C++ Concepts

Validate your templates compile-time

Sandor Dargo

www.sandordargo.com

C++ Concepts

Validate your templates compile-time

Sandor Dargo

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

This version was published on 2022-07-15

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

Contents

Introduction

The purpose of this book is to give you all the information you need to get started with C++ concepts.

This simple phrase might spark two questions in your head. Why "all the information you need to start with C++ concepts" instead of simply giving all the details?

And why C++ concepts? Why not another feature? Why not C++20 in general?

Let me answer these questions.

Why not a full reference with all the details?

There are books aiming to replace and complement C++ Reference[1] and other full(ish) online documentation of the C++ language and the standard library. Some of them are great, but regardless of their quality, they all share a common characteristic. They are not meant to be a joyful read cover to cover and to my taste, they repeat too much information that is already available and digestible to you.

I prefer books that give you the main concepts (pun intended) and reveal the ideas behind and explain how to benefit from those ideas in your codebase and for the rest, you have the references.

Hence, for example, you won't find an enumeration of all the standard concepts in this book just to pump up the number of pages. Instead, I'll explain a few of them and you'll get the references for all the rest.

I'm a strong believer in the Pareto principle[2] and I know that most of us don't need 100% of the available knowledge to be efficient. We need much less, but we need it in an understandable form and just in time. This book will give you all that you need right now.

Why concepts and not something else? Why not a full review of C++20?

Concepts fascinated me from the first moment I read about them. I found the idea great and the examples expressive. They also helped me to delve into templates that are something I was afraid of quite a bit.

I'm not sure why.

[1] https://en.cppreference.com/w/
[2] https://en.wikipedia.org/wiki/Pareto_principle

Maybe because template metaprogramming doesn't follow the usual runtime programming, maybe because their error messages are intimidating, maybe because…

Anyway, concepts gave me the push and they can give you too.

If you don't need that nudge because you already use templates with ease, even better. You'll still learn about a brand new feature of C++20, concepts.

I didn't want to write a full book on C++20 because there are already some great books on this topic such as C++20: Get the Details by Rainer Grimm[3]; and also because I wouldn't have had enough time and space to focus on this particular feature on which I'm going to provide you with all what you need to have efficient start with concepts.

Thanks for buying this book and I wish you a pleasant journey with concepts!

[3] https://leanpub.com/c20

The concept behind C++ concepts

Concepts are one of the major new features added to C++20. Concepts are an extension for templates allowing direct expression of the programmer's intent. Though the language creator Bjarne Stroustrup rather wrote in the conclusion of one of his papers[4] that *"concepts complete templates as originally envisioned "*.

Concepts can be used to perform compile-time validation of template arguments through boolean predicates. Checks are performed at the point of call, therefore the error messages are more understandable than previously with bare templates when the error messages appear at the time of instantiation.

Concepts can also be used to perform function dispatch based on the properties of types.

With concepts, you can *require* both syntactic and semantic conditions. In terms of syntactic requirements, imagine that you can impose the existence of certain functions in the API of any class. For example, you can create a concept `car` that requires the existence of an `accelerate` function:

```
#include <concepts>

template <typename C>
concept car = requires (C car) {
  car.accelerate()
};
```

> In this book, you'll mostly encounter examples with only a few requirements. And while we still lack the best practices on how to write good concepts as it's a new feature, there is already an agreement in the community that good concepts do not specify a minimum number of requirements.
>
> While it might seem a good idea at first, you're likely to create concepts that are satisfied accidentally. Write concepts completely modelling an idea in a semantically coherent way and you'll end up with more widely usable concepts. But don't expect to reach that point on the first iterations.
>
> Having said that, in this book you'll mostly find "incomplete" concepts so that we can focus on particular parts.

Anything that you'd put in the `requires` clause describes syntactic requirements of the types *modelling* a concept.

[4] https://www.stroustrup.com/good_concepts.pdf

Don't worry about the syntax, we'll discuss that in detail soon.

Concepts don't only express syntactic requirements but also semantic ones. A concept carries a semantic meaning, even though it's difficult and not always possible to force the modelling types to satisfy the semantic meanings.

It makes sense somehow. Syntax focuses on how to express something, in our case, what API should a class have. On the other hand, semantics focus on meaning. That's a point where compilers can help less and more responsibility is on the shoulders of the developers.

Some semantic requirements are related to mathematical axioms, for example, you can think about associativity or commutativity:

```
1 a + b == b + a // commutativity
2 (a + b) + c == a + (b + c) // associativity
```

These can be expressed with code, in fact, we just partly did that.

Others, like the time or space complexity of an operation, cannot be and we have to rely on code comments or library documentation.

If you are looking for examples, it's worth looking into the standard library. Take for example `std::equality_comparable`[5].

It requires that

- the two equality comparison between the passed in types are commutative,
- == is symmetric, transitive and reflexive,
- and `equality_comparable_with<T, U>` is modeled only if, given any lvalue `t` of type `const std::remove_reference_t<T>` and any lvalue `u` of type `const std::remove_reference_t<U>`, and let `C` be `std::common_reference_t<const std::remove_reference_t<T>&, const std::remove_reference_t<U>&>`, then `bool(t == u) == bool(C(t) == C(u))`.

Now, let's think about a more real-life example, cars. If we consider a car an interface, how would a - simple - interface look like?

It probably have methods like:

- `operDoor()/closeDoor()`
- `startEngine()/stopEngine()`
- `accelerate()`
- `brake()`

We could write a very simple concept:

[5] https://en.cppreference.com/w/cpp/concepts/equality_comparable

```
template<typename C>
concept car = requires (C c) {
    c.openDoor();
    c.closeDoor();
    c.startEngine();
    c.stopEngine();
    c.accelerate();
    c.brake();
};
```

And then let's say, you have a class called `Tank`:

```
class Tank {
public:
    void openDoor();
    void closeDoor();
    void startEngine();
    void stopEngine();
    void accelerate();
    void brake();
    void fireCannon();
    void fireMachineGun();
private:
    // ...

};
```

This class satisfies the syntactic requirements of a `car` and function taking one can be called with an instance of a `Tank`:

```
#include <iostream>
#include <numeric>
#include <string>
#include <vector>

template<typename C>
concept car = requires (C c) {
    c.openDoor();
    c.closeDoor();
    c.startEngine();
    c.stopEngine();
    c.accelerate();
```

```
13      c.brake();
14  };
15
16  class Tank {
17  public:
18      void openDoor();
19      void closeDoor();
20      void startEngine();
21      void stopEngine();
22      void accelerate();
23      void brake();
24      void fireCannon();
25      void fireMachineGun();
26  private:
27      // ...
28
29  };
30
31  void foo(car auto c) {
32      std::cout << "foo\n";
33  }
34
35  int main()
36  {
37      Tank t;
38      foo(t);
39  }
40  /*
41  foo
42  */
```

Still we cannot say that a tank is a car, therefore or `Tank` class doesn't semantically satisfy the `car` concept, yet for syntactic reasons it compiles.

A good source to learn specifically about semantic requirements and some tricks to express them in concepts is this article by Andrzej Krzemieński[6].

The motivation behind concepts

We have briefly seen from a very high level what we can express with concepts. But why do we need them in the first place?

[6] https://akrzemi1.wordpress.com/2020/10/26/semantic-requirements-in-concepts/

For the sake of example, let's say you want to write a function that adds up two numbers. You want to accept both integral and floating-point numbers. What are you going to do?

You could accept `doubles`, maybe even `long doubles` and return a value of the same type.

```
#include <iostream>

long double add(long double a, long double b) {
    return a+b;
}

int main() {
  int a{42};
  int b{66};
  std::cout << add(a, b) << '\n';
}
```

The problem is that when you call `add()` with two `ints`, they will be cast to `long double`. You might want a smaller memory footprint, or maybe you'd like to take into account the maximum or minimum limits of a type. And anyway, it's not the best idea to rely on implicit conversions.

> **Implicit conversions** are performed whenever an expression of some type is used in a context that does not accept that type, but the two are compatible. For example, it might happen that you have a number with the type of `short` at hand, but the function you're calling takes an `int`. In that case, an implicit conversion will be performed and the `short` will be promoted to an `int`. Such conversions where a smaller type is transformed into a bigger one is called *promotion* and they are guaranteed to provide the same value.
>
> On the other hand, if the transformation happens in the other direction, the value of the transformed value might be different. It's called narrowing and it can be very dangerous.
>
> And there is even worse than that. Implicit conversions might allow code to compile that was not at all in your intentions. Constructors taking exactly one parameter might perform a type conversion implicitly in order to satisfy the compiler and make your code run even if it is by mistake. Hence it's a best practice to declare constructors taking one parameter explicit.
>
> Implicit conversions are not bad by definition, but you should use them on purpose and not accidentally.

Defining overloads for the different types is another way to take in order to support multiple types, but it is definitely tedious.

```cpp
#include <iostream>

long double add(long double a, long double b) {
  return a+b;
}

int add(int a, int b) {
  return a+b;
}

int main() {
  int a{42};
  int b{66};
  std::cout << add(a, b) << '\n';
}
```

Imagine that you want to do this for all the different numeric types[7]. Should we also do it for combinations of different *numeric* types, like `long doubles` and `shorts`? Eh... Thanks, but no thanks.

Another option is to define a template!

```cpp
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a+b;
}

int main() {
  int a{42};
  int b{66};
  std::cout << add(a, b) << '\n';
  long double x{42.42L};
  long double y{66.6L};
  std::cout << add(x, y) << '\n';

}
```

If you have a look at C++ Insights[8] you will see that code was generated both for an `int` and for a `long double` overload. There is no `static_cast` taking place at any point.

Are we good yet?

[7] https://en.cppreference.com/w/cpp/language/types
[8] https://cppinsights.io/s/5b15a333

Unfortunately, no.

There are a couple of issues. The requirements of a template are implicit, you might say hidden in the function body. You have to read through the implementation to see what "requirements" the template arguments have to satisfy.

Another often cited problem with templates is the verbose, yet difficult to understand error messages. One of the reasons behind such error messages is that the errors appear not when the template is called, but when it is instantiated.

And there are even more issues.

What happens if you try to call `add(true, false)`? You'll get a `1` as `true` is promoted to an integer, summed up with `false` promoted to an integer and then they will be turned back (`static_cast`ed) into a boolean.

What if you add up two string? They will be concatenated. But is that really what you want? Maybe you don't want that to be a valid operation and you prefer a compilation failure.

And we haven't even mentioned `chars`. You add up two `chars` and they first are cast to `ints` and once those are summed up they will be converted back to a `char` with a fair chance of overflowing. The max value of a `char` is 127, there is no big margin available anyway.

So you might have to forbid some template specialization[9]. But for how many types do you want to do the same?

```
1  #include <iostream>
2  #include <string>
3
4  template <typename T>
5  T add(T a, T b) {
6      return a+b;
7  }
8
9  template<>
10 std::string add(std::string, std::string) = delete;
11
12 int main() {
13   std::cout << add(std::string{"a"}, std::string{"b"}) << '\n';
14 }
15 /*
16 main.cpp: In function 'int main()':
17 main.cpp:13:54: error: use of deleted function
18 'T add(T, T) [with T = std::__cxx11::basic_string<char>]'
19    13 |   std::cout << add(std::string{"a"}, std::string{"b"}) << '\n';
20       |                                                      ^
```

[9] https://dev.to/pgradot/forbid-a-particular-specialization-of-a-template-4348

```
main.cpp:10:13: note: declared here
   10 | std::string add(std::string, std::string) = delete;
      |             ^~~
*/
```

The problem with forbidding template specializations is that it's not scalable. For how many types would you like to forbid the templates? Only for some borderline cases, or for all types that are available. Obviously, it wouldn't scale.

Would you prefer using type traits instead? Maybe that's a bit better, you have less overhead. You don't forbid, but you allow template instantiations. In general, you have lesser things to allow than to forbid.

Yet, it's not so easily readable, it's still a bit verbose and you have to repeat all the assertions if you want the same constraints at other places.

```
#include <iostream>
#include <string>

template <typename T>
T add(T a, T b) {
   static_assert(std::is_integral_v<T> || std::is_floating_point_v<T>,
                 "add cannot be called with strings");
   return a+b;
}

int main() {
 std::cout << add(std::string{"a"}, std::string{"b"}) << '\n';
}

/*
main.cpp: In instantiation of 'T add(T, T)
[with T = std::__cxx11::basic_string<char>]':
main.cpp:11:53:   required from here
main.cpp:6:40: error: static assertion failed: add cannot be called with strings
    6 |    static_assert(std::is_integral_v<T> ||
            std::is_floating_point_v<T>, "add cannot be called with strings");
*/
```

What if you could simply say that you only want to add up integral or floating-point types in the function header. Here come `concepts` into the picture.

With concepts, you can easily express such requirements on template parameters. In addition, using concepts doesn't imply any run-time consts compared to traditional (unconstrained) templates. They

serve as a selection mechanism that happens at compile-time. The generated code is identical to traditional template code.[10]

You can precise requirements on

- the validity of expressions or, in other words, class interfaces
- the return types of certain functions
- the existence of inner types
- the validity of template specializations
- the type-traits of the accepted types

In the following chapters, you will learn exactly how.

Key takeaways

Concepts are one of the four major new features added to C++20. Before the appearance of concepts, limiting the types accepted by templates was verbose and cumbersome if possible at all.

- With concepts, you can model both syntactic and semantic requirements for types, though the semantic parts will mostly come from names and comments.
- Concepts provide a scalable and extremely readable way to validate template arguments at compile-time through boolean predicates without any run-time costs.

In the next chapter, we are going to learn how to use them with functions.

[10] https://www.stroustrup.com/good_concepts.pdf

4 ways to use concepts in functions

In this chapter, we are going to learn the 4 different ways we can use concepts with C++ functions, 4 different ways to constrain function templates. Once we understood them and their differences, we are going to see how to choose among the different forms.

The 4 ways to use concepts

For our examples, let's assume that we have a concept called `number` that requires a built-in arithmetic type of C++. As such, user-defined numeric types are ignored. We are going to use a very simplistic implementation. We'll complete the `number` concept in a later chapter so that only numbers are accepted. At this point, I share an oversimplified implementation:

```
#include <concepts>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;
```

I write it's oversimplified because not only numbers satisfy the `std::integral` concept but also `bool`s, and different `char` types. But it's a good enough approximation for our examples.

The `requires` clause

In the first of the four different ways, we use the `requires` clause between the template parameter list and the function return type - which is `auto` in our example.

```
template <typename T>
requires number<T>
auto add(T a, T b) {
  return a+b;
}
```

Note how we use the concept, how we define our constraint in the `requires` clause expecting that any `T` template parameter must satisfy the requirements of the concept `number`.

In order to determine the return type we simply use `auto` type deduction, but we could use `T` instead as well, though `auto` gives some more flexibility in case of combining narrower types where the result should be promoted.

Using `T` as the return type would cause an integer overflow. On the other hand, with `auto` type deduction, as long as there is a wider type available, you won't face this issue.

It's also worth noting that we can only add up two numbers of the same type. We cannot add a `float` with an `int` using the above template.

If we tried to do so, we'd get a bit long, but quite understandable error message:

```
main.cpp: In function 'int main()':
main.cpp:15:27: error: no matching function for call to 'add(int, float)'
   15 |   std::cout << add(5,42.1f) << '\n';
      |                           ^
main.cpp:10:6: note: candidate: 'template<class T>
            requires  number<T> auto add(T, T)'
   10 | auto add(T a, T b)  {
      |      ^~~
main.cpp:10:6: note:   template argument deduction/substitution failed:
main.cpp:15:27: note:   deduced conflicting types for parameter 'T'
          ('int' and 'float')
   15 |   std::cout << add(5,42.1f) << '\n';
      |                           ^
```

In brief, both `a` and `b` should be of the same type. If we wanted the capability of adding up numbers of multiple types, we'd need to introduce a second template parameter and constrain it similarly.

```
template <typename T,
          typename U>
requires number<T> && number<U>
auto add(T a, U b) {
  return a+b;
}
```

Then calls such as `add(1, 2.14)` would also work. Please note that the constraints were modified. The drawback is that for each new function parameter you'd need to introduce a new template parameter and a requirement on it.

With the `requires` clause, we can also express more complex constraints. For the sake of example, let's just "inline" the definition of `number`:

```
template <typename T>
requires std::integral<T> || std::floating_point<T>
auto add(T a, T b) {
  return a+b;
}
```

Though for better readability, in most cases, I consider it a better practice to name your concept when you have a complex expression.

The trailing `requires` clause

We can also use the so-called *trailing `requires` clause* that comes after the function parameter list (and the qualifiers - `const`, `override`, `noexcept`, etc. - if any) and before the function implementation.

```
template <typename T>
auto add(T a, T b) requires number<T> {
  return a+b;
}
```

We have the same result as we had with the `requires` clause we just wrote it with different syntax. It means that we still cannot add two numbers of different types. We'd need to modify the template definition similarly as we did before:

```
template <typename T, typename U>
auto add(T a, U b) requires number<T> && number<U> {
  return a+b;
}
```

We still have the drawback of scalability. Each new function parameter potentially of a different type needs its own template parameter.

Just as for the `requires` clause, you can express more complex constraints in the *trailing `requires` clause*.

```
template <typename T>
auto add(T a, T b) requires std::integral<T> || std::floating_point<T> {
  return a+b;
}
```

The constrained template parameter

The third way to use a concept is a bit terser than the previous ones, which doesn't only bring readability but some limitations as well.

```
template <number T>
auto add(T a, T b) {
  return a+b;
}
```

As you can see, we don't need any `requires` clause, we can simply define a requirement on our template parameters right where we list them. We use a concept name instead of the keyword `typename`. We achieve the very same result as with the previous two methods.

If you don't believe it, I'd urge you to check it on Compiler Explorer[11].

At the same time, it's worth noting that this method has a limitation. When you use the *`requires` clause* in any of the two presented ways, you can define an expression such as `requires std::integral<T> || std::floating_point<T>`. When you use the *constrained template parameter* way, you cannot have such expressions; `template <std::integral || std::floating_point T>` **is not valid**.

So with this way, you can only use single concepts as constraints, but in a more concise form as with the previous ones.

But can you use a concept that takes multiple parameters? Yes, it's possible.

In case you want to use the concept `std::same_as` which takes two parameters and checks whether two types are the same, you can do it. You have to pass the non-deducible type(s):

```
template<std::same_as<int>, T>
auto foo(T a) {
    // ...
}
```

It's true that with this solution we hardcoded the type that we use with `std::same_as`. We can use other template arguments, given that they have been already declared:

```
template<typename T, std::same_as<T> U>
auto foo(T a) {
    // ...
}
```

Abbreviated function templates

Oh, you looked for brevity? Here you go!

[11] https://godbolt.org/z/sGTsab

```
auto add(number auto a, number auto b) {
  return a+b;
}
```

There is no need for any template parameter list or *`requires` clause* when you opt for *abbreviated function templates*. You can directly use the concept where the function arguments are enumerated.

There is one thing to notice and more to mention.

After the concept `number` we put `auto`. As such we can see that `number` is a constraint on the type, not a type itself. Imagine if you'd simply see `auto add(number a, number b)`. How would you know as a user that `number` is not a type but a concept?

The other consideration I wanted to mention is that when you follow the *abbreviated function template* way, you can mix the types of parameters. You can add an `int` to a `float`.

```
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;

auto add(number auto a, number auto b) {
  return a+b;
}

int main() {
  std::cout << add(1, 2.5) << '\n';
}
/*
3.5
*/
```

With *abbreviated function templates* we can take different types without specifying multiple template parameters. It makes sense as we don't have any template parameters in fact.

The disadvantage of this way of using concepts is that just like with *constrained template parameters*, we cannot use complex expressions to articulate our constraints, and in addition, we can only use concepts that take only one parameter.

But it's remarkably concise.

They are all compiled the same way

As we have four different ways to use concepts with functions, we might ask ourselves the question of whether they are compiled differently? Can there be such a subtle reason behind allowing all

these different syntaxes?

Nothing is simpler than verifying that! Let's start first with going to CppInsights[12] and see what goes on behind the scenes.

```
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;

template <typename T>
requires number<T>
auto addRequiresClause(T a, T b) {
  return a+b;
}

template <typename T>
auto addTrailingRequiresClause(T a, T b) requires number<T> {
  return a+b;
}

template <number T>
auto addConstrainedTemplate(T a, T b) {
  return a+b;
}

auto addAbbreviatedFunctionTemplate(number auto a, number auto b) {
  return a+b;
}

int main() {
    std::cout << "addRequiresClause(1, 2): " << addRequiresClause(1, 2) << '\n';
    std::cout << "addTrailingRequiresClause(1, 2): "
              << addTrailingRequiresClause(1, 2) << '\n';
    std::cout << "addConstrainedTemplate(1, 2): "
              << addConstrainedTemplate(1, 2) << '\n';
    std::cout << "addAbbreviatedFunctionTemplate(1, 2): "
              << addAbbreviatedFunctionTemplate(1, 2) << '\n';
}
```

I don't dump here the full output of CppInsights, you have everything at your hands to try it on your own.

[12] https://cppinsights.io/s/644c2329

The first and third options become literally the same. In case we use requires clause with a simple concept and not with a multi-element expression, we can always use the constrained template parameter form instead as a shorthand notation.

```
template<>
int addRequiresClause<int>(int a, int b)
{
  return a + b;
}

template<>
int addConstrainedTemplate<int>(int a, int b)
{
  return a + b;
}
```

The second option is a little bit different at the first sight:

```
template<>
int addTrailingRequiresClause<int>(int a, int b) requires number<int>
{
  return a + b;
}
```

There is no reason to check if `int` satisfies the concept `number` as it was already checked when it was substituted.

So is there a difference between the two version? Let's go and see it in the Compiler Explorer.

```
mov     esi, OFFSET FLAT:.LC1
mov     edi, OFFSET FLAT:_ZSt4cout
call    std::basic_ostream<char, std::char_traits<char> >&
        std::operator<< <std::char_traits<char> >(
            std::basic_ostream<char, std::char_traits<char> >&, char const*)
mov     rbx, rax
mov     esi, 2
mov     edi, 1
call    auto addTrailingRequiresClause<int>(int, int)
mov     esi, eax
mov     rdi, rbx
call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
mov     esi, 10
mov     rdi, rax
```

```
15 call    std::basic_ostream<char, std::char_traits<char> >&
16         std::operator<< <std::char_traits<char> >(
17             std::basic_ostream<char, std::char_traits<char> >&, char)
```

I included only one version, because apart from the label used on the first line, the generated assembly is identical.

And the assembly is the same even for the fourth option, the abbreviated function templates. Even though the CppInsights output is slightly different.

```
1  template<>
2  int addAbbreviatedFunctionTemplate<int, int>(int a, int b)
3  {
4    return a + b;
5  }
```

The difference is due to the fact that with abbreviated function templates each parameter can be bound to a different type as there is no explicit template parameter list.

You might say that this example is too simplistic. Maybe it is, but it's good enough to show the basic ideas. I let you do the homework and try also to see what happens with more complex examples.

How to choose among the 4 ways?

We have just seen 4 ways to use concepts, let's have a look at them together.

```
1  #include <concepts>
2  #include <iostream>
3
4  template <typename T>
5  concept number = std::integral<T> || std::floating_point<T>;
6
7  template <typename T>
8  requires number<T>
9  auto addRequiresClause(T a, T b) {
10   return a+b;
11 }
12
13 template <typename T>
14 auto addTrailingRequiresClause(T a, T b) requires number<T> {
15   return a+b;
16 }
```

```
17
18 template <number T>
19 auto addConstrainedTemplate(T a, T b) {
20   return a+b;
21 }
22
23 auto addAbbreviatedFunctionTemplate(number auto a, number auto b) {
24   return a+b;
25 }
26
27 int main() {
28     std::cout << "addRequiresClause(1, 2): " << addRequiresClause(1, 2) << '\n';
29     // std::cout << "addRequiresClause(1, 2.5): "
30     //           << addRequiresClause(1, 2.5) << '\n';
31     // error: no matching function for call to 'addRequiresClause(int, double)'
32     std::cout << "addTrailingRequiresClause(1, 2): "
33               << addTrailingRequiresClause(1, 2) << '\n';
34     // std::cout << "addTrailinRequiresClause(1, 2): "
35     //           << addTrailinRequiresClause(1, 2.5) << '\n';
36     // error: no matching function for call to
37     //        'addTrailinRequiresClause(int, double)'
38     std::cout << "addConstrainedTemplate(1, 2): "
39               << addConstrainedTemplate(1, 2) << '\n';
40     // std::cout << "addConstrainedTemplate(1, 2): "
41     //           << addConstrainedTemplate(1, 2.5) << '\n';
42     // error: no matching function for call to 'addConstrainedTemplate(int, double)'
43     std::cout << "addAbbreviatedFunctionTemplate(1, 2): "
44               << addAbbreviatedFunctionTemplate(1, 2) << '\n';
45     std::cout << "addAbbreviatedFunctionTemplate(1, 2): "
46               << addAbbreviatedFunctionTemplate(1, 2.14) << '\n';
47 }
```

Which form should we use? As always, the answer is *it depends*...

If you have a complex requirement, to be able to use an expression you need either the *`requires` clause* or the *trailing `requires` clause*.

What do I mean by a complex requirement? Anything that has more than one concept in it! Like `std::integral<T> || std::floating_point<T>`. That is something you cannot express either with a *constrained template parameter* or with an *abbreviated template function*.

If you still want to use them, you have to extract the complex constraint expressions into their own concept.

This is exactly what we did when we defined the concept `number`. On the other hand, if your concept

uses multiple parameters (something we'll see soon), you still cannot use *abbreviated template functions*.

If you have complex requirements and you don't want to define a named concept, you should go with either of the first two options, namely with *`requires` clause* or with *trailing `requires` clause*.

In case you have a simple requirement with one template parameter, you should go with the *abbreviated function template*. Though we must remember that if your function has multiple parameters constrained with the same concept, then *abbreviated function templates* let you call your function with multiple different types at the same time, like how we called `add` with an `int` and with a `float`. If that is a problem and or you have multiple template parameters, choose *constrained template parameters*.

This recommendation is in accordance with T.13: Prefer the shorthand notation for simple, single-type argument concepts or the Core Guidelines[13].

If you wonder why there are so many different ways, you could probably already guess the answer. Some have more flexibility fo the price of verbosity, why the others are terser but more limited. We have these options available for us, so we can pick the most suitable option.

Conclusion

In this chapter, we have seen how to use concepts with function parameters. There are `4` different ways we examined.

- The biggest flexibility is provided by *`requires` clause* and *trailing `requires` clause*, though they are quite verbose
- The briefest option is to use *abbreviated function templates*, though they cannot handle complex constraints or multiple template parameters
- In the middle, you have *constrained template parameters*, still not supporting complex constraints, but accepting concepts with multiple parameters
- All the options are compiled the same way, what you use should depend on the complexity of your requirement, striving for the most readable option

In the next chapter, we are going to throw away half of these methods to learn how to use concepts with classes.

[13] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t13-prefer-the-shorthand-notation-for-simple-single-type-argument-concepts

C++ concepts with classes

In the previous chapter, we learned to use concepts with function templates and how we are going to do the same for class templates. As we learned there are four ways to use concepts with functions:

- the `requires` clause
- the trailing `requires` clause
- constrained template parameters
- abbreviated function templates

With classes, we cannot use the abbreviated function templates form. It is not a big surprise as even its name tells us that it's for function templates.

We could still give it a try, but as you are going to see, it's not going to compile.

```
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;

class WrappedNumber {
public:
  WrappedNumber(number auto num) : m_num(num) {}
private:
  number auto m_num; // error: non-static data member declared with placeholder
};
```

We cannot declare data members with `auto`, it's prohibited by the standard.[14].

> According to the standard, `auto` cannot be used with non-static members. The rationale behind lays in N2713 - Allow auto for non-static data members[a].
>
> Let's suppose we have such a data structure:
>
> ```
> template<class T>
> struct MyType: T {
>     auto data = func();
>     static const size_t data_size = sizeof(data);
> };
> ```

[14] https://stackoverflow.com/questions/11302981/c11-declaring-non-static-data-members-as-auto

If you want to determine the layout and size of `MyType` we have 2-phase name lookup and argument-dependent lookup (ADL)[b]. `func` might be either a type or a function, it might be found in `T`, in the namespace of `MyType`, in any associated namespace of `T`, in the global namespace, etc. Depending on the order of header inclusions, the results for ADL might even be different and break the One Definition Rule[c].

Due to this controversy, `auto` is not allowed for non-static data members.

[a] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2713
[b] https://en.cppreference.com/w/cpp/language/adl
[c] https://en.wikipedia.org/wiki/One_Definition_Rule

If we remove the `auto`, we'll have a different error message saying that we must use `auto` (or `decltype(auto)`) after the concept `number`. Of course, without adding `auto` how would we know that `number` is a concept and not a type...?

So what is left?

- the `requires` clause
- constrained template parameters
- and the trailing `requires` clause.

The trailing `requires` clause? But what can the `requires` clause follow? No, not the whole class definition, but only functions. We'll see how in a few minutes. Until then, let's focus on the first two options.

For our examples, we are going to use the same incomplete `number` concept for built-in numeric types that we used in the previous chapter.

```
1  #include <concepts>
2
3  template <typename T>
4  concept number = std::integral<T> || std::floating_point<T>;
```

The `requires` clause

We can use the *`requires` clause* to define constraints on a class template. All we have to do is the same as writing a class template and after the template parameter list, we have to put the `requires` clause with all the constraints we'd like to apply on the inputs.

```cpp
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;

template <typename T>
requires number<T>
class WrappedNumber {
public:
  WrappedNumber(T num) : m_num(num) {}
private:
  T  m_num;
};

int main() {
    WrappedNumber wn{42};
    // WrappedNumber ws{"a string"};
    // template constraint failure for
    // 'template<class T>  requires  number<T> class WrappedNumber'
}
```

As you can see in the example, apart from the additional line with the *`requires` clause*, it's the same as a class template.

If you use the template type name `T` at multiple places, the replacing values must be of the same type. In case you take two constrained `T`s in the constructor, they must be of the same type. You won't be able to call it with an `int` and a `float` despite the fact that they both satisfy the concept `number`.

In case you need to support different types, for each - potentially different - usage of the template parameter, you need a different declaration in the template parameter list and also among the constraints:

```cpp
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;


template <typename T, typename U>
requires number<T> && number<U>
class WrappedNumber {
public:
```

```
  WrappedNumber(T num, U anotherNum) : m_num(num), m_anotherNum(anotherNum) {}
private:
  T  m_num;
  U  m_anotherNum;
};

int main() {
    WrappedNumber wn{42, 4.2f};
}
```

The above example also shows that we can use compound expressions as constraints. That's something not possible the other way, with constrained template parameters.

Constrained template parameters

With *constrained template parameters* it's even easier to use concepts and constrain our template types. In the template parameter list, instead of the `typename` keyword, you can simply write the concept you want to your template arguments to model.

Here is an exmaple:

```
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;


template <number T>
class WrappedNumber {
public:
  WrappedNumber(T num) : m_num(num) {}
private:
  T  m_num;
};

int main() {
    WrappedNumber wn{42};
    // WrappedNumber ws{"a string"};
    // template constraint failure for
    // 'template<class T>  requires  number<T> class WrappedNumber'
}
```

In this example, you can see how we constrained `T` to satisfy the `number` concept.

The clear advantage of *constrained template parameters* is that they are so easy to write, they are so easy to read and there is no extra verbosity.

The downside is that you cannot use compound expressions as constraints.

While with the `requires` clause you can write something like this:

```
template <typename T>
requires std::integral<T> || std::floating_point<T>
class WrappedNumber {
  // ...
};
```

With the *constrained template parameters* something similar is impossible. If you have to use some complex constraints, you must extract them into their own named concept.

Similarly to the `requires` clause, in case you have multiple parameters that need to satisfy `number`, but with different types at the same time, you must use multiple template parameters:

```
template <number T, number U>
class WrappedNumber {
public:
  WrappedNumber(T num, U anotherNum) : m_num(num), m_anotherNum(anotherNum) {}
private:
  T  m_num;
  U  m_anotherNum;
};
```

In case you want to use a concept that takes more than one parameter, it's completely possible.

```
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;

template <number T, std::same_as<T> U>
class WrappedNumbers {
public:
  WrappedNumbers(T num, U anotherNum) : m_num(num), m_anotherNum(anotherNum) {}
private:
  T  m_num;
```

```
  U  m_anotherNum;
};

int main() {
    WrappedNumbers wns{5, 6};

    // main.cpp: In function 'int main()':
    // main.cpp:18:31: error: class template argument deduction failed:
    // 18 |     WrappedNumbers wns2{5, 6.2};
    //  |                               ^
    // main.cpp:18:31: error: no matching function for call to
    //   'WrappedNumbers(int, double)'
    // main.cpp:10:3: note: candidate: 'template<class T, class U>
    //    WrappedNumbers(T, U)-> WrappedNumbers<T, U>'
    // 10 |   WrappedNumbers(T num, U anotherNum) :
    //          m_num(num), m_anotherNum(anotherNum) {}
    // WrappedNumbers wns2{5, 6.2};
}
```

Note how the declaration of `wns2` fails with two different types after we constrained `U` to be the same as `T` by using `std::same_as<T>` in the template argument list.

The trailing `requires` clause

When you read that we can use the *trailing `requires` clause* with classes templates, you might have had something similar in mind:

```
template <typename T>
class WrappedNumber {
public:
  WrappedNumber(T num) : m_num(num) {}
private:
  T  m_num;
} requires number<T>;
```

This wouldn't have any benefits. Imagine that you have to scroll down to the end of some longer classes just to see these constraints... At least it doesn't even compile.

Yet, we can still use the *trailing `requires` clause* with class templates. On functions. Yes, on functions.

So far we have seen examples of constraining the template parameter for the whole class. But what if... what if you wanted to constrain the usage of a certain function of a class template? Or what if you wanted to provide different implementations based on the characteristics, let's say based on

the traits of the template parameter type? It's possible without using the difficult to pronounce and difficult to understand, write-only SFINAE.

E.g., it's possible to restrict the usage of a function based on the type of the class template parameter. In the following example, our `Ignition` class has a function called `insertKey()`. But not all modern cars have keys to insert anymore. In fact, there are fewer and fewer of them and calling `insertKey()` with so-called *smart keys* does not make much sense.

Therefore we'd like to find a way to ban the `insertKey()` function for keys that are considered smart. There are different ways to do that, here we are going to focus on the one with concepts. We already saw in the first chapter on the motivations for concepts why the others methods are less efficient.

We can restrict function specializations with `= delete` for whatever types satisfying certain concepts. In our example, we cannot call the `insertKey` function with any type satisfying the `smart` concept, while the rest of the types are not affected.

```
template <typename Key>
class Ignition {
public:
  void insertKey(Key key) {
    // ...
  };

  void insertKey(Key key) requires smart<Key> = delete;

};
```

As we can observe, to achieve this we needed to

- create a class template and
- use the trailing `requires` clause after a function that we wanted to `delete`

This means that we can apply constraints only on specific methods of a class template, while in other methods the template parameters can be used unconstrained.

As you might have guessed, this way we cannot only ban function calls, but easily provide special implementations based on different type constraints.

Let's say that we have a class wrapping a number. Depending on what kind of type it wraps, the implementation of certain functions might differ.

```cpp
template <typename T>
class WrappedNumber {
 public:
  T foo() requires std::floating_point<T> {
    // ...
  }

  T foo() requires std::integral<T> {
    // ...
  }

  T foo() requires std::integral<T> && std::is_signed_v<T> {
    // ...
  }

  // ...
};
```

In the above example `foo()` has 3 different implementations based on whether `T` is an integral or a floating-point number, and we even provided a specialization for signed integrals.

We could do this by using the same signatures overloading simply the `requires` clause. Note that it's also possible to give a general implementation without specifying any constraint, thus without a `requires` clause.

If you define multiple implementations for a method with concepts, it's a good practice to provide a general implementation. When you look at the code, you don't have to think which is the least constrained one, but it clearly strikes out. If you don't want to allow a general implementation, you can still delete it.

Let's see a concrete example with the class `Ignition`.

```cpp
template <typename Key>
class Ignition {
public:
  void start(Key key) {
    // ...
  }

  void start(Key key) requires smart<Key> {
    // ...
  }

  void start(Key key) requires smart<Key> && personal<Key> {
```

```
13      // ...
14    }
15  };
```

The generic implementation of `Ignition::start()` has no constraints. We could say that it requires that the `Key` must not satisfy the concept `smart` (`void start(Key key) requires (!smart<Key>) {/*...*/}`), but it would actually decrease the readability.

When you start using constrained methods in class templates, first it might seem like a good idea to use so-called *complimentary constraints*:

```
1  template <typename Key>
2  class Ignition {
3  public:
4    void start(Key key) requires !smart<Key> {
5      // ...
6    }
7
8    void start(Key key) requires smart<Key> {
9      // ...
10   }
11 };
```

As we saw, negations don't always reflect our intentions, reading the code with this complementary constraint became more difficult, plus it might become worse if you later add different constrained overloads.

It's better to simply have a general implementation instead, exactly like the Core Guidelines recommends in T.25: Avoid complementary constraints[a].

```
1  template <typename Key>
2  class Ignition {
3  public:
4    void start(Key key) {
5      // ...
6    }
7
8    void start(Key key) requires smart<Key> {
9      // ...
10   }
11 };
```

[a] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t25-avoid-complementary-constraints

Then we define two constrained overloads of the method `start()`. One requires that the `Key` template

type satisfies the `smart` concept and the other satisfies in addition to the `smart` concepts also the `Personal` one.

The compiler will be able to choose for each template type the most constrained implementation.

By providing a generic implementation you make sure that in all cases there is something to fall back to and if it's deleted, you also make that very explicit. By having an unconstrained implementation you also make it extremely readable what is the least constrained method.

It's a good practice to list the implementations going from the least constrained towards the most constrained ones if that's possible and the different implementations are not orthogonal.

There are a couple of rules to keep in mind though, but we'll discuss them in a coming chapter.

Conclusion

In this chapter, we discovered three ways to use concepts with classes. Both with *the `requires` clause* and with *constrained template parameters* we have an easy and readable way to use our concepts to constrain the types our template classes can accept. In addition, we saw how to use the *trailing `requires` clause* to constrain only certain methods of a class template.

- With the *`requires` clause*, we can even define some complex requirements without having to extract them into separate concepts
- While with the *constrained template parameters* we can only use one concept per template parameter, but on the contrary, it's very terse. Up to you to choose based on your needs.
- We can also use the *the `requires` clause* when we want to provide an overload only to certain methods of the class template based on constraints applied on the template arguments of the class template.
- In case the only thing you need is a simple parameter concept and you have no additional constraints to apply, you can take advantage of the extremely terse syntax of *abbreviated function templates*.

In the next chpter, we are going to learn what kind of concepts we get from the standard library.

Concepts shipped with the C++ standard library

In this chapter, we are going to have an overview of the different kinds of concepts that are shipped with the C++ standard library. Don't worry, we are not going to have a complete enumeration, rather we look at a few to understand the concepts behind the standard concepts.

Besides the ability to write powerful concepts, C++20 also includes more than 50 concepts in the standard library. They are shared across three different headers:

- `<concepts>`
- `<iterator>`
- `<ranges>`

Concepts in the `<concepts>` header

In the `<concepts>` header you will find the most generic ones expressing core language, comparison and object concepts. Many of them can be familiar to you from the `<type_traits>` header, though the concepts are definitely less numerous.

We are not going to explore all of them here for obvious reasons, you can find the full list here[15]. Let me just pick three concepts so that we can get the idea.

`std::convertible_to` for conversions with fewer surprises

`std::convertible_to` helps you to express that you only accept types that are convertible to another specific type. The conversion can be both explicit or implicit. For example, you can say that you only accept types that can be converted into a `bool`. As the first parameter, you pass the type you want a conversion to be valid `From` and as the second, the type you want to be able to convert `To`, in our case, `bool`.

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

```cpp
#include <concepts>
#include <iostream>
#include <string>

template <typename T>
void fun(T bar) requires std::convertible_to<T, bool> {
  std::cout << std::boolalpha << static_cast<bool>(bar) << '\n';
}

int main() {
 fun(5); // OK an int can be converted into a boolean
//  fun(std::string("Not OK"));
//  void fun(T) requires  convertible_to<T, bool>
// [with T = std::__cxx11::basic_string<char>]' with unsatisfied constraints
}
```

In this example, we have successfully tested that an `int` can be converted into a `bool`, but a `std::string` has no valid conversions into an `int`.

`std::totally_ordered` for defined comparisons

`std::totally_ordered` helps to accept types that specify all the 6 comparison operators (==,!=,<,>,<=,>=) and that the results are consistent with a strict total order[16] on T.

> The strict total order of items in math means that having `a`, `b` and `c` coming from the same set, all satisfy the following requirements:
>
> ```
> Not a < a // Irreflexive
> if a < b and b < c then a < c // Transitive
> if a != b then a < b or b < a // Connected
> ```

[16] https://en.wikipedia.org/wiki/Total_order#Strict_total_order

```cpp
#include <concepts>
#include <iostream>
#include <typeinfo>

struct NonComparable {
  int a;
};

struct Comparable {
  auto operator<=>(const Comparable& rhs) const = default;
  int a;
};


template <typename T>
void fun(T t) requires std::totally_ordered<T> {
  std::cout << typeid(t).name() << " can be ordered\n";
}

int main() {
  NonComparable nc{666};
//   fun(nc);
// Not OK: error: use of function 'void fun(T) requires  totally_ordered<T>
// [with T = NonComparable]' with unsatisfied constraints
  Comparable c{42};
  fun(c);
}
```

A type without the comparison operators defined is not totally ordered, while a type that defines those operators is totally ordered. A type that is totally ordered must have all of its members totally ordered. This means that if we have a type that defines all the comparison operators but it has a member that doesn't then the type is not totally ordered.

> In the above example, you can also observe how to easily use the three-way comparison operator (`<=>`) (a.k.a. the spaceship operator).
>
> If you ever implemented comparison operators in C++, you know what a menial, tedious task it is. You have to define six operators (`==`, `!=`, `<`, `<=`, `>`, `>=`).
>
> With C++20, you can simply `=default` the spaceship operator and it will generate all the six operators for you with `constexpr` and `noexcept`. These operators will perform a lexicographical comparison.

```cpp
class WrappedInt {
```

```
  public:
     constexpr WrappedInt(int value): m_value{value} { }
     auto operator<=>(const WrappedInt& rhs) const = default;
  private:
     int m_value;
};
```

If you are looking for more information on the <=> operator, I highly recommend reading this article from Modernes C++[a].

[a] https://www.modernescpp.com/index.php/c-20-more-details-to-the-spaceship-operator

`std::copyable` for copyable types

`std::copyable` helps you to ensure that only such types are accepted whose instances can be copied. `std::copyable` object must be copy constructible, assignable and movable.

```
#include <concepts>
#include <iostream>
#include <typeinfo>

class NonMovable {
public:
  NonMovable() = default;
  ~NonMovable() = default;

  NonMovable(const NonMovable&) = default;
  NonMovable& operator=(const NonMovable&) = default;

  NonMovable(NonMovable&&) = delete;
  NonMovable& operator=(NonMovable&&) = delete;
};

class NonCopyable {
public:
  NonCopyable() = default;
  ~NonCopyable() = default;

  NonCopyable(const NonCopyable&) = delete;
  NonCopyable& operator=(const NonCopyable&) = delete;

```

```
25    NonCopyable(NonCopyable&&) = default;
26    NonCopyable& operator=(NonCopyable&&) = default;
27  };
28
29  class Copyable {
30  public:
31    Copyable() = default;
32    ~Copyable() = default;
33
34    Copyable(const Copyable&) = default;
35    Copyable& operator=(const Copyable&) = default;
36
37    Copyable(Copyable&&) = default;
38    Copyable& operator=(Copyable&&) = default;
39  };
40
41  template <typename T>
42  void fun(T t) requires std::copyable<T> {
43    std::cout << typeid(t).name() << " is copyable\n";
44  }
45
46  int main() {
47    NonMovable nm;
48  //  fun(nm);
49  //  error: use of function 'void fun(T) requires  copyable<T>
50  //    [with T = NonMovable]' with unsatisfied constraints
51    NonCopyable nc;
52  //  fun(nc);
53  //  error: use of function 'void fun(T) requires  copyable<T>
54  //    [with T = NonCopyable]' with unsatisfied constraints
55    Copyable c;
56    fun(c);
57  }
```

As you can see in the above example, class `NonMovable` doesn't satisfy the concept as its move assignment and move constructor are deleted.

For `NonCopyable`, it's a similar case, but while the move semantics are available, it lacks the copy assignment and the copy constructor.

Finally, `Copyable` class defaults all the 5 special member functions and as such, it satisfies the concept of `std::copyable`.

It's worth reminding ourselves what are the special functions are in C++ and what is special about them.

Since C++11, there are one plus five special functions in C++ that are generated by the compiler under certain conditions:
- a default constructor (`T()`), that calls the default constructor of each class member and base class. It doesn't affect the rest of the special member functions.

- a copy constructor (`T(const T&)`), that calls the copy constructor on each member and base class
- a copy assignment operator (`T& operator=(const T&)`), that calls a copy assignment operator on each class member and base class
- a move constructor (`T(X&&)`), that calls the move constructor on each class member and base class
- a move assignment (T& operator=(T&&)), that calls the move constructor on each member and base class
- the destructor (`~T()`), that calls the destructor of each class member and base class. Note that this default-generated destructor is never virtual (unless it is for a class inheriting from one that has a virtual destructor).

If there is no constructor specified, the default one is automatically generated.

The other five special functions are also automatically generated unless at least one is implemented manually. If you implement one, you should think about implementing the others.

That's what the rule of 5 is about. If you implement by hand either the copy or move constructor, the copy or move assignment operator or the destructor, the compiler will not generate those that you didn't write. So if you need to write any of these give, the assumption is that you'll need the rest as well.

Concepts in the `<iterator>` header

In the `<iterator>`[17] header, you'll mostly find concepts that will come in handy when you deal with algorithms[18]. It makes sense if you think about it, as the functions of the `<algorithm>` header operate on the containers through iterators, not directly on the containers.

`std::indirect_unary_predicate<F, I>`

There are concepts related to callables, e.g. you can specify that you accept only unary predicates. First, what is a predicate? A predicate is a callable that returns either a `bool` value or something that is convertible to `bool`. A unary predicate is a predicate that takes one parameter as its input.

[17] https://en.cppreference.com/w/cpp/iterator#Algorithm_concepts_and_utilities
[18] https://www.sandordargo.com/blog/2019/01/30/stl-algos-intro

I know that the following example is not very realistic, it's only for demonstrational purposes.

```
#include <iostream>
#include <iterator>
#include <vector>

template <typename F, typename I>
void foo(F fun, I iterator) requires std::indirect_unary_predicate<F, I> {
    std::cout << std::boolalpha << fun(*iterator) << '\n';
}

int main()
{
  auto biggerThan42 = [](int i){return i > 42;};
  std::vector numbers{15, 43, 66};
  for(auto it = numbers.begin(); it != numbers.end(); ++it) {
      foo(biggerThan42, it);
  }
}
```

In the above example `foo()` takes a function and an iterator and the concept `std::indirect_unary_predicate` ensures that the passed-in function can take the value pointed by the iterator and that a `bool` is returned.

`std::indirectly_comparable`

In the `<iterator>`[19] header you'll not only find concepts related to callables but more generic ones as well. Such as whether two types are indirectly comparable. That sounds interesting, let's have a look at a simple example:

```
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

template <typename Il, typename Ir, typename F>
void foo(Il leftIterator, Ir rightIterator, F function)
requires std::indirectly_comparable<Il, Ir, F> {
    std::cout << std::boolalpha << function(*leftIterator, *rightIterator) << '\n';
}

```

[19] https://en.cppreference.com/w/cpp/iterator#Algorithm_concepts_and_utilities

```
12 int main()
13 {
14   using namespace std::string_literals;
15
16   auto binaryLambda = [](int i, int j){ return 42; };
17   auto binaryLambda2 = [](int i, std::string j){return 666;};
18
19   std::vector ints{15, 42, 66};
20   std::vector floats{15.1, 42.3, 66.6};
21   foo(ints.begin(), floats.begin(), binaryLambda);
22 //   foo(ints.begin(), floats.begin(), binaryLambda2);
23 //   error: use of function 'void foo(Il, Ir, F) requires
24 //      indirectly_comparable<Il, Ir, F, std::identity, std::identity>
25 }
```

In this case, I've been left a bit puzzled by the documentation[20]:

- As a third template parameter it has `class R` which normally would refer to ranges.
- But then according to its definition, it calls `std::indirect_binary_predicate` with `R` forwarded in the first position.
- In `std::indirect_binary_predicate`[21], as a first argument `class F` is taken and `F` stands for a callable (often a function).

Why isn't `R` called `F`? Why binary predicates are not mentioned in the textual description?

Probably only because this is still the beginning of the concepts journey and the documentation is still to be improved.

Concepts in the `<ranges>` header

In the `<ranges>`[22] header you'll find concepts describing constraints for different types of ranges.

Or simply that a parameter is a `range`. But you can assert for any kind of ranges, like `input_range`, `output_range`, `forward_range`, etc.

Let's have a look at a bit more exotic range, `std::ranges::borrowed_range`.

[20] https://en.cppreference.com/w/cpp/iterator/indirectly_comparable
[21] https://en.cppreference.com/w/cpp/iterator/indirect_binary_predicate
[22] https://en.cppreference.com/w/cpp/ranges#Range_concepts

```cpp
#include <iostream>
#include <ranges>
#include <string>
#include <vector>
#include <typeinfo>

template <typename R>
void foo(R range) requires std::ranges::borrowed_range<R> {
  std::cout << typeid(range).name() << " is a borrowed range\n";
}

int main()
{
  std::vector numbers{15, 43, 66};
  std::string_view stringView{"is this borrowed?"};
//  foo(numbers);
//  error: use of function 'void foo(R) requires  borrowed_range<R>
//    [with R = std::vector<int, std::allocator<int> >]' with unsatisfied constraints
  foo(stringView);
}
```

The above example checks whether a type satisfies the concept of a `borrowed_range`. We can observe that a `std::string_view` does, while a `vector` doesn't.

If you are curious, having a borrowed range means that a function can take it by value and can return an iterator obtained from it without any dangers of dangling. For more details, click here[23].

As a `string_view` is not an owning type, it's not a problem that returned iterator is taken from a value type. On the other hand, a `vector` owns its items. In case, a `vector` is taken by value any returned iterator would reference a value that went out of scope

[23] https://en.cppreference.com/w/cpp/ranges/borrowed_range

Conclusion

In this chapter, we've seen a few examples of concepts shipped with the C++20 standard library. There are about 50 standard concepts shared among 3 headers:

- In `<concepts>` we can find generic concepts expressing core language features, comparison and object requirements.
- In `<iterators>` we can find concepts that model different kinds of iterators and concepts that come in handy with algorithms.
- Finally, in `<ranges>` we can find concepts modelling different kinds of ranges that are another major feature of C++20.

It's worth having a look at these concepts before we start implementing our own concepts to understand what kind of concepts we have at our disposal, which - by the way - we are going to learn about in the next chapter.

How to write your own C++ concepts?

Welcome to the longest chapter of this book. We already discussed the motivations behind the conception of C++ concepts and we already saw how to use them with functions and with classes. We've also seen a few that are part of the standard library. But we have hardly written any. We defined a functionally incomplete `concept` called `number` for the sake of example, but that's it. In the following pages, we are going into details on what kind of constraints we can express in a concept and how to write them.

The simplest `concept`

In P0557r1 - Concepts: The Future of Generic Programming[24], Bjarne Stroustrup said that we can think about `auto` as "the least constrained concept", a concept that accepts just any type.

Let's define now that simplest, least constrained concept. Obviously, we cannot name it `auto`, that name is already taken, so let's call it `any`.

```
1 template <typename T>
2 concept any = true;
```

First, we list the template parameters, in this case, we only have one, `T`, but we could have multiple ones separated by commas. Then after the keyword `concept`, we declare the name of the concept and then after the `=`, we define the concept by its constraints.

In this example, we simply set `true` as the sole constraint, meaning that for any type `T` the concept will be evaluated to `true`; just any type is accepted. Should we wrote `false`, nothing would be accepted.

Now that we saw the simplest concept, let's check what building blocks are at our disposal to construct a more detailed concept.

Use already defined concepts

Arguably the easiest way to define new concepts is by combining existing ones.

For instance, in the next example, we are going to create - once again - a concept called `number` accepting both integers and floating-point numbers.

[24] https://www.stroustrup.com/good_concepts.pdf

```cpp
#include <concepts>

template<typename T>
concept number = std::integral<T> || std::floating_point<T>;
```

As you can see in the above example, we could easily combine the two concepts with the `||` operator. We can also use `&&` and `!` operators. Though the negation might not mean exactly what you'd expect. More about that soon.

Probably it's self-evident, but we can combine user-defined concepts as well.

```cpp
#include <concepts>

template<typename T>
concept integer = std::integral<T>;

template<typename T>
concept float = std::floating_point<T>;

template<typename T>
concept number = integer<T> || float<T>;
```

In this example, we basically just aliased (and added a layer of indirection to) `std::integral` and `std::floating_point` to show that user-defined concepts can also be used in a combination of concepts.

As we saw in the previous chapter, there are plenty of concepts defined in the different headers of the standard library so there is an endless way to combine them.

But what are the restrictions for combining them?

`!a` is not the opposite of `a`

The opposite of `true` expression is not an expression evaluated to `false` when we talk about concepts.

Let's suppose we have a function `foo()` that takes two parameters, `T bar` and `U baz`. We have some constraints on them. One of them must have a nested type `Blah` that is unsigned.

```cpp
#include <concepts>

template <typename T, typename U>
requires std::unsigned_integral<typename T::Blah> ||
  std::unsigned_integral<typename U::Blah>
void foo(T bar, U baz) {
    // ...
}

class MyType {
public:
    using Blah = unsigned int;
    // ...
};

int main() {
    MyType mt;
    foo(mt, 5);
    foo(5, mt);
    // error: no operand of the disjunction is satisfied
    // foo(5, 3);
}
```

When we call `foo()` with an instance of `MyType` in the first position, the requirements are satisfied by the first part of the disjunction and the second one is shortcircuited. All seems expected, though we could have already noticed something...

If you haven't noticed anything particular, don't worry. Let's go for the second case. We call `foo()` with an integer in the first place. Is its nested type `Blah` unsigned? It doesn't even have a nested type! Com'on, it's just an `int`!

What does this mean for us? It means that having something evaluated not to `true` doesn't require that that an expression returns `false`. It can simply not be compilable at all.

Whereas for a normal boolean expression, we expect that it's well-formed and each subexpression is compilable.

That's the big difference.

For concepts, the opposite of a `true` expression is not `false`, but something that is either not well-formed, or `false`!

Negations come with parentheses

In the `requires` clause sometimes we wrap everything in between parentheses, sometimes we don't have to do so.

It depends on the simplicity of the expression. What is considered simple enough so that no parentheses are required?

- `bool` literals
- `bool` variables in any forms among `value`, `value<T>`, `T::value`, `trait<T>::value`
- concepts, such as `Concept<T>`
- nested requires expressions
- conjunctions (`&&`)
- disjunctions (`||`)

This list implies that negations cannot be used without parentheses.

Try to compile this function:

```
template <typename T>
requires !std::integral<T>
T add(T a, T b) {
   return a+b;
}
```

It will throw at you a similar error message:

```
main.cpp:8:10: error: expression must be enclosed in parentheses
    8 | requires !std::integral<T>
```

Why is this important? And should we avoid not simple enough constraints that need parentheses?

Subsumption and negations

All these matters, when there are multiple versions of the same method differentiated only by their constraints. Then the compiler has to look for the most constrained version.

Let's assume that we have a class `Ignition` with two versions of `start` method:

```
template <typename Key>
class Ignition {
public:
  void start(Key key) requires (!smart<Key>) {}

  void start(Key key) requires (!smart<Key>) && personal<Key> {}
};
```

The compiler uses boolean algebra to find the most constrained version of `start()` to take. If you want to learn more about the theories behind this process that is called subsumption, I'd also recommend you to read about syllogism[25].

If we called `Ignition` with a `Key` that is not `smart`, you expect the compiler to subsume that the first constraints of the two overloads of `Ignition::start()` are common and we have to check whether the second one applies to our `Key` type or not.

It seems simple.

It's not so simple.

If you try to call `Ignition::start()` with a `Key` that is not `smart` and compile the code, you'll get an error message complaining about an ambiguous overload.

```
main.cpp: In function 'int main()':
main.cpp:25:23: error: call of overloaded 'start(MyKey&)' is ambiguous
   25 |     ignition.start(key);
```

Even though we used the parentheses!

The problem is that `()` is part of the expression and when it comes to subsumption, the compiler checks the source location of the expression. Only if two expressions are originating from the same place, they are considered the same, so the compiler can subsume them.

As `()` is part of the expression, `(!smart<Key>)` originates from two different points and those 2 are not considered the same, they cannot be subsumed. In fact, only with the `!` without the enclosing `()` we would face the same problem.

They are considered 2 different constraints, hence the call to `start` is ambiguous.

That's why if you need negation and thus you need parentheses, and you rely on subsumption rules, it's better to put those negated expressions into their own concepts.

[25] https://en.wikipedia.org/wiki/Syllogism

```
template <typename T>
concept not_smart = !smart<T>;

template <typename Key>
class Ignition {
public:
  void start(Key key) requires not_smart<Key> {}

  void start(Key key) requires not_smart<Key> && personal<Key> {}
};
```

`not_smart<T>` has the same source location whenever it is used, therefore it can be subsumed.

Not using negations directly, but making those expressions part of concepts seems to go against the rule of using standard concepts whenever possible instead of writing our own concepts[26]. In the above example we used an *application concept*, but even if we used something like `std::floating_point`, we'd need to create another concept called `not_floating_point` - or similar. Yet, due to the subsumption rules, this is necessary.

Requirements on operations

Now that we saw some important rules, let's continue our quest to write our own concepts.

We can simply express that we require that a template parameter support a certain operation or operator by *wishful writing*. What is *wishful writing*? Just have to write down what you wish if it was compiled.

If you require that template parameters are addable you can create a concept for that:

```
#include <iostream>
#include <concepts>

template <typename T>
concept addable = requires (T a, T b) {
  a + b;
};

template <addable T>
auto add(T x, T y) {
  return x + y;
}

```

[26] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t11-whenever-possible-use-standard-concepts

```
14 struct WrappedInt {
15   int m_int;
16 };
17
18 int main () {
19   std::cout << add(4, 5) << '\n';
20   std::cout << add(true, true) << '\n';
21   // In substitution of 'template<class T>  requires
22   // addable<T> auto add(T, T) [with T = WrappedInt]':
23   //   required from here
24   //   required for the satisfaction of 'addable<T>'
25   //   [with T = WrappedInt]
26   //   in requirements with 'T a', 'T b' [with T = WrappedInt]
27   //   the required expression '(a + b)' is invalid
28 }
29 /*
30 9
31 2
32 */
```

We can observe that when `add()` is called with parameters of type `WrappedInt` - as they do not support `operator+` - the compilation fails with a rather descriptive error message (though not the whole error message is copied over into the above example).

Writing the `addable` concept seems rather easy, right? After the `requires` keyword we basically wrote down what kind of syntax we expect to compile and run.

Simple requirements on the interface

Let's think about operations for a little longer. What does it mean after all to require the support of a + operation?

It means that we constrain the accepted types to those having a function `T T::operator+(const T& other) const` function. Or it can even be `T T::operator+(const U& other) const`, as maybe we want to add to an instance of another type, but that's not the point here. My point is that we made a requirement on having a specific function.

So we should be able to define a requirement on any function call, shouldn't we?

Right, let's see how to do it.

```cpp
#include <iostream>
#include <string>
#include <concepts>

template <typename T> // 2
concept has_square = requires (T t) {
    t.square();
};

class IntWithoutSquare {
public:
  IntWithoutSquare(int num) : m_num(num) {}
private:
  int m_num;
};

class IntWithSquare {
public:
  IntWithSquare(int num) : m_num(num) {}
  int square() {
    return m_num * m_num;
  }
private:
  int m_num;
};

void printSquare(has_square auto number) { // 1
  std::cout << number.square() << '\n';
}

int main() {
  // printSquare(IntWithoutSquare{4});
  // error: use of function 'void printSquare(auto:11)
  //        [with auto:11 = IntWithoutSquare]' with unsatisfied constraints,
  //        the required expression 't.square()' is invalid
  printSquare(IntWithSquare{5});
}
```

In this example, we have a function `printSquare` (1) that requires a parameter satisfying the concept `has_square` (2). In that concept, we can see that it's really easy to define what interface we expect. After the `requires` keyword, we have to write down how what calls should be supported by the interface of the accepted types.

Our expectations are written after the `requires` keyword. First, there is a parameter list between

parentheses - like for a function - where we have to list all the template parameters that would be constrained and any other parameters that might appear in the constraints. More on that later.

If we expect that any passed in type have a function called `square`, we simply have to write `(T t){t.square();}`. `(T t)` because we want to define a constraint on an instance of `T` template type and `t.square()` because we expect that `t` instance of type `T` must have a public function `square()`.

If we have requirements on the validity of multiple function calls, we just have to list all of them separated by a semicolon like if we called them one after the other:

```
template <typename T>
concept has_square = requires (T t) {
  t.square();
  t.sqrt();
};
```

What about parameters? Let's define a `power` function that takes an `int` parameter for the exponent:

```
template <typename T>
concept has_power = requires (T t, int exponent) {
    t.power(exponent);
};

// ...

void printPower(has_power auto number) {
  std::cout << number.power(3) << '\n';
}
```

The `exponent` variable that we pass to the `T::power()` function has to be listed after the `requires` keyword with its type, along with the template type(s) we constrain. As such, we fix that the parameter will be something that is (convertible to) an `int`.

But what if we wanted to accept just any integral number as an exponent? Where is a will, there is a way! Well, it's not always true when it comes to syntactical questions, but we got lucky in this case.

First, our concept `has_power` should take two parameters. One for the base type and one for the exponent type.

```
template <typename Base, typename Exponent>
concept has_power = std::integral<Exponent> && requires (Base base, Exponent exponen\
t) {
    base.power(exponent);
};
```

We make sure that template type `Exponent` is an integral and that it can be passed to `Base::power()` as a parameter.

The next step is to update our `printPower()` function. The concept `has_power` has changed, now it takes two types, we have to make some changes accordingly:

```
template<typename Exponent>
void printPower(has_power<Exponent> auto number, Exponent exponent) {
  std::cout << number.power(exponent) << '\n';
}
```

As `Exponent` is explicitly listed as a template type parameter, there is no need for the `auto` keyword after it. On the other hand, `auto` is needed after `has_power`, otherwise, how would we know that it's a concept and not a specific type?! As `Exponent` is passed as a template type parameter to `has_power` constraints are applied to it too (as specified in `has_power`).

Now `printPower` can be called the following way - given that we renamed `IntWithSquare` to `IntWithPower` following our API changes:

```
printPower(IntWithPower{5}, 3);
printPower(IntWithPower{5}, 4L);
```

At the same time, the call `printPower(IntWithPower{5}, 3.0);` will fail because the type `float` does not satisfy the constraint on integrality.

Let's put that aside for a minute.

Do we miss something else? Yes! We can't use `IntWithPower` as an exponent. We want to be able to call `Base::power(Exponent exp)` with a custom type, like `IntWithPower` and for that, we need two things:

- `IntWithPower` should be considered an `integral` type
- `IntWithPower` should be convertible to something accepted by `pow` from the `cmath` header.

Let's go one by one.

By explicitly specifying the `type_trait std::is_integral` for `IntWithPower`, we can make `IntWithPower` an integral type. Or... Let's stop for a second. The following piece of code would compile:

```cpp
template<>
struct std::is_integral<IntWithPower> : public std::integral_constant<bool, true> {};
```

At the same time, sadly but rightly, it would lead to *undefined behaviour*. Specializing most of the type traits results in *undefined behaviour* according to the standard [meta.type.synop (1)]:

> *"The behavior of a program that adds specializations for any of the templates defined in this subclause is undefined unless otherwise specified."*

What is in that subsection? You can check it in a draft standard if you don't have access to the paid version. The list is very long and `std::is_integral` is part of it. In fact, all the primary or composite type categories are in there.

Why?

As Howard Hinnant, the father of <chrono> explained on StackOverflow[27] "for any given type T, exactly one of the primary type categories has a value member that evaluates to `true`." If a type satisfies `std::is_integral` then we can safely assume that `std::is_class` will evaluate to `false`. As soon as we are allowed to add specializations, we cannot rely on this assumption.

```cpp
#include <type_traits>

class MyInt {};

template<>
struct std::is_integral<MyInt> : public std::integral_constant<bool, true> {};

int main() {
    static_assert(std::is_integral<MyInt>::value, "MyInt is not integral type");
    static_assert(std::is_class<MyInt>::value, "MyInt is not class type");
}
```

In the above example, `MyInt` breaks the explained assumption and this is *undefined behaviour*.

The above example shows us another reason why not to rely on such techniques. Developers cannot be trusted that much. We either made a mistake or simply lied by making `MyInt` an integral type as it doesn't behave at all like an integral.

This means that you cannot make your type satisfy a type trait in most cases. (As mentioned, the traits that are not allowed to be specialized are listed in the C++ standard).

In spite of this, we have two other solutions:

- define a concept `number` that doesn't simply constrain types based on traits, but that models how numeric types behave and use that to constrain `Exponent`

[27] https://stackoverflow.com/a/25345871/3238101

- make a constraint for `Exponent` in a way that it must be convertible to a number, and as we've had `MyInt` in our mind, let's say to an `int`.

The first solution is beyond the scope of this chapter, so let's go with the second one. The standard library gives us `std::convertible_to<From, To>` that does exactly what we need.

Let's update the concept `has_power`:

```
template <typename Base, typename Exponent>
concept has_power = std::convertible_to<Exponent, int>
                    && requires (Base base, Exponent exponent) {
    base.power(exponent);
};
```

What is left is to make sure that `IntWithPower` is convertible into a type that is accepted by pow[28]. It accepts floating-point types, but when it comes to `IntWithPower`, in my opinion, it's more meaningful to convert it to an `int` and let the compiler perform the implicit conversion to `float` - even though it's better to avoid implicit conversions in general. But after all, `IntWithPower` might be used in other contexts as well - as an integer.

For that we have to define `operator int()`, a.k.a. a conversion operator:

```
class IntWithPower {
public:
  IntWithPower(int num) : m_num(num) {}
  int power(IntWithPower exp) {
    return pow(m_num, exp);
  }
  operator int() const { return m_num; }
private:
  int m_num;
}
```

If we check our example now, - after having removed the constraints on the exponent - we'll see that both all our calls succeed.

Let's have a look at the full example now:

[28] http://www.cplusplus.com/reference/cmath/pow/

```cpp
#include <cmath>
#include <iostream>
#include <string>
#include <concepts>
#include <type_traits>

template <typename Base, typename Exponent>
concept has_power = std::convertible_to<Exponent, int> &&
    requires (Base base, Exponent exponent) {
        base.power(exponent);
};

class IntWithPower {
public:
  IntWithPower(int num) : m_num(num) {}
  int power(IntWithPower exp) {
    return pow(m_num, exp);
  }
  operator int() const { return m_num; }
private:
  int m_num;
};

template<typename Exponent>
void printPower(has_power<Exponent> auto number, Exponent exponent) {
  std::cout << number.power(exponent) << '\n';
}

int main() {
  printPower(IntWithPower{5}, IntWithPower{4});
  printPower(IntWithPower{5}, 4L);
  printPower(IntWithPower{5}, 3.0);
}
```

In this example, we can observe how to write a concept that expects the presence of a certain function that can accept a parameter of different constrained types. We also learned that we cannot specialize type traits - such as `std::is_integral` - without risking the consequences of *undefined behaviour* - **so we should completely avoid it**.

Requirements on return types (a.k.a compound requirements)

We've seen how to write a requirement expressing that certain functions must exist in the class' API.

But did we also constrain the return type of those functions?

```
template <typename T>
concept has_square = requires (T t) {
  t.square();
  t.sqrt();
};
```

No, we didn't. A class would satisfy the constraints of `has_square` concept both with `int square()` and with `void square()`.

If you want to specify the expected return type, you must use something that is called a compound requirement.

Here is an example:

```
template <typename T>
concept has_square = requires (T t) {
    {t.square()} -> std::convertible_to<int>;
};
```

Notice the following:

- The expression on what you want to set a return type requirement must be surrounded by braces (`{}`), then comes the arrow (`->`) followed by the constraint on the return type.
- A constraint cannot simply be a type. Had you written simply `int`, you would have received an error message: *return-type-requirement is not a type-constraint.* The original concepts TS allowed the direct usage of types, so if you experimented with that, you might be surprised by this error. This possibility was removed by P1452R2[29].

There are a number of reasons for this removal. One of the motivations was that it would interfere with a future direction of wanting to adopt a generalized form of `auto`, like `vector<auto>` or `vector<my_concept auto>`.

So instead of simply naming a type you have to choose a concept! If you want to set the return type, you have the next two choices:

[29] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1452r2.html

```
{t.square()} -> std::same_as<int>;
{t.square()} -> std::convertible_to<int>;
```

The difference is obvious. In case of `std::same_as`, the return value must be the same as specified as the template argument, while with `std::convertible_to` conversions are allowed.

To demonstrate this, let's have a look at the following example:

```
#include <iostream>
#include <concepts>

template <typename T>
concept has_int_square = requires (T t) {
    {t.square()} -> std::same_as<int>;
};

template <typename T>
concept has_convertible_to_int_square = requires (T t) {
    {t.square()} -> std::convertible_to<int>;
};

class IntWithIntSquare {
public:
  IntWithIntSquare(int num) : m_num(num) {}
  int square() const {
    return m_num * m_num;
  }
private:
  int m_num;
};

class IntWithLongSquare {
public:
  IntWithLongSquare(int num) : m_num(num) {}
  long square() const {
    return m_num * m_num;
  }
private:
  int m_num;
};

class IntWithVoidSquare {
public:
```

```
36    IntWithVoidSquare(int num) : m_num(num) {}
37    void square() const {
38      std::cout << m_num * m_num << '\n';
39    }
40  private:
41    int m_num;
42  };
43
44
45  void printSquareSame(has_int_square auto number) {
46    std::cout << number.square() << '\n';
47  }
48
49  void printSquareConvertible(has_convertible_to_int_square auto number) {
50    std::cout << number.square() << '\n';
51  }
52
53
54  int main() {
55    printSquareSame(IntWithIntSquare{1}); // int same as int
56    //   printSquareSame(IntWithLongSquare{2}); // long not same as int
57    //   printSquareSame(IntWithVoidSquare{3}); // void not same as int
58    printSquareConvertible(IntWithIntSquare{4}); // int convertible to int
59    printSquareConvertible(IntWithLongSquare{5}); // int convertible to int
60    //   printSquareConvertible(IntWithVoidSquare{6}); // void not convertible to int
61  }
62  /*
63  1
64  16
65  25
66  */
```

In the above example, we can observe that the class with `void square() const` doesn't satisfy either the `has_int_square` or the `has_convertible_to_int_square` concepts.

`IntWithLongSquare`, so the class with the function `long square() const` doesn't satisfy the concept `has_int_square` as long is not the same as `int`, but it does satisfy the `has_convertible_to_int_square` concept as `long` is convertible to `int`.

Class `IntWithIntSquare` satisfies both concepts as an `int` is obviously the same as `int` and it's also convertible to an `int`.

Type requirements

With type requirements, we can express that a certain type is valid in a specific context. Type requirements can be used to verify that

- a certain nested type exists
- a class template specialization names a type
- an alias template specialization names a type

Require the existence of a nested type

You have to use the keyword `typename` along with the type name that is expected to exist:

```
1  template<typename T>
2  concept type_requirement = requires {
3    typename T::value_type;
4  };
```

The concept `type_requirement` requires that the type `T` has a nested type `value_type`.

Let's see how it works:

```
1  #include <iostream>
2  #include <vector>
3
4  template<typename T>
5  concept type_requirement = requires {
6    typename T::value_type; // (1)
7  };
8
9  int main() {
10   type_requirement auto myVec = std::vector<int>{1, 2, 3}; // (2)
11   // type_requirement auto myInt {3}; // (3)
12   // error: deduced initializer does not satisfy placeholder constraints ...
13   //   the required type 'typename T::value_type' is invalid
14 }
```

The expression `type_requirement auto myVec = std::vector<int>{1, 2, 3}` (2) is valid, while `type_requirement auto myInt {3}` would not compile.

The reason is that `std::vector` has an inner member type `value_type` that is expected at line (1). At the same time, `int` doesn't have any member, in particular `value_type`, so it doesn't satisfy the constraint of `type_requirement`.

A class template specialization should name a type

Just like for the nested types, you have to use the keyword `typename` along with the specialized class template that is expected to be a valid specialization:

```
template<typename T>
concept type_requirement = requires {
  typename Other<T>;
};
```

In this case, the concept `type_requirement` requires that the class template `Other` can be instantiated with `T`.

Let's see how it works. We declare a class template `Other` and make a requirement on the template parameter by making sure that `Other` cannot be instantiated with a `vector` of `int`s.

```
template <typename T>
requires (!std::same_as<T, std::vector<int>>) // parentheses are mandatory!
struct Other{};
```

Now, the line `type_requirement auto myVec = std::vector<int>{1, 2, 3};` fails with the following error message:

```
#include <iostream>
#include <vector>
#include <concepts>

template <typename T>
requires (!std::same_as<T, std::vector<int>>)
struct Other{};

template<typename T>
concept type_requirement = requires {
  typename Other<T>;
};

int main() {
  type_requirement auto myVec = std::vector<int>{1, 2, 3};
}
/*
main.cpp: In function 'int main()':
main.cpp:15:56: error: deduced initializer does not satisfy placeholder constraints
   15 |   type_requirement auto myVec = std::vector<int>{1, 2, 3};
```

```
      |                                                           ^
main.cpp:15:56: note: constraints not satisfied
main.cpp:10:9:   required for the satisfaction of 'type_requirement<
 auto [requires ::type_requirement<<placeholder>, >]>' [with auto
 [requires ::type_requirement<<placeholder>, >] = std::vector<int,
 std::allocator<int> >]
main.cpp:10:27:   in requirements  [with T = std::vector<int, std::allocator<int> >]
main.cpp:11:12: note: the required type 'Other<T>' is invalid
   11 |   typename Other<T>;
      |   ~~~~~~~~~^~~~~~~~
cc1plus: note: set '-fconcepts-diagnostics-depth=' to at least 2 for more detail
*/
```

We can observe that concepts have much better error messages than former templates.

The compiler clearly says that the initializer (that is `std::vector<int>(1, 2, 3)`) does not satisfy the constraint we set.

Then it goes a level deeper and tells that `std::vector<int, std::allocator<int>>` is invalid as `T` in the `Other<T>` template instantiation.

Indeed, that is what is expected as in `Other` we required that that `T` cannot be `std::vector<int>`.

With type requirements, we can make sure that a given template specialization is possible and get a meaningful error message in case we want to use a type that doesn't satisfy the constraints.

An alias template specialization should name a type

To show that a concept can be used to make sure that an alias template specialization names a type, let's create a template alias `ValueTypeOf`:

```
template<typename T> using ValueTypeOf = T::value_type;
```

And use it in the concept `type_requirement`:

```
template<typename T>
concept type_requirement = requires {
  typename ValueTypeOf<T>;
};
```

Let's try to declare two variables with the just defined constraint, a `vector` of `int`s and an `int`.

```cpp
#include <vector>

template<typename T> using ValueTypeOf = T::value_type;

template<typename T>
concept type_requirement = requires {
  typename ValueTypeOf<T>;
};

int main() {
  type_requirement auto myVec = std::vector<int>{1, 2, 3};
  //  type_requirement auto myInt = int{3};
  //  error: deduced initializer does not satisfy placeholder constraints
  //  note: constraints not satisfied
  //  required for the satisfaction of
  //   'type_requirement<auto [requires ::type_requirement<<placeholder>, >]>'
  //   [with auto [requires ::type_requirement<<placeholder>, >] = int]
  //   in requirements  [with T = int]
  //   note: the required type 'ValueTypeOf<T>' is invalid
  //   typename ValueTypeOf<T>;
  // cc1plus: note: set '-fconcepts-diagnostics-depth=' to at least 2 for more detail

}
```

Given that we know that concepts generate meaningful error messages, let's have a look and try to decipher what has just happened. `type_requirement auto myVec = std::vector<int>{1, 2, 3};` compiles fine as `type_requirement` concept is well-modelled by a vector of integers.

`type_requirement` requires the `ValueTypeOf<T>` alias template specialization is valid. If we expand `ValueTypeOf<T>`, we see that `T` must have a `value_type` type and a vector has. Therefore `type_requirement auto myVec = std::vector<int>{1, 2, 3};` compiles.

At the same time, we you uncomment `type_requirement auto myInt = int{3};`, the compilation breaks. An `int` doesn't model the `type_requirement` concept as an `int` doesn't have an inner `value_type` - or any type as a matter of fact; `ValueTypeOf<int>` is invalid.

Nested requirements

We can use nested requirements to specify additional constraints in a concept without introducing another named concepts.

You can think of nested requirements as one would think about lambda functions for STL algorithms. You can use lambdas to alter the behaviour of an algorithm without the need of naming a function or a function object.

In this case, you can write a constraint more suitable for your needs without the need of naming one more constraint that you'd only use in one (nested) context.

Its syntax follows the following form:

```
requires constraint-expression;
```

Let's start with a simpler example. Where the concept `coupe` uses two other concepts `car` and `convertible`.

```
#include <iostream>

struct AwesomeCabrio {
  void openRoof(){}
  void startEngine(){}
};

struct CoolCoupe {
    void startEngine(){}
};

template<typename C>
concept car = requires (C car) {
    car.startEngine();
};


template<typename C>
concept convertible = car<C> && requires (C car) {
    car.openRoof();
};


template<typename C>
concept coupe = car<C> && requires (C car) {
    requires !convertible<C>;
};


int main() {
  convertible auto cabrio = AwesomeCabrio{};

  //  coupe auto notACoupe = AwesomeCabrio{};
  // nested requirement '! convertible<C>' is not satisfied

```

```
36    coupe auto coupe = CoolCoupe{};
37  }
```

Let's have a look at the concept `coupe`. First, we make sure that only types satisfying the `car` concept are accepted. Then we introduce a nested concept that requires that our template type is not a `convertible`.

It's true that we don't *need* the nested constraint, we could express ourselves without it:

```
1  template<typename C>
2  concept coupe = car<C> && !convertible<C>;
```

Nevertheless, we saw the syntax in a working example.

Nested requires clauses can be used more effectively with local parameters that are listed in the outer `requires` scope, like in the next example with `C clonable`:

```
1  #include <iostream>
2
3  struct Droid {
4    Droid clone(){
5      return Droid{};
6    }
7  };
8
9  struct DroidV2 {
10   Droid clone(){
11     return Droid{};
12   }
13 };
14
15 template<typename C>
16 concept clonable = requires (C clonable) {
17     clonable.clone();
18     requires std::same_as<C, decltype(clonable.clone())>;
19 };
20
21
22 int main() {
23   clonable auto c = Droid{};
24   // clonable auto c2 = DroidV2{};
25   // nested requirement 'same_as<C, decltype (clonable.clone())>'
26   // is not satisfied
27 }
```

In this example, we have two droid types, `Droid` and `DroidV2`. We expect that droids should be clonable meaning that each type should have a `clone()` method that returns another droid of the same type. With `DroidV2` we made a mistake and it still returns `Droid`.

Can we write a concept that catches this error?

We can, in fact as you probably noticed, we already did. In the concept `clonable` we work with a `C cloneable` local parameter. With the nested requirement `requires std::same_as<C, decltype(clonable.clone())>` we express that the clone method should return the same type as the parameter's type.

You might argue that there is another way to express this, without the nested clause and you'd be right:

```
template<typename C>
concept clonable = requires (C clonable) {
    { clonable.clone() } -> std::same_as<C>;
};
```

For a more complex example, I'd recommend you to check the implementation of `SemiRegular` concepts on C++ Reference[30].

To incorporate one of the requirements of `Semiregular` to our `clonable` concept, we could write this:

```
template<typename C>
concept clonable = requires (C clonable) {
    { clonable.clone() } -> std::same_as<C>;
    requires std::same_as<C*, decltype(&clonable)>;
};
```

This additional line makes sure that the address of operator (`&`) returns the same type for the `cloneable` parameter as `C*` is.

I agree it doesn't make much sense in this context (it does for `SemiRegular`), but it's finally an example that is not easier to express without a nested requirement than with.

[30] https://en.cppreference.com/w/cpp/language/constraints

Conclusion

In this chapter, we discussed in detail how to write our own concepts.

- We started by combining already existing ones such as `std::integral<T>` or `std::floating_point<T>`
- We also discussed what kind of combinations are valid, what it means to negate in the context of concepts and how the compiler can find the most constrained constraint
- We learned how we can express constraints. We defined constraints on what API an accepted type must have, then we expressed expectations on inner types, template aliases and specializations.

We saw how to nest requirements, even though for nested concepts we haven't seen a good example yet. That's something that we are going to make up for in our final chapter about real-life concepts.

How to use C++ concepts in real life?

We already know everything necessary to write concepts as we examined the corresponding syntax' and rules' ins and outs. Moreover, we checked the main motivations behind concepts, we saw how we can use them and what kind of concepts are shipped with the standard library.

In the previous chapter, we learned all the necessary details to write our own constraints, though our examples were simplistic, serving the purpose of understandable examples.

Now let's see a couple of real-world examples and areas where concepts are applicable.

Numbers finally

We've been playing with a concept called `number` throughout the entire book. I've kept writing that it's incomplete. Let's have a quick reminder why:

```
#include <concepts>
#include <iostream>

template <typename T>
concept number = std::integral<T> || std::floating_point<T>;

auto add(number auto a, number auto b) {
  return a+b;
}

int main() {
    std::cout << "add(1, 2): " << add(1, 2) << '\n';
    std::cout << "add(1, 2.5): " << add(1, 2.14) << '\n';
    // std::cout << "add(\"one\", \"two\"): " << add("one", "two") << '\n'; // error\
: invalid operands of types 'const char*' and 'const char*' to binary 'operator+'
    std::cout << "add(true, false): " << add(true, false) << '\n';
}

/*
add(1, 2): 3
add(1, 2.5): 3.14
add(true, false): 1
*/
```

Our problem is that even though we only want to accept integrals and floating-point numbers, `bool`s are also accepted. They are accepted because `bool` is an integral type.

There is even worse! `add(0, 'a')` returns 97 as `a` is a character and as such it's considered an integral type. The ASCII code of `a` is 97 and if you add that to `0`, you get the result of this call.

But let's say, we really want to accept any numbers and let's say in the constrained world of *real numbers*.

We have to limit the types we accept. As `std::is_floating_point` returns `true` only for `float`, `double` and `long double`, we can fully accept its results. But it's not enough and as we already saw, `std::is_integral` returns `true` for some types that we might not want to accept as numbers.

The following types and their `const` and/or `unsgined` versions are considered integral:

- `bool`,
- `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`,
- `short`, `int`, `long`, `long long`

But we only want to accept the types from the third line, booleans and characters are not our cups of tea.

Before C++20, we'd have to either disallow certain overloads or use static assertions with templates to make sure that only certain types would be accepted as we saw in the chapter *The concept behind C++ concepts*

```
1  template<typename T>
2  T addPreCpp20(T a, T b) {
3      static_assert(std::is_integral_v<T>, "addPreCpp20 requires integral types");
4      return a+b;
5  }
```

The main problem with these that we'd have to do the same steps for each function, for each parameter.

With overloads, we might end up with a too long list of combinations, while templates are too permissive and they accept just any type regardless of your intentions.

C++20 brought us concepts and we have to define our `number` concept only once, and then it's easy to use it.

Just repeat our requirements:

- we only deal with built-in types, we ignore user-defined numeric types
- we want to accept floating-point numbers
- we want to accept integral numbers
- we don't want to accept integral types that can be converted to `int`s such as `bool`s and `char`s.

As the first trial, you might try something like this

```cpp
#include <concepts>

template <typename T>
concept number = (std::integral<T> || std::floating_point<T>)
                 && !std::same_as<T, bool>
                 && !std::same_as<T, char>
                 && !std::same_as<T, char8_t>
                 && !std::same_as<T, char16_t>
                 && !std::same_as<T, char32_t>
                 && !std::same_as<T, wchar_t>;

auto add(number auto a, number auto b) {
  return a+b;
}
```

But we are not done yet. The following compiles and prints 139!

```cpp
unsigned char a = 'a';
std::cout << add(a, 42);
```

We have to include all the unsigned versions! Luckily only `char` has an unsigned variant. `const`s we don't have to forbid as types as `const char` would be implicitly considered as a `char` and therefore it wouldn't compile.

```cpp
#include <concepts>
#include <iostream>

template <typename T>
concept number = (std::integral<T> || std::floating_point<T>)
                 && !std::same_as<T, bool>
                 && !std::same_as<T, char>
                 && !std::same_as<T, unsigned char>
                 && !std::same_as<T, char8_t>
                 && !std::same_as<T, char16_t>
                 && !std::same_as<T, char32_t>
                 && !std::same_as<T, wchar_t>;

auto add(number auto a, number auto b) {
  return a+b;
}

int main() {
```

```
    std::cout << "add(1, 2): " << add(1, 2) << '\n';
    std::cout << "add(1, 2.14): " << add(1, 2.14) << '\n';
    // std::cout << "add(\"one\", \"two\"): " << add("one", "two") << '\n';
    //   error: invalid operands of types 'const char*' and
    //    'const char*' to binary 'operator+'
    // std::cout << "add(true, false): " << add(true, false) << '\n';
    //   unsatisfied constraints
    // const char c = 'a';
    // std::cout << add(c, 42) << '\n'; // unsatisfied constraints
    // unsigned char uc = 'a';
    // std::cout << add(uc, 42) << '\n'; // unsatisfied constraints
}
/*
add(1, 2): 3
add(1, 2.14): 3.14
*/
```

Utility functions constrained

Utility functions are most often free or `static` functions and they are meant to be used with certain user-defined types.

Usually using them makes sense only with a handful of types. If the number of types is limited enough, or maybe they are even tied to a class hierarchy, it's straightforward how to use these utilities or at least with what types you can use them.

But if the available types are broad enough, often they are templatized. In such cases, documentation and (template) parameter names can come to the rescue. It's better than nothing, but not optimal. Yet, these utility templates often have no documentation and the names are poorly chosen.

As we all learned, the best documentation is code. The best way to document behaviour is through unit tests and code that expresses its intentions. Preferably by compilation errors, or at worst with runtime failures.

Concepts provide a concise and readable way to tell the reader about the types that are supposed to be used with the constrained entities, in this case with the utilities.

By checking a codebase I often work with I found some helper methods encoding messages using some data objects as inputs. The business object that it takes is a template parameter and the expected types are nowhere listed. Not surprisingly, the parameter name offers very little help. As such, you'll end up either with a try and fail to approach or you have to dig deep in the code to discover what it does with its input.

```
template <typename BusinessObjectT>
void encodeSomeStuff(BusinessObjectT iBusinessObject) {
  // ...
}
```

With concepts, we could make this simpler by creating a concept that lists all the characteristics of the business objects this encoder have to deal with and that's it.

```
template <typename BusinessObjectWithEncodeableStuff_t>
concept business_object_with_encodeable_stuff = requires (BusinessObjectWithEncodeab\
leStuff_t bo) {
  bo.interfaceA();
  bo.interfaceB();
  { bo.interfaceC() } -> std::same_as<int>;
};


void encodeSomeStuff(business_object_with_encodeable_stuff auto iBusinessObject) {
  // ...
}
```

Or if we the concept would not be used in other places, you might not want to name it. In that case, you can simply use its constraints directly:

```
template <typename BusinessObjectWithEncodeableStuff_t>
requires requires (BusinessObjectWithEncodeableStuff_t bo) {
  bo.interfaceA();
  bo.interfaceB();
  { bo.interfaceC() } -> std::same_as<int>;
}
void encodeSomeStuff(BusinessObjectWithEncodeableStuff_t iBusinessObject) {
  // ...
}
```

Do you see that requires is written twice $_{\text{written twice}}$? It's not a typo! This is finally a good place to use nested constraints. We cannot directly use a parameter in a template function with a `requires` clause, but it's possible to use an unnamed constraint, or if you prefer to say so a nested constraint. As it's in a concept definition, that's why I didn't list this example in the section of nested constraints.

With the shown ways, we won't simplify our utilities, but we'll make them self documenting. By using concepts they reveal with kind of types they were meant to be used. Should you try to compile them with many different parameters, you'll receive quite decent error messages from the compiler.

Multiple destructors with C++ concepts

We probably all learnt that one cannot overload the destructor. Hence I write about ***"the"*** destructor and *a* destructor... After all, it has no return type and it doesn't take parameters. It's also not really `const` as it destroys the underlying object.

Yet, there were techniques existing to have multiple destructors in a class and those techniques are getting simplified with C++20.

The need for multiple destructors

But first of all, why would you need multiple destructors?

For optimization reasons for example!

Imagine that you have a class template and you want to have destruction depending on the traits of the template parameters. Trivially destructible types can work with the compiler-generated destructor and it's much faster than the user-defined ones...

Also, while RAII is great and we should write our classes by default with that paradigm having in mind, with a good wrapper we can make non-RAII classes at least to do the clean-up after themselves.

Multiple destructors before C++20

How to implement it without having access to a C++20 compiler?

As I've learned from C++ Weekly[31], you can use std::conditional[32].

std::conditional[33] lets us choose between two implementation at compile-time. If the condition that we pass in as a first parameter evaluates to `true`, then the whole call is replaced with the second parameter, otherwise with the third.

```
1  #include <iostream>
2  #include <string>
3  #include <type_traits>
4
5  class Wrapper_Trivial {
6    public:
7      ~Wrapper_Trivial() = default;
8  };
9
```

[31] https://www.youtube.com/watch?v=A3_xrqr5Kdw
[32] https://en.cppreference.com/w/cpp/types/conditional
[33] https://en.cppreference.com/w/cpp/types/conditional

```
10 class Wrapper_NonTrivial {
11   public:
12     ~Wrapper_NonTrivial() {
13         std::cout << "Not trivial\n";
14     }
15 };
16 
17 template <typename T>
18 class Wrapper : public std::conditional_t<
19         std::is_trivially_destructible_v<T>,
20         Wrapper_Trivial,
21         Wrapper_NonTrivial>
22 {
23     T t;
24 };
25 
26 int main()
27 {
28     Wrapper<int> wrappedInt;
29     Wrapper<std::string> wrappedString;
30 }
31 /*
32 Not trivial
33 */
```

Our `Wrapper` class doesn't include a destructor, but it inherits it either from `Wrapper_Trivial` or `Wrapper_NonTrivial` based on a condition, based on whether the contained type `T` is trivially destructible or not.

It's a bit ugly, almost write-only code.

Multiple destructors with C++20

C++ concepts help us to simplify the above example. Still with no run-time costs, and probably with cheaper write costs.

```cpp
#include <iostream>
#include <string>
#include <type_traits>

template <typename T>
class Wrapper
{
    T t;
 public:
    ~Wrapper() requires (!std::is_trivially_destructible_v<T>) {
        std::cout << "Not trivial\n";
    }

    ~Wrapper() = default;
};

int main()
{
    Wrapper<int> wrappedInt;
    Wrapper<std::string> wrappedString;
}
/*
Not trivial
*/
```

We still have a class template, but instead of using the cumbersome to decipher `std::conditional`, first, we wrote a destructor with a `requires` clause. Then we defaulted the unconstrained one.

In the `requires` clause, we constrain that implementation so that the wrapped type cannot be trivially destructible.

The above example works fine with gcc[34]. We receive the expected output. On the other hand, if you try to compile it with the latest clang[35] (as of June 2021), you get a swift compilation error.

[34] godbolt.org/z/rKeETW5jE
[35] godbolt.org/z/rqME8W1rn

```
<source>:19:18: error: invalid reference to function '~Wrapper':
  constraints not satisfied
    Wrapper<int> wrappedInt;
                 ^
<source>:10:26: note: because '!std::is_trivially_destructible_v<int>'
  evaluated to false
    ~Wrapper() requires (!std::is_trivially_destructible_v<T>) {
                         ^
1 error generated.
ASM generation compiler returned: 1
<source>:19:18: error: invalid reference to function '~Wrapper':
  constraints not satisfied
    Wrapper<int> wrappedInt;
                 ^
<source>:10:26: note: because '!std::is_trivially_destructible_v<int>'
  evaluated to false
    ~Wrapper() requires (!std::is_trivially_destructible_v<T>) {
                         ^
1 error generated.
```

Basically, the error message says that the code is not compilable, because `int` is trivially destructible, therefore it doesn't satisfy the requirements of the first destructor which requires a not trivially destructible type.

But wait, `int` should use the other destructor!

While I was looking at the code, I realized that I dislike something about it - apart from the compilation failure. We started with the most specific, with the most constrained overload, instead of going from the general implementation towards the specific.

So I updated the order of the two destructors:

```
#include <iostream>
#include <string>
#include <type_traits>

template <typename T>
class Wrapper
{
    T t;
 public:
    ~Wrapper() = default;

    ~Wrapper() requires (!std::is_trivially_destructible_v<T>) {
```

```
13          std::cout << "Not trivial\n";
14      }
15  };
16
17  int main()
18  {
19      Wrapper<int> wrappedInt;
20      Wrapper<std::string> wrappedString;
21  }
```

Lo and behold! It compiles with clang! But it doesn't produce the expected output. In fact, what happens is that just like previously, only the first declared destructor is taken into account.

We can draw the conclusion that clang doesn't support - yet - multiple destructors and cannot handle concepts well in the context of destructors. A bug report has been filed for LLVM.

Just for the note, I asked a colleague who was access to MSVC, the above examples work fine not only with gcc but with the MS compiler too.

Conclusion

In this last chapter, we saw some real-life examples to demonstrate how concepts can increase the readability of your code.

- We wrote the concept `number` to constrain the expected types only to integral or floating-point arithmetic types
- We saw how to enhance the understandability of utility functions by constraining them with listing the API of the inputs that they are going to use
- We learned how to provide different destructor implementations for a class template based on the characterisctics, based on the satisfied concepts of the template arguments. Keep in mind that as of June 2021, the compiler support for this feature is not unanimous yet.

Now, it's time to wrap up.

Summary

Thanks for reading this book, now you have the necessary 80%[36] of the knowledge or even more to write beautiful C++ concepts.

The feature of concepts is something we should use as soon as we have a capable compiler at our hands.

With concepts, we can and should improve our templates! Concepts help us validate template arguments at compile-time and they do this in a reusable and scalable way.

We could already define many of the offered constraints before C++20, but if I just mentioned the term SFINAE and `std::enable_if`, half of the audience begged me to stop.

While with C++17 and `if constexpr` the situation significantly improved, concepts are still a game changer. They are extremely readable and they scale well.

The standard library provides us with about half a hundred thoroughly designed concepts and the Core Guidelines[37] recommend we use them whenever possible instead of creating user-defined concepts in vain, we have many good ways to define our own ones.

It's easy to write concepts, but it's difficult to write good ones. Mostly because we don't have yet the experience of what it takes to write good concepts. We - as a community - will need a couple of years to build that experience, to identify the best practices. Don't expect that your concepts will be perfect on the first try. You'll need multiple iterations to make them right and it's fine.

It's good that we have recommendations by the core guidelines, but they cannot replace experience. Hence I urge you that as soon as you close this book - given that you have access to a C++20 compiler - to start using concepts immediately.

Experiment with them, learn about them, try to understand how they can help your application, your library. Remember, you should not use naked typenames anymore as you almost always have some expectations towards types that are used with your templates.

Model, constrain, iterate!

[36] https://en.wikipedia.org/wiki/Pareto_principle
[37] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t-templates-and-generic-programming