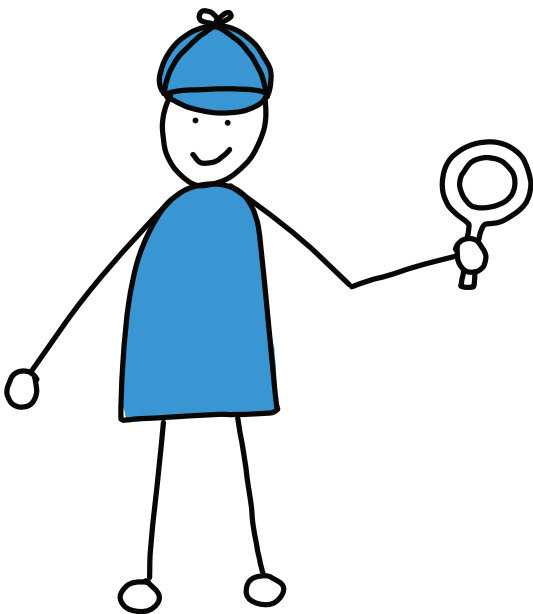


C++20 Templates - The next level

Concepts and more

Presentation Material



C++ Russia, Online, 2021-11-15



© 2021 Andreas Fertig
AndreasFertig.com
All rights reserved

All programs, procedures and electronic circuits contained in this book have been created to the best of our knowledge and belief and have been tested with care. Nevertheless, errors cannot be completely ruled out. For this reason, the program material contained in this book is not associated with any obligation or guarantee of any kind. The author therefore assumes no responsibility and will not accept any liability, consequential or otherwise, arising in any way from the use of this program material or parts thereof.

Version: v1.0

The work including all its parts is protected by copyright. Any use beyond the limits of copyright law requires the prior consent of the author. This applies in particular to duplication, processing, translation and storage and processing in electronic systems.

The reproduction of common names, trade names, product designations, etc. in this work does not justify the assumption that such names are to be regarded as free in the sense of trademark and brand protection legislation and can therefore be used by anyone, even without special identification.

Planning, typesetting and cover design: Andreas Fertig
Cover art and illustrations: Franziska Panter <https://franziskapanter.com>
Production and publishing: Andreas Fertig

Style and conventions

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

```
$ ./a.out  
Hello, C++!
```

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



fertig
adjective /'fɛrtɪç/

finished
ready
complete
completed



Concepts

- With Concepts we can formulate requirements for a type.
- Comparable to `std::enable_if`.
- Concepts consist of the definition of the concept (**concept**) and requirements (**requires**):

```

template-head
template<typename T, typename U>
concept MyConcept = std::same_as<T, U> &&
concept name      (std::is_class_v<T>
                  || std::is_enum_v<T>);
                  requirements

```

- A concept is always a template and can be recognized by the new keyword **concept**. A concept itself consists of other concepts or requirements. The latter are defined by the keyword **requires**.



Variadic template parameters of the same type

```

1 const auto x = Add(2,3,4,5);
2 const auto y = Add(2,3);
3 const auto z = Add(2,3, 3.14); // ERROR

```



Variadic template parameters of the same type

C++17 variant: `enable_if` to block instantiation.

```

1 template<typename T, typename... Ts>
2 constexpr inline bool are_same_v = std::conjunction_v<std::is_same<T, Ts>...>;
3
4 template<typename T, typename...>
5 struct first_arg {
6     using type = T;
7 };
8
9 template<typename... Args>
10 using first_arg_t = typename first_arg<Args...>::type;
11

```

```

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

```



Variadic template parameters of the same type

C++20 variant: `are_same_v` as requirement.

```

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



Application areas for Concepts

Diagram illustrating the application areas for Concepts in a C++20 template function signature:

```

template<C1 T>
requires C2<T>
C3 auto Fun(C4 auto param) requires C5<T>
```

The diagram uses arrows and brackets to identify the following components:

- type-constraint**: Points to the template parameter `T`.
- requires-clause**: Points to the `requires C2<T>` clause.
- constrained placeholder type**: Points to the `C3 auto` parameter type.
- trailing requires-clause**: Points to the `requires C5<T>` clause.



There is more

- Currently Add only prevents
 - Ⓐ mixed types.
- The current version of Add leaves a lot unspecified:
 - Ⓑ Add can nonsensically be called with only one parameter.
 - Ⓒ The type used in *Args* must support the + operation.
 - Ⓓ The operation + should be **noexcept** since Add itself is **noexcept**.
 - Ⓔ The return type of the operation + should match that of *Args*.



Requires-expression

Parameter list of the
requires-expression.

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

Body of the
requires-expression

One or multiple requirements.

- Requires-clause (C3, C5) is a boolean expression. A requires-expression is more complicated. Hello, **noexcept**.
- We can see a requires-clause like an **if** that evaluates a boolean expression and a requires-expression returns such a boolean value.



Requirement kinds

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



Simple requirement

- Checks whether an expression is valid and will compile.

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

```
struct NoAdd{};
```

```
Add(NoAdd{}, NoAdd{});  C
```



Nested requirement

- Evaluates the boolean return value of an expression.

```

1 requires(Args... args)
2 {
3   (... + args);
4   requires are_same_v<Args...>;
5   requires sizeof...(Args) > 1;
6 }

```

C SR: args provides +
A NR: All types are the same
B NR: Pack contains at least two elements

```
struct NoAdd{};
```

```

Add(NoAdd{}, NoAdd{});
Add(2, NoAdd{});
Add(2);

```

C
A
B



Compound requirement

- Checks the return type of an expression. Can be recognized by the curly brackets and the trailing return type.

```

1 requires(Args... args)
2 {
3   (... + args);
4   requires are_same_v<Args...>;
5   requires sizeof...(Args) > 1;
6
7   { (... + args) } noexcept;
8   { (... + args) } noexcept;
9   { (... + args) } noexcept;
10  { (... + args) } noexcept;
11 }

```

C SR: args provides +
A NR: All types are the same
B NR: Pack contains at least two elements
D E CR: ...+args is noexcept and the return type is the same as the first argument type

```

struct NoAdd{};
struct NotNoexcept { NotNoexcept& operator+(const NotNoexcept&); };
struct DifferentReturnType { int& operator+(const DifferentReturnType&) noexcept; };

```

```

Add(NoAdd{}, NoAdd{});
Add(2, NoAdd{});
Add(2);
Add(NotNoexcept{}, NotNoexcept{});
Add(DifferentReturnType{}, DifferentReturnType{});

```

C
A
B
D
E



Ad hoc constraints: requires requires

- The requirements are defined directly after the requires-clause (C2, C5).
- The first **requires** introduces the requires-clause, the second **requires** begins the requires-expression.
- First sign of a code-smell.

```

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

```



Defining a concept

- Definition of a concept is *probably* better:

```

1 template<typename... Args>
2 concept Addable = requires(Args... args)
3 {
4     (... + args);
5     requires are_same_v<Args...>;
6     requires sizeof...(Args) > 1;
7     { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
8 };
9
10 template<typename... Args>
11 requires Addable<Args...>
12 auto Add(Args&&... args)
13 {
14     return (... + args);
15 }

```



Testing the created concept

```

1 // Class template stub to create the different needed properties
2 template<bool noexcept, bool operatorPlus, bool validReturnType>
3 struct Stub {
4     // Operator plus with controlled noexcept can be enabled
5     Stub& operator+(const Stub& rhs) noexcept(noexcept)
6         requires(operatorPlus && validReturnType)
7     { return *this; }
8
9     // Operator plus with invalid return type
10    int operator+(const Stub& rhs) noexcept(noexcept)
11        requires(operatorPlus && not validReturnType)
12    { return {}; }
13 };
14
15 // Create the different stubs from the class template
16 using NoAdd = Stub<true, false, true>;
17 using ValidClass = Stub<true, true, true>;
18 using NotNoexcept = Stub<false, true, true>;
19 using DifferentReturnType = Stub<true, true, false>;

```



Testing the created concept

```

1 A Assert, that mixed types are not allowed
2 static_assert(not Addable<int, double>);
3
4 B Assert that Add is used with at least two parameters
5 static_assert(not Addable<int>);
6
7 C Assert that type has operator+
8 static_assert(Addable<int, int>);
9 static_assert(Addable<ValidClass, ValidClass>);
10 static_assert(not Addable<NoAdd, NoAdd>);
11
12 D Assert that operator+ is noexcept
13 static_assert(not Addable<NotNoexcept, NotNoexcept>);
14
15 E Assert that operator+ returns the same type
16 static_assert(not Addable<DifferentReturnType, DifferentReturnType>);

```



Abbreviated function templates

C++17:

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



Abbreviated function templates

C++20:

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

- **Note:** Functions with `auto` parameters are always templates!

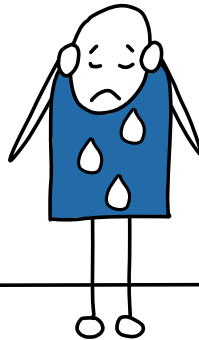


Error messages

```

1 template<typename T, typename U = void> A
2 struct is_container : std::false_type {};
3
4 template<typename T>
5 struct is_container<
6     T,
7     std::void_t<typename T::value_type, B
8                 typename T::size_type,
9                 typename T::allocator_type,
10                typename T::iterator,
11                typename T::const_iterator,
12                decltype(std::declval<T>().size()),
13                decltype(std::declval<T>().begin()),
14                decltype(std::declval<T>().end()),
15                decltype(std::declval<T>().cbegin()),
16                decltype(std::declval<T>().cend())>>
17 : std::true_type {};
18
19 struct A {};
20
21 static_assert(!is_container<A>::value); C
22 static_assert(is_container<std::vector<int>>::value);

```



Error messages: Now helpful

```
1 template<typename T>
2 concept container = requires(T t)
3 {
4     typename T::value_type;
5     typename T::size_type;
6     typename T::allocator_type;
7     typename T::iterator;
8     typename T::const_iterator;
9     t.size();
10    t.begin();
11    t.end();
12    t.cbegin();
13    t.cend();
14 };
15
16 struct A {};
17
18 static_assert(not container<A>);
19 static_assert(container<std::vector<int>>);
```



C++20 non-type template parameters (NTPs)

```

1 template<double D>
2 void Fun();
3
4 void Use()
5 {
6     Fun<+0.0>();
7     Fun<-0.0>();
8 }

```



C++20 NTPs

C++20

```

1 template<typename CharT, std::size_t N>
2 struct fixed_string {
3     CharT data[N]{};
4
5     constexpr fixed_string(const CharT (&str)[N]) { std::copy_n(str, N, data); }
6 };
7
8 fixed_string fs{"Hello, C++20"};

```



C++20 NTTPs

C++20

```

1 template<fixed_string Str> A Here we have a NTTP
2 struct FixedStringContainer {
3     B Use Str
4     void print() { std::cout << Str.data << '\n'; }
5 };
6
7 void Use()
8 {
9     C We can instantiate the template with a string
10    FixedStringContainer<"Hello, C++"> fc{};
11    fc.print(); D For those who believe it only if they see it
12 }

```



Lambdas with a template-head

C++20

```

1 int main()
2 {
3     auto max = [](auto x, auto y) {
4         return (x > y) ? x : y;
5     };
6
7     max(2, 3); // ok
8     max(2, 3.0); // not wanted
9 }

```



Lambdas with a template-head

C++20

```

1 int main()
2 {
3     auto max = []<typename T>(T x, T y)
4     {
5         return (x > y) ? x : y;
6     };
7
8     max(2, 3); // ok
9     // max(2, 3.0); // does not compile anymore
10 }

```

Andreas Fertig
v2.0

C++20 Templates - The next level

28

Lambdas with a template-head

C++20

```

1 auto lamb = []<typename T>(std::vector<T> t) {};
2 std::vector<int> v{};
3
4 lamb(v);
5 // lamb(20);

```

```

1 auto lamb = []<size_t N>(std::array<int, N> x) {};
2
3 std::array<int, 2> a{};
4
5 lamb(a);

```

Andreas Fertig
v2.0

C++20 Templates - The next level

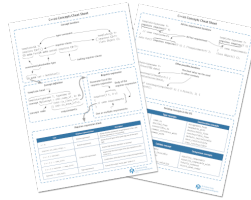
29



}

I am Fertig.

C++20 Concepts Cheat Sheet



fertig.to/subscribe



Andreas Fertig
v2.0

C++20 Templates - The next level

30

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.

- g++ 11.1.0
- clang version 13.0.0

Typography

- Main font:

- Camingo Dos Pro by Jan Fromm (<https://janfromm.de/>)

- Code font:

- CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 <http://creativecommons.org/licenses/by-nd/3.0/>



Andreas Fertig
v2.0

C++20 Templates - The next level

31



References

Images:

21: Franziska Panter

23: Franziska Panter

33: Franziska Panter

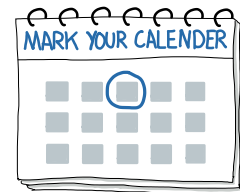


Upcoming Events

For my upcoming talks you can check <https://andreasfertig.com/talks/>.

For my courses you can check <https://andreasfertig.com/courses/>.

Like to always be informed? Subscribe to my newsletter: <https://andreasfertig.com/newsletter/>.



About **Andreas Fertig**



Photo: Kristijan Matic www.kristijanmatic.de

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

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

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

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

You can find Andreas online at andreasfertig.com.

