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

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:

```
a + b == b + a // commutativity
(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

```cpp
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`:

```cpp
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`:

```cpp
#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();
    c.brake();
};

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

};

void foo(car auto c) {
    std::cout << "foo\n";
}

int main()
{
    Tank t;
    foo(t);
}
/*
foo
*/
```

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 `double`s, maybe even `long doubles` and return a value of the same type.

```cpp
#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 `int`s, 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 double`s and `short`s? 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_casted`) 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 `char`s. You add up two `char`s and they first are cast to `int`s 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?

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

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

template<>
std::string add(std::string, std::string) = delete;

int main() {
  std::cout << add(std::string{"a"}, std::string{"b"}) << '\n';
}
/*
main.cpp: In function 'int main()':
main.cpp:13:54: error: use of deleted function
'T add(T, T) [with T = std::__cxx11::basic_string<char>]'
   13 |   std::cout << add(std::string{"a"}, std::string{"b"}) << '\n';
      |                                                      ^
```

[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.

```cpp
#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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The 4 ways to use concepts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The `requires` clause

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The trailing `requires` clause

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The constrained template parameter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Abbreviated function templates

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

They are all compiled the same way

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

How to choose among the 4 ways?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

C++ concepts with classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The `requires` clause

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Constrained template parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The trailing `requires` clause

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Concepts shipped with the C++ standard library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Concepts in the `<concepts>` header

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

`std::convertible_to` for conversions with fewer surprises

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

`std::totally_ordered` for defined comparisons

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

`std::copyable` for copyable types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Concepts in the `<iterator>` header

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

`std::indirect_unary_predicate<F, I>`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

`std::indirectly_comparable`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Concepts in the `<ranges>` header

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

How to write your own C++ concepts?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The simplest `concept`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Use already defined concepts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Negations come with parentheses

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Subsumption and negations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Requirements on operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Simple requirements on the interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Type requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Require the existence of a nested type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

A class template specialization should name a type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

An alias template specialization should name a type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Nested requirements

How to use C++ concepts in real life?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Numbers finally

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Utility functions constrained

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Multiple destructors with C++ concepts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

The need for multiple destructors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Multiple destructors before C++20

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Multiple destructors with C++20

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconcepts.

Summary