` functions are is essential. One of them is: `constexpr` functions *can* be evaluated at compile-time. The emphasis is on *can!* In Listing 12.1, most likely `StrLen` is executed at run-time. Whenever we want to ensure that a certain `constexpr` function is executed at compile-time, we need to force the constant evaluation by assigning the result into a `constexpr` variable. So in order to ensure `StrLen` is executed at compile-time, we need to change the variable `len` as a `constexpr`:

Listing 12.2

```

1  constexpr char name[] {"Scott"};
2  constexpr auto len = StrLen(name);  A constexpr now
3
4  printf("%zd\n", len);
```

Now, the compiler is forced to evaluate that variable's initializer at compile-time to assign the value for `len` at compile-time. We are now in the compile-time path. Consequently, all data passed to `StrLen` must also be `constexpr`.

The critical thing with `constexpr` functions is that we need to force a `constexpr`-context, which in the standard is called a *constant expression*.

Luckily, with the progress of the language, certain limitations were lifted. Table 12.1 provides an overview of all the lifted limitations `constexpr` functions received up to C++20.

Table 12.1: Updates to `constexpr` functions from C++11 to C++20.

Functionality	11	14	17	20
<code>void</code> as return-type		✓	✓	✓
More than just a single return		✓	✓	✓
Using <code>throw</code> ^a		✓	✓	✓
<code>try/catch</code> -Block ^b				✓
<code>constexpr</code> member-function implicitly <code>const</code>	✓			
<code>inline</code> for static members with <code>constexpr</code>			✓	✓
Lambda can be implicitly <code>constexpr</code>			✓	✓
Use of <code>new</code> / <code>delete</code>				✓
<code>constexpr</code> virtual member functions				✓
Inline <code>asm</code>				✓

^a May not be called on the `constexpr` path.

^b As we can't throw, we can never reach the catch-block on the `constexpr` path.

We can see a couple of interesting changes in Table 12.1. In C++14, the restriction that a `constexpr` function could not return `void` was lifted. But why? As I demonstrated in the `StrLen` example, we need to assign the result to a `constexpr` variable, which is not possible if a function returns nothing. The reason lies in classes. Say we have a class `Point`:

```

1 constexpr Point move(Point p, double x, double y)
2 {
3     p.SetX(p.GetX() + x);
4     p.SetY(p.GetY() + y);
5
6     return p;
7 }
```

Listing 12.3

```

8
9 void Use()
10 {
11     constexpr Point p = move({2, 2}, 3, 2);
12
13     printf("x:%lf y:%lf\n", p.GetX(), p.GetY());
14 }
```

The code listing has two `constexpr`-methods that return `void`: `SetX` and `SetY`. Before C++14, these two methods were not allowed because `void` was not a literal type. With the change in C++14, we can use more normal constructs in a `constexpr`-context.

Since C++14, more than one `return` statement is allowed. This removes the urge for recursive functions and makes `constexpr`-functions look much more like regular run-time functions. We can also now have `throw` in a `constexpr` function, but only in the run-time path. What does that mean? See the example below first.

Listing 12.4

```

1 constexpr double divide(int a, int b)
2 {
3     if(b == 0) {
4         A If not 0 at compile-time, it is fine
5         throw std::overflow_error("Divide by zero");
6     }
7
8     return a / b;
9 }
10
11 int main()
12 {
13     constexpr auto goodConstexpr = divide(3, 2);    B constexpr path
14     auto          badRunTime   = divide(3, 0);    C run-time path
15 }
```

We have a function `divide` that divides two numbers `a` and `b`. As we all know, only Chuck Norris can divide by zero, but I'm not Chuck. A potential division by zero is caught in `divide`, and an exception is thrown if someone calls `divide` with zero. Due to the dual nature of `divide`, the input values, more precisely the value of `b`

, control whether we reach the `throw` in a `constexpr`-context. The example used in the code does not reach `throw` in the `constexpr` path but does in the run-time path. Now, the compilation would terminate if we would pass in the value 0 for `b` in the `constexpr`-context.

This is reasonable if you think about it. What would an exception mean at compile-time? With the given inputs, that exceptional path is always hit. That can't be good. This is a rule of thumb for all things going on at compile-time. We are allowed to have more and more statements in `constexpr`-functions, as long as we do not reach any bad statements during compile-time. The simple reason is that the dual nature of `constexpr` functions once again enables us to write a single function that is runnable both at compile- and run-time.

12.1.1 The benefit of compile-time execution

The benefit of calculating things at compile-time is that we do not pay with run-time for it. Further, if that function is never invoked in a run-time context, its code will not be in the resulting binary. Why should it? The function was only needed at compile-time.

Having a possibly faster and slimmer binary has advantages for users and vendors. Faster could either mean that a particular application can do more things now. For example, a device's battery could last longer because now there are fewer computations to achieve the same result.

The smaller binary size can enable vendors to add more features to the application or device. With that, users have a benefit as well.

12.2 `is_constant_evaluated`: Is this a `constexpr`-context?

Before C++20, there was no mechanism available to detect, in a function, whether the evaluation is compile- or run-time. This information is interesting, as certain constructs are not allowed in `constexpr` functions in the `constexpr`-evaluation path.

An early example is `throw`. Since C++14, we can have a `throw` expression in a `constexpr`-function so long as this expression is not hit during constant evaluation time. Once such an expression is hit during compile-time, the compiler announces an error stating that the function does not satisfy the requirements of an `constexpr`-function.

```

1 constexpr bool Fun(bool b)
2 {
3     if(b) { throw int{42}; }
4
5     return true;
6 }
7
8 static_assert(Fun(false)); A OK
9 //static_assert(Fun(true)); B Leads to compile error due to throw

```

Listing 12.5

Listing 12.5 demonstrates a very simple function containing a `throw`. Depending on the parameter `b`, `Fun` either throws or doesn't. The moment the function throws, compile-time evaluation fails.

In practice, we have a lot of cases like Listing 12.5, and all is good. The various relaxations on `constexpr` allow more and more elements in `constexpr`-functions. However, sometimes we need to know the difference to do various things.

Consider Listing 12.6, which illustrates another `constexpr`-function. The difference is that this time, `Fun` contains a call to a function `Log`. See this code as very loosely based on what we've seen in Listing 5.29 on page 169. Under the hood, `Log` may use `std::format`, `printf`, or `std::clog`, not all of which are not `constexpr` in C++20¹. Sure, printing during compile-time or logging raises the question of where the values end up. But what if I'm not interested in the log output during compile-time but want to have the output at run-time? Like in Listing 12.6?

```

1 constexpr bool Fun()
2 {
3     Log("%s\n", "hello");
4
5     return true;
6 }

```

Listing 12.6

Well, Listing 12.6 doesn't compile. `Log` must be `constexpr`, which isn't possible, if `Log` contains a call to `printf`. So let me introduce you to `std::is_constant_evaluated`. This new type-trait tells you in which evaluation context you are or your function. With that new tool, we can write either a wrapper

¹Chances are good that `std::format` will become `constexpr` in C++23, see [5]

or prepare Log, as Listing 12.7 shows, wrapping the non-constexpr call in an `if` that checks for the context.

```
1 template<typename... Args>
2 constexpr void Log(std::string_view fmt, const Args&... args)
3 {
4     if(not std::is_constant_evaluated()) {
5         printf(fmt.data(), args...);
6     }
7 }
```

Listing 12.7

The code in Listing 12.7 now allows us to call Log in both contexts. At compile-time, Log does nothing. At run-time, the function does its usual logging job. This change may not be much, but it allows more functions to become `constexpr`, as long as you can live with the fact that some don't do anything at compile-time.

12.2.1 Different things at compile- and run-time

We already discussed byte swapping with Listing 10.3 on page 269 in §10.2 on page 268. The solution as presented in Listing 10.3 on page 269 does three things:

- allowing chars to be passed to `ByteSwap` for consistency. Nothing happens for them;
- nothing is done on a big-endian machine;
- `ByteSwap` is `constexpr`.

With the features we discussed at the time, this was the best we could do, and I think the generic `ReverseBytes` is just incredible, but I like templates and lambdas. In reality, as great as `ByteSwap` was in Listing 10.3 on page 269, the solution had one flaw: the code may be less performant at run-time than by using just plain `htonl` and others. The reason for this is that compilers have special intrinsics for swapping bytes. For example, LLVM provides `_builtin_bswap32`. By hand-rolling this builtin, there is a risk that the compiler won't understand what we are doing anymore. Let's do better here. The `htonl` and others have always been good enough for run-time, so why remove them? This is where `std::is_constant_evaluated` comes in. With this facility, we can have different functions called for compile-time and run-time, as Listing 12.8 on page 308 shows.

Listing 12.8

```

1  template<std::integral T>
2  constexpr T ByteSwap(T value)
3  {
4      if constexpr(std::endian::native == std::endian::big ||
5                  (sizeof(value) == 1)) {
6          return value;
7      } else {
8          if(std::is_constant_evaluated()) {
9              return ReverseBytes(value);
10         } else {
11             if constexpr(std::same_as<T, uint64_t>) {
12                 return htonl(value >> 32) |
13                     (static_cast<uint64_t>(htonl(value)) << 32);
14             } else if constexpr(std::same_as<T, uint32_t>) {
15                 return htonl(value);
16             } else if constexpr(std::same_as<T, uint16_t>) {
17                 return htons(value);
18             }
19         }
20     }
21 }
```

The code in Listing 12.8 slightly differs from that in Listing 10.3 on page 269. The reason is that because the type is needed various times, a function template is better than an abbreviated function template. Other than that, all that's changed is that a `std::is_constant_evaluated` is used to decide whether `ReverseBytes` should be used or the `htons` and family. That way, you can be sure to always get the best performance.

As a side-note, in my tests, both Clang and GCC were able to optimize `ReverseBytes` into a `bswap` instruction at `-O1`. Compilers are awesome these days.

12.2.2 `is_constant_evaluated` is a run-time value

While I may have hooked you up with `std::is_constant_evaluated`, there is one important thing to understand: this function produces the value depending on whether something is constant evaluated, but that will always be true at compile-time. Let me show you an example using the previously shown `Log` implementation.

Someone clever might come up and say, why not use a `constexpr if` to evaluate `is_constant_evaluated`. This approach is shown in Listing 12.9.

```
1 template<typename... Args>
2 constexpr void Log(std::string_view fmt, const Args&... args)
3 {
4     if constexpr(not std::is_constant_evaluated()) {
5         printf(fmt.data(), args...);
6     }
7 }
```

Listing 12.9

Writing code like this may be tempting. On second thought, you may realize that asking in an *always* constant evaluated context, whether the evaluation is constant, isn't the right thing to do. In a `constexpr if`, `std::is_constant_evaluated` will always return `true` because the function is always evaluated during compile-time. Since this `constexpr if` always fails at compile-time, code inside the `if` block will not even be compiled, and hence, you will never see your log output, not even at run-time.

12.3 Less restrictive `constexpr`-function requirements

Just like the previous modern C++ standards, C++20 dropped a few restrictions to `constexpr`-functions, as Table 12.1 on page 303 shows. Among them are more usual candidates, like inline assembler in a `constexpr`-function, as well as allowing a `try`-`catch`-block. All are only permitted in the run-time path, so they have practically no effect during compile-time.

With dynamic allocations allowed in `constexpr`-functions, we are looking at a totally new ability, which opens the doors for various new applications.

12.3.1 new / delete: Dynamic allocations during compile-time

Potentially, the most significant change to `constexpr` is allowing dynamic allocations in these functions. Not only on the run-time path, no, we are allowed to use them during compile-time as well! Usually, `constexpr` allows more and more statements, but only for the run-time path. Before you get too excited, let's see how these dynamic allocations work.

The rule is that all memory allocated during compile-time must also be released at compile-time. This is a significant constraint on how such dynamic allocations can be used. The rationale is, where would the dynamic memory, allocated during compile-time, end up during run-time? One answer is that compile-time dynamic memory could be converted to memory on the stack. Such objects would then have to be immutable, such that nobody could free this memory. While this was discussed during standardization, this approach was discarded. Instead, we will start with allocations that can only live in exactly one of the two worlds.

The following dynamic allocation elements are allowed within a `constexpr`-function:

- a `new`-expression;
- a `delete`-expression;
- a call to an instance of `std::allocator<T>::allocate`;
- a call to an instance of `std::allocator<T>::deallocate`;
- `construct_at`;
- `destroy_at`.

Please note that a placement-`new` is not allowed for now. You can also not use overloaded `new`-operators inside your class. Only the global `new` is allowed.

12.3.2 A `constexpr std::vector`

The option to use dynamic allocations at compile-time brings a whole new load of applications. In C++20, this lifted restriction brings us a `constexpr std::vector`, as well as a `constexpr std::string`. Both are possible because of the dynamic allocations. But also the `try catch` block inside a `constexpr` function. To do all these tricks, `std::is_constant_evaluated` is also necessary.

12.4 Utilizing the new compile-time world: Sketching a car racing game

Now that we have seen the various `constexpr` improvements let's explore what we can do with them. For this, assume we are creating a car racing game. Well, only a sketch of it.

```
1  struct Car { A Base class for all cars
2      virtual ~Car() = default;
3      virtual int speed() const = 0;
4  };
5
6  B Various concrete cars with individual speed
7  struct Mercedes : Car {
8      int speed() const override { return 5; }
9  };
10 struct Toyota : Car {
11     int speed() const override { return 6; }
12 };
13 struct Tesla : Car {
14     int speed() const override { return 9; }
15 };
16
17 C A factory function to create a car
18 Car* CreateCar(int i)
19 {
20     switch(i) {
21         case 0: return new Mercedes{};
22         case 1: return new Toyota{};
23         case 2: return new Tesla{};
24     }
25
26     return nullptr;
27 }
```

Listing 12.10

All cars in our racing game derive from the base `Car` in **A**. Each concrete car is required to override `speed`, a pure virtual function in the base class `Car`. In **B**, you

see a few different cars. The entire game would, of course, provide more. Next, in **C**, you see a factory function, `CreateCar`, able to create the different cars by index. This method is crucial for our example. It knows the various concrete car types. Based on an integer value, `CreateCar` returns a concrete car object allocated on the heap. Maybe this method is created by the built environment, depending on the edition of our car racing game.

With that minimal part of a game, let's assume there is a dialog in the game showing the fastest car available. A dedicated function, `FastestCar`, determines the car by instantiating each available car's object and comparing the speed. Obviously, the one with the highest speed is the fastest car. Listing 12.11 shows an implementation.

```

1 int FastestCar()
2 {
3     int max    = -1;
4     int maxId = -1;
5     for(int i = 0; i < 3; ++i) {
6         auto* car = CreateCar(i);
7
8         if(car->speed() > max) {
9             max    = car->speed();
10            maxId = i;
11        }
12
13        delete car;
14    }
15
16    return maxId;
17 }
```

Listing 12.11

The method is pretty simple, but it gives us the desired number, which we can later use to display the fastest car. The downside of this approach is that the information about which car is the fastest is there at compile-time. No need to waste run-time. This is where we can apply all the new compile-time elements C++20 gives us, which is what Listing 12.12 on page 313 illustrates.

Listing 12.12

```
1  struct Car { A Base class for all cars
2      virtual ~Car() = default;
3      constexpr virtual int speed() const = 0;
4  };
5
6  B Various concrete cars with individual speed
7  struct Mercedes : Car {
8      constexpr int speed() const override { return 5; }
9  };
10 struct Toyota : Car {
11     constexpr int speed() const override { return 6; }
12 };
13 struct Tesla : Car {
14     constexpr int speed() const override { return 9; }
15 };
16
17 C A factory function to create a car
18 constexpr Car* CreateCar(int i)
19 {
20     switch(i) {
21         case 0: return new Mercedes{};
22         case 1: return new Toyota{};
23         case 2: return new Tesla{};
24     }
25
26     return nullptr;
27 }
```

We start by adding a `constexpr` to the `virtual` abstract method `speed` in the base class `Car`. Like the constructor, the destructor is implicitly `constexpr` when the compiler provides the implementation. Hence, there is no need to change the `virtual` default destructor.

Accordingly, to `speed` in the base class, all derived classes overridden methods are marked with `constexpr` as well.

What is left to do is mark `CreateCar` and `FastestCar` `constexpr`. The last one is shown in Listing 12.13 on page 314, despite this change being a no-brainer.

```

1 constexpr int FastestCar()
2 {
3     int max    = -1;
4     int maxId = -1;
5     for(int i = 0; i < 3; ++i) {
6         const auto* car = CreateCar(i);
7
8         if(car->speed() > max) {
9             max    = car->speed();
10            maxId = i;
11        }
12
13        delete car;
14    }
15
16    return maxId;
17 }
```

Listing 12.13

That's all. We are now ready to use `FastestCar` at compile-time and determine the index of the fastest car there. Store this index and have the value ready at run-time with no overhead or computation time.

This is possible because in C++20, we can have

- a `constexpr` destructor;
- `virtual` functions that are `constexpr`;
- dynamic memory allocations in `constexpr` functions.

The only caveat is the raw pointer that is returned by `CreateCar`. To my great sadness, `std::unique_ptr` isn't `constexpr` in C++20. In production code, I would probably write a tiny custom `unique_ptr` to tidy up my code. Listing 12.14 shows how this changes the code. However, I improved this point in C++23 with [6]. In C++23 `std::unique_ptr` is `constexpr`.

```

1 constexpr std::unique_ptr<Car> CreateCar(int i)
2 {
3     switch(i) {
4         case 0: return std::make_unique<Mercedes>();
```

Listing 12.14

```
5     case 1: return std::make_unique<Toyota>();
6     case 2: return std::make_unique<Tesla>();
7   }
8
9   return nullptr;
10 }
11
12 constexpr int FastestCar()
13 {
14     int max    = -1;
15     int maxId = -1;
16     for(int i = 0; i < 3; ++i) {
17         if(auto car = CreateCar(i); car->speed() > max) {
18             max    = car->speed();
19             maxId = i;
20         }
21     }
22
23     return maxId;
24 }
```

Listing 12.14

I hope very much that in C++23, you can write the code as shown in Listing 12.14 on page 314.

12.5 consteval: Do things guaranteed at compile-time

C++20 brings a new keyword to the compile-time world: `consteval`. This keyword can be used to mark functions, forcing the compiler to always evaluate them always at compile time. Should the input values not allow this, you get a compiler error. In general, `consteval` works exactly as `constexpr`. All statements that aren't allowed there aren't allowed in a `consteval` function. The big difference to `constexpr` is that `consteval` doesn't have a dual nature. We have now the ability to distinguish between three cases:

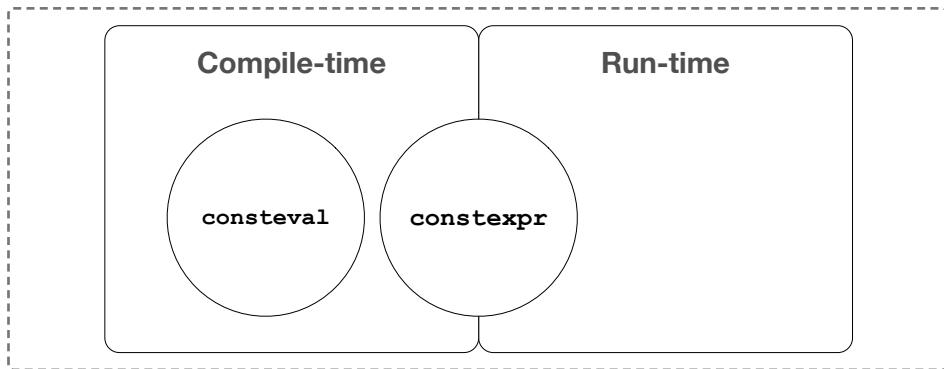


Figure 12.1: Categorization of the different compile-time modifiers.

- 1 `consteval`: Compile-time only. We get a compile error if the function cannot be evaluated at compile-time or if the function contains forbidden statements in the compile-time path.
- 2 `constexpr`: Either compile- or run-time. Depending on the input values and whether the resulting value is used at compile time, it is evaluated at compile time but can also be executed at run time.
- 3 No attribution: A regular run-time function. Executed solely at run-time.

Figure 12.1 visually represents the description.

Using `consteval` is helpful in scenarios where you want to ensure that the code of a specific function *never* hits the binary or that the value is computed at compile time.

12.5.1 as_constant a use-case for `consteval`

From `constexpr`, you probably know that having a `constexpr`-function is only one part. To force its execution at compile-time, the function needs to be forced into compile-time evaluation. The approach to do this is to assign the function's result to a `consteval` variable, as Listing 12.15 shows.

```

1  constexpr int ExpensiveCalculation(int base)
2  {
3      return base + 1; // Well, ... expensive

```

Listing 12.15

```

4 }
5
6 void Use()
7 {
8     auto          value1 = ExpensiveCalculation(2);   A run-time
9     constexpr auto value2 = ExpensiveCalculation(2);   B compile-time
10
11    //++value2;  C Doesn't compile, value2 is const
12 }
```

Listing 12.15

In Listing 12.15 on page 316, A is a run-time call because the value isn't needed at compile-time. Your compiler and optimizer may disagree and give you the result anyway. Still, from a standards perspective and without betting on optimization capabilities, A is called at run-time.

This is why we often make the variable `constexpr`, as B illustrates. We practically force the compiler to evaluate the variable's value at compile time. Except for the additional characters for the `constexpr` keyword on the variable, this approach is fine... as long as you don't need to later alter the variable. This is what C shows, and because the increment doesn't compile, this line of code is only a comment. The reason is that a `constexpr` variable implies `const`. You can use C++ Insights to peek behind what's going on here.

With the help of `constexpr` and abbreviated function templates (see §1.9 on page 34), we can build ourselves a handy little helper `as_constant`:

```

1 constexpr auto as_constant(auto v)
2 {
3     return v;
4 }
```

Listing 12.16

Using this helper, we have superpowers that allow us to do the following:

```

1 auto value2 = as_constant(ExpensiveCalculation(2));   B compile-time
2
3 //++value2;  C Compiles
```

Listing 12.17

Because `consteval` is always evaluated at compile-time, we can pass any `constexpr` function or constant value to `as_constant`, forcing compile-time evaluation on it. The lovely result is that in B we can get rid of the `constexpr` for the variable, which enables us to later modify `value2` because the variable is no longer implicitly `const`.

There is one caveat to `as_constant`. This helper requires not only the constructor to be `constexpr` but also the destructor.

12.5.2 Force compile-time evaluation for compile-time checks

While the `as_constant` helper has its benefits, I wish to explore a bit more what you can do with `consteval` to improve your code. Let's assume we want to build a string formatting function, `format`, as shown in Listing 12.18.

```
1 template<class... Args>
2 string format(std::string_view fmt, const Args&... args);
```

Listing 12.18

The function `format` is a typical variadic template, helping us to maintain the type-safety. One issue we have is how do we ensure that the format string matches the parameter? For standardized elements like `printf`, compilers do such a check at compile-time and warn once they find that a format specifier doesn't match the type. The problematic part is, they can do this only for the functions and types they know. For our custom `format` function, they don't because they cannot know whether `std::string_view` (Std-Box 3.1 on page 117) here really means format string. Plus, when we add custom format specifiers, they are also lost. We can teach compilers this using various compiler-specific attributes. Such an approach requires additional knowledge and makes the code compiler-dependent. The more critical part is we have to remember this step! Shouldn't we tell the compiler about such a check so no one will notice?

The ability to write such a check in the language we use to write the rest of our code is beneficial, as is sharing code. Why bother writing a compile-time check if we still need a run-time check? Let's approach this by altering the signature of `format`, or more precisely, the format string slightly as illustrated in Listing 12.19.

```
1 template<class... Args>
2 string format(format_string<Args...> fmt, const Args&... args);
```

Listing 12.19

Yes, that isn't much. All we did was change `std::string_view` to a type called `format_string`. The parameter pack `Args` of `format` is forwarded to `format_string`, but only the types, not the values. This may seem like a minor change, but this change is what enables the rest of the machinery. Now, `format_string` knows the format string and the types. Remember, we do not need to care for actual values at this point. The check is only about whether a format specifier matches the given type. What is behind `format_string`? Look for yourself in Listing 12.20.

```
1 template<class... Args>
2 struct basic_format_string {
3     std::string_view str; A Holds the actual format string
4
5     template<class T>
6     requires std::is_convertible_v<const T&, std::string_view>
7     consteval basic_format_string(
8         const T& s) B A consteval constructor
9     : str{s}
10    {
11        string out{};
12        C Use the regular format function
13        Formatter(out, str, Args{}...);
14    }
15};
16
17 D Make constructor working without arguments
18 template<class... Args>
19 using format_string =
20     basic_format_string<std::type_identity_t<Args>...>;
```

Listing 12.20

What you can see is that `format_string` is a `struct` template that stores a `std::string_view`. This `std::string_view` was previously exposed in the signature from `format`. All `basic_format` has in addition is a `consteval` constructor. But this is the key. Thanks to other relaxations, namely dynamic allocations in `constexpr` functions, we can use `std::string` together with the regular `Formatter` function. Sharing the code that does the actual formatting at

run-time with the compile-time code that leverages the `throw` triggered in case of a formatting error, which yields a compile-time error.

This is an example of how various independent-looking features come together. The power of `consteval` helps us to force the constructor into a compile-time execution. Then, the dynamic allocations during compile-time allow a `constexpr std::string`. Such a compile-time `std::string` will enable us to share the formatter code in compile-time. Absolutely no need for a dedicated check function.

One caveat I wish to point out lies in [C](#). The whole thing only works with default constructible types. A close look reveals that in [C](#) I'm using `Args` to create all necessary parameters via braced initialization.

12.5.3 `is_constant_evaluated` doesn't make it compile-time

Coming from the last section, we learned to use `std::is_constant_evaluated` only in a regular `if`. This brings another caveat. We cannot call a `consteval` function in the `std::is_constant_evaluated` path. Consider the example in Listing 12.21.

```

1  consteval int CompileTime(int i)
2  {
3      return i;
4  }
5
6  constexpr int Dual(int i)
7  {
8      if(std::is_constant_evaluated()) {
9          return CompileTime(i) + 1;
10     } else {
11         return 42;
12     }
13 }
```

Listing 12.21

We have a `constexpr` function `Dual`. Inside of this function, a `std::is_constant_evaluated()` is used to determine the mode. The desire is to call `CompileTime` during compile-time. This is a `consteval` function and, therefore, must be executed at compile-time. At first glance, the code looks reasonable, but if you feed this to your compiler, it will reject the code. The thing is how the machinery

works internally, specifically for run-time. The `std::is_constant_evaluated()` is used in a run-time `if`. It must be as we learned in §12.2.2 on page 308. The result for the run-time path is, that `std::is_constant_evaluated()` returns `false`, making the code look like this:

```
1 constexpr int Dual(int i)
2 {
3     if(false) {
4         return CompileTime(i) + 1;
5     } else {
6         return 42;
7     }
8 }
```

Listing 12.22

Now, `if` is constantly false, but its body is there. The optimizer or the compiler will kick the body out at some point, as they understand that the code there is dead code for run-time. However, this happens after the body of the `if` is evaluated. But this evaluation triggers a call to `CompileTime`, our `consteval` function. A call to this function requires passing the parameter `i`, which is now a run-time value. This is a run-time value, which makes the evaluation of `CompileTime` outside of a compile-time context impossible, leading to a compiler error.

12.6 `constinit`: Initialize a non-const object at compile-time

The new keyword we talk about in this section, `constinit`, is slightly different from what we have with `constexpr` or `consteval`. While both `constexpr` and `consteval` directly lead to a change, `constinit` is an assertion that asserts that a variable is initialized with constant initialization.

The scope of `constinit` is therefore limited to variables with static storage duration, e.g., global variables or `static` variables. Table 12.2 on page 322 provides an overview of the different compile-time keywords and their application areas.

Table 12.2: The application areas of `constexpr`, `constexpr`, and `constinit`.

Keyword	Function	Local variable	Global variable	Local static variable
<code>constexpr</code>	✓	✓	✓	✓
<code>consteval</code>	✓			
<code>constinit</code>			✓	✓

12.6.1 The static initialization order problem

Asserting that a variable is initialized at compile-time is valuable in the case you are fighting with what is called the static initialization problem. Consider the code in Listing 12.23.

```

1  struct Air {
2      Air(int amount)
3      : _amount{amount}
4  {}
5
6  void Consume(int v) { _amount -= v; }
7  int Available() const { return _amount; }
8
9  private:
10    int _amount;
11 };
12
13 struct Human {
14     Human(int breath);
15 };
16
17 Air air{9};    A Create global air object
18 Human human{5}; B Create global human object
19
20 Human::Human(int breath)
21 {
22     air.Consume(breath); C Depends on air
23 }
```

Listing 12.23

Here we have two data structures, `Air` and `Human`. Whenever a `Human` object is created, the object consumes air. Hence, there is a call to `air.Consume` in the constructor of `Human`. Certainly, this is something you should be careful of anyway when dealing with global variables. Still, sometimes, this is the code we (have to) write.

The first thing I wish to point your attention to is that `Air` is fully implemented in the declaration. It could as well be one of these header-only types. Now, for some reason, `Human` only declares its constructor, but the implementation is out-of-line. Imagine it being in a `cpp`-file.

The second thing is that in **A** and **B** two objects of each datatype are created. The `Air` object initially contains enough air for a single human object.

The question now is, if we check, say in `main`, `air.Available`, what is the number returned? Okay, the math is easy, $9 - 5 = 4$. Do we agree that 4 is... our expectation? Looking at the code as presented, this expectation is true.

Now, what if the code is changed slightly, as shown in Listing 12.24?

```
1 Human human{5};    B Create global human object
2 Air    air{9};     A Create global air object
```

Listing 12.24

The difference here is, that the `Human` object is created *before* the `Air` object. Do you still think that the result of `air.Available` is 4? If not, what is the result?

The answer is 9. Because `air` has not been constructed while `human` was constructed. This is what is called the static initialization problem. The code may work for some time until some refactoring moves the two variables around. Maybe they are in two different translation units, and the new build systems chose a separate compilation order. Bam, our program starts behaving differently for no apparent reason.

Making the constructor of `Air constexpr` is a way to fix this situation. The code shown in Listing 12.25 does work, regardless of the declaration order of the two objects `air` and `human`.

```
1 struct Air {
2     constexpr Air(int amount)
3     : _amount{amount}
4 }
5
```

Listing 12.25

```

6   void Consume(int v) { _amount -= v; }

7

8   int Available() const { return _amount; }

9

10 private:
11     int _amount;
12 };

```

12.6.2 Ensure compile-time initialization

Now that `constexpr` helps us to fix this situation by introducing constant initialization, it is excellent, but how can we be sure? The answer is we can't. Some refactoring, removing the `constexpr` from the constructor, is all that is needed. This is where `constinit` comes into play. The job of `constinit` is to assert that constant initialization happens. Once applied to the variable that must be initialized at compile-time, as shown in Listing 12.26, the compiler returns an error if the condition isn't met.

```

1 Human           human{5};   B Create global human object
2 constinit Air air{9};    A Create global air object

```

You can say that all `constinit` does is check that the constructor of the object is `constexpr` or `consteval`.

Table 12.3: Meaning of `constexpr` and `constinit`.

Keyword	Function
<code>constexpr</code>	Evaluated at compile-time, <code>const</code> at run-time
<code>constexpr const</code>	Redundant, use only <code>constexpr</code> , which implies <code>const</code>
<code>constinit</code>	Evaluated at compile-time, changed at run-time
<code>constinit const</code>	A code-smell. Use only <code>constexpr</code>

Applying `constinit` here is precisely what's needed. We cannot use `constexpr` for the variable `air`. This implies an implicit `const`, making it impossible to take a deep breath from `air`. You can see `constexpr` as `constinit` plus `const`. This gives us the ability to differentiate.

Table 12.3 presents an overview of what `constexpr` and `constinit` mean. The table shows what to avoid as well.

Cliff notes

- Never use `std::is_constant_evaluated` in a `constexpr if`.
- Use `constinit` to assert constant initialization and, with that, prevent the static initialization problem.

Acronyms

ABI	Application Binary Interface.	ISBN	International Standard Book Number.
ADL	Argument Dependent Lookup.	MRN	Medical Record Number.
API	Application Programming Interface.		
ASCII	American Standard Code for Information Interchange.	NaN	Not a Number.
AST	Abstract Syntax Tree.	NTTP	non-type template parameter.
BCD	Binary Coded Digit.	OOP	Object Oriented Programming.
CRTP	Curiously Recurring Template Pattern.	PIMPL	Pointer to implementation.
CTAD	Class Template Argument Deduction.	POSIX	Portable Operating System Interface.
DesDeMo	Destructor defined Deleted Move Assignment.	RVO	Return value optimization.
EBO	Empty Base Optimization.	SFINAE	substitution failure is not an error.
FSM	finite state machine.	STL	Standard Template Library.
GP	Generic Programming.	TCP/IP	Transmission Control Protocol / Internet Protocol.
IDE	Integrated Development Environment.	TMP	Template Meta-Programming.
IEEE	Institute of Electrical and Electronics Engineers.	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_lIA
- [5] V. Zverovich, “std::format improvements,” feb 2021. [Online]. Available: <http://wg21.link/p2216r3>
- [6] A. Fertig, “P2273r3: Making std::unique_ptr constexpr,” ISO JTC1/SC22/WG21, 11 2021. [Online]. Available: <https://wg21.link/p2273r3>

Index

A

ABI	139, 140
ADL	179
aggregate	193, 223–227, 229, 232, 234–237, 239, 241, 242, 244, 246, 248, 250, 292, 293
API	70, 73, 138, 141, 158, 212, 241, 272
ASCII	78
AST	140

B

BCD	194
-----------	-----

C

C++11	
User-defined literals	255
User-provided vs. user-declared	224
C++14	
A digit separator	148
A sample element	10
Generic lambdas	210
Lambda init-capture	215
Variable templates	19
C++17	
constexpr if	41
std::optional	44
std::string_view	117
Class Template Argument Deduction	241
Class Template Argument deduction guides	244

Fold expressions	20
concept	18
consteval	301, 321
constexpr	301, 317
function	301, 318
if	309
variable	301, 303, 316
constinit	301, 325
constrained	
auto return-type	39
auto types	38
auto variables	38
destructor	47
constraint	
ad hoc	30, 50
placeholder type	22
placeholder variable	39
type	22, 23, 41
coroutine	65, 67–71, 73–76, 78, 82–85, 88, 90, 91, 94, 96, 98–101, 103–105
async_generator	83, 86, 92
generator	70, 72, 73, 81, 88, 90, 94, 96
stackful	69, 100
stackless	69
CRTP	126
CTAD	127, 241, 243, 244, 252, 270, 271

D

DesDeMovA	96
designated initializers	225–232, 234, 246

E

EBO 293, 294

F

fmtlib 153

fold expression 20, 27, 259, 260

FSM 69, 70, 73, 94

function template

 variadic 18

G

global module fragment 135

GP 17, 34

H

header unit 134, 136, 143

I

IDE 38

IEEE 250

iostream 148–153, 155, 159, 161, 165, 173

ISBN 205, 206, 209

iterator

 bidirectional 123

 contiguous 123

 forward 123

 input 123

 output 123

 random_access 123

L

lambda

 captureless 208

 generic 210, 211

 generic variadic 212

M

module 134, 143

 implementation 135, 136

 interface 135, 136

 named 134–136, 139, 140, 143

MRN 176, 177, 187

N

NaN 188, 249

NTTP 31, 33, 37, 93, 172, 210, 247–252, 254, 256, 262, 263

O

OOP 81

out-of-line 142

P

pack expansion 20

parameter pack 20

PIMPL 140

POSIX 152, 241, 242, 268

promise type 71

pull model 118

Purview 135

R

range 114

 adaptor 118–122, 125, 127, 128, 130

 algorithm 116, 122

 common 114, 120

 concept 122

 lazy 118

 sized 115

 view 117, 125

requires

 clause 22, 23, 26, 41, 44–50, 63

 expression 23–29, 38, 40, 63

 RVO 230, 231

S

- s 70, 139, 140, 143
- sentinel type 115
- SFINAE 21, 40
- STL 17, 28, 29, 39, 56, 57, 62, 105, 107, 110–115, 119, 125, 127, 132, 186, 188, 197, 219, 232, 254, 267, 274, 276, 277, 282, 284, 287

T

- template
 - abbreviated function ... 34–36, 39, 269, 308, 317
 - class 56, 297
 - function .. 34, 35, 40, 41, 56, 63, 66, 243, 246, 247, 254, 258, 259, 267, 308
 - variable 262
- variadic . 17, 87, 128, 153, 169, 211, 214, 215, 256, 261, 262, 318
- variadic function 20, 21, 78
- TMP 17, 19

U

- UB . 88, 105, 110, 111, 118, 204, 218, 266, 274, 290
- UDL 255, 256, 261, 262
- UI 145