



C++20 Concepts - We have to rethink what we did in the past



Andreas Fertig
<https://AndreasFertig.Info>
post@AndreasFertig.Info
@Andreas_Fertig

fertig
adjective /'fɛrtɪç/

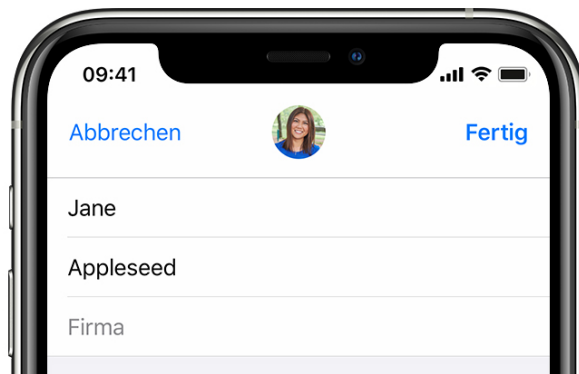
finished
ready
complete
completed



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

3



©Apple



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

4



Concepts

```

1 template<typename T> A template-head
2 concept arithmetic = B concept keyword and name of the concept
3   std::integral<T> or std::is_floating_point_v<T>; C Requirements

```



Application areas for Concepts

```

          type-constraint
          |
          v
template<C1 T>
requires C2<T> }
C3 auto Fun(C4 auto param) requires C5<T>
          |
          v
    constrained placeholder type

```

requires-clause

trailing requires-clause



Requires-expression

```

1 template<typename T>
2 concept HashableSTLContainer = requires(T a) { A requires expression
3   B Body of the requires expression
4   std::hash<T>{}(a); C Simple requirement
5   requires sizeof(a) > 1; D Nested requirement
6   E Compound requirement
7   { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
8   typename T::size_type; F Type requirement
9 };

```



Other places

```

1 const std::vector v{3, 4, 5, 6};
2
3 struct A {
4   void Print();
5 };
6

```

```

1 template<typename T>
2 void Print(T&& data)
3 {
4   A Concept returns a bool value
5   if constexpr(std::input_or_output_iterator<T>) {
6     PrintContainer(data);
7
8     B We can also use an ad hoc requirement
9   } else if constexpr(requires(T t) { t.Print(); }) {
10    data.Print();
11  }
12 }

```



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!



What can we do with Concepts?



Inheritance for everything

```
1 struct WiFi {
2     virtual ~WiFi() = default;
3     virtual void enableRadio() = 0;
4     virtual void disableRadio() = 0;
5 };
6
7 struct GPS {
8     virtual ~GPS() = default;
9     virtual GPSCoords getCoordinates() = 0;
10 };
11
12 class Smartphone : public WiFi, public GPS {
13 public:
14     void enableRadio() override;
15     void disableRadio() override;
16
17     GPSCoords getCoordinates() override;
18 };
19
20 void ToggleWiFi(WiFi& phone);
```



Down with inheritance! ...?



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

13

We want abstraction not inheritance!



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

14



Implements

```

1 interface WiFi {
2     void enableRadio();
3     void disableRadio();
4 }
5
6 interface GPS {
7     GPSCoords getCoordinates();
8 }
9
10 class Smartphone implements WiFi, GPS {
11     public void enableRadio() { /* ... */ }
12     public void disableRadio() { /* ... */ }
13     public GPSCoords getCoordinates() { /* ... */ }
14 }

```



Implements

Defining an interface in C++

```

1 template<typename T>
2 concept WiFi = requires(T t) {
3     { t.enableRadio() } -> std::same_as<void>;
4     { t.disableRadio() } -> std::same_as<void>;
5 };
6
7 template<typename T>
8 concept GPS = requires(T t) {
9     { t.getCoordinates() } -> std::same_as<GPSCoords>;
10 };

```



Implements

Implementing the interface

```

1 class Smartphone {
2     static_assert(WiFi<Smartphone> and GPS<Smartphone>);
3
4 public:
5     void enableRadio();
6     void disableRadio();
7
8     GPSCoords getCoordinates();
9 };
10
11 void ToggleWiFi(WiFi auto& phone);

```



Implements

Checking the interface

```

1 class Smartphone {
2     static_assert(WiFi<Smartphone> and GPS<Smartphone>);
3
4 public:
5     void enableRadio();
6     void disableRadio();
7
8     GPSCoords getCoordinates();
9 };
10
11 void ToggleWiFi(WiFi auto& phone);

```



Implements

Checking the interface... second attempt

```

1 class Smartphone {
2 public:
3     void enableRadio();
4     void disableRadio();
5
6     GPSCoords getCoordinates();
7 };
8
9 static_assert(WiFi<Smartphone> and GPS<Smartphone>);
10
11 void ToggleWiFi(WiFi auto& phone);

```



Implements

Checking the interface... how it should look like

```

1 class Smartphone {
2     implements(WiFi, GPS);
3
4 public:
5     void enableRadio();
6     void disableRadio();
7
8     GPSCoords getCoordinates();
9 };
10
11 void ToggleWiFi(WiFi auto& phone);

```



Implements

Checking the interface... a solution... apologies for the macro

```

1 #define implements(name, ...) \
2 constexpr void __check_implements() { \
3     enum class Helper { name, __VA_ARGS__ __VA_OPT__(, ) Last }; A \
4     B \
5     auto impl = []<auto N>(auto& self, auto* t, auto*... ts) { \
6         if constexpr ((sizeof...(ts) + 1) < N) { \
7             self.template operator<>N>(self, t, t, ts...); C \
8         } else { D \
9             E \
10            []<name __VA_OPT__(, ) __VA_ARGS__>() { \
11                }.template operator<>std::remove_pointer_t<decltype(t)>, \
12                std::remove_pointer_t<decltype(ts)>...>(); \
13            } \
14        }; \
15        F \
16        impl.template operator<>static_cast<int>(Helper::Last)>(impl, this); \
17    }

```



You still think inheritance is good?



Prepare to see the pain!



Dependent destructor

```
1 struct NotTriviallyDestructible {  
2     ~NotTriviallyDestructible() {} A Make it not trivially destructible  
3 };  
4  
5 struct TriviallyDestructible {}; B A type which is trivially destructible  
6  
7 static_assert(  
8     not std::is_trivially_destructible_v<Wrapper<NotTriviallyDestructible>>);  
9 static_assert(std::is_trivially_destructible_v<Wrapper<TriviallyDestructible>>);
```



Dependent destructor

```

1 template<typename T>
2 class Wrapper {
3     union {
4         char __null_state_;
5         T    mData;
6     };
7
8 public:
9     ~Wrapper() requires(not std::is_trivially_destructible_v<T>) { mData.~T(); }
10    ~Wrapper() = default;
11 };

```



The Power of Concepts

- Updating a Standard Template Library (STL) data type to C++20
 - Using `std::optional` as an example
 - Using `libc++`
 - with `g++` (well, ... yes, this is not a typo)
 - <https://github.com/andreasfertig/llvm-project/tree/optionalCpp20>



__optional_destructor_base

```

1 template<class _Tp>
2 struct __optional_destructor_base<_Tp, false> {
3     typedef _Tp value_type;
4     static_assert(is_object_v<value_type>,
5                 "instantiation of optional with a non-object type "
6                 "is undefined behavior");
7     union {
8         char __null_state_;
9         value_type __val_;
10    };
11    bool __engaged_;
12
13    _LIBCPP_CONSTEXPR_AFTER_CXX17 ~__optional_destructor_base()
14    {
15        if (__engaged_) __val_.~value_type();
16    }
17
18    constexpr __optional_destructor_base() noexcept : __null_state_(), __engaged_(false)
19    {}
20
21    template<class... _Args>
22    constexpr explicit __optional_destructor_base(in_place_t, _Args&&... __args)
23    : __val_(VSTD::forward<_Args>(__args)...), __engaged_(true)
24    {}
25
26    template<class _Fp, class... _Args>
27    _LIBCPP_HIDE_FROM_ABI constexpr __optional_destructor_base(
28        __optional_construct_from_invoke_tag,
29        _Fp&& __f,
30        _Args&&... __args)
31    : __val_(VSTD::invoke(__VSTD::forward<_Fp>(__f), __VSTD::forward<_Args>(__args)...))
32    , __engaged_(true)
33    {}
34
35    _LIBCPP_CONSTEXPR_AFTER_CXX17 void reset() noexcept
36    {
37        if (__engaged_) {
38            __val_.~value_type();
39            __engaged_ = false;
40        }
41    }
42 };

```

```

1 template<class _Tp>
2 struct __optional_destructor_base<_Tp, true> {
3     typedef _Tp value_type;
4     static_assert(is_object_v<value_type>,
5                 "instantiation of optional with a non-object type "
6                 "is undefined behavior");
7     union {
8         char __null_state_;
9         value_type __val_;
10    };
11    bool __engaged_;
12
13    constexpr __optional_destructor_base() noexcept : __null_state_(), __engaged_(false)
14    {}
15
16    template<class... _Args>
17    constexpr explicit __optional_destructor_base(in_place_t, _Args&&... __args)
18    : __val_(VSTD::forward<_Args>(__args)...), __engaged_(true)
19    {}
20
21    template<class _Fp, class... _Args>
22    _LIBCPP_HIDE_FROM_ABI constexpr __optional_destructor_base(
23        __optional_construct_from_invoke_tag,
24        _Fp&& __f,
25        _Args&&... __args)
26    : __val_(VSTD::invoke(__VSTD::forward<_Fp>(__f), __VSTD::forward<_Args>(__args)...))
27    , __engaged_(true)
28    {}
29
30    _LIBCPP_CONSTEXPR_AFTER_CXX17 void reset() noexcept
31    {
32        if (__engaged_) { __engaged_ = false; }
33    }
34 };

```



__optional_copy_base

```

1 template<class _Tp>
2 struct __optional_copy_base<_Tp, false> : __optional_storage_base<_Tp> {
3     using __optional_storage_base<_Tp>::__optional_storage_base;
4
5     _LIBCPP_INLINE_VISIBILITY
6     __optional_copy_base() = default;
7
8     _LIBCPP_INLINE_VISIBILITY
9     _LIBCPP_CONSTEXPR_AFTER_CXX17
10    __optional_copy_base(const __optional_copy_base& __opt)
11    {
12        this->_construct_from(__opt);
13    }
14
15    _LIBCPP_INLINE_VISIBILITY
16    __optional_copy_base(__optional_copy_base&&) = default;
17
18    _LIBCPP_INLINE_VISIBILITY
19    __optional_copy_base& operator=(const __optional_copy_base&) = default;
20
21    _LIBCPP_INLINE_VISIBILITY
22    __optional_copy_base& operator=(__optional_copy_base&&) = default;
23 };

```

```

1 template<class _Tp, bool = is_trivially_copy_constructible<_Tp>::value>
2 struct __optional_copy_base : __optional_storage_base<_Tp> {
3     using __optional_storage_base<_Tp>::__optional_storage_base;
4 };

```



__optional_move_base

```

1 template<class _Tp>
2 struct __optional_move_base<_Tp, false> : __optional_copy_base<_Tp> {
3     using value_type = _Tp;
4     using __optional_copy_base<_Tp>::__optional_copy_base;
5
6     _LIBCPP_INLINE_VISIBILITY
7     __optional_move_base() = default;
8     _LIBCPP_INLINE_VISIBILITY
9     __optional_move_base(const __optional_move_base&) = default;
10
11     _LIBCPP_INLINE_VISIBILITY
12     _LIBCPP_CONSTEXPR_AFTER_CXX17
13     __optional_move_base(__optional_move_base&& __opt) noexcept(
14         is_nothrow_move_constructible_v<value_type>)
15     {
16         this->__construct_from(_VSTD::move(__opt));
17     }
18
19     _LIBCPP_INLINE_VISIBILITY
20     __optional_move_base& operator=(const __optional_move_base&) = default;
21     _LIBCPP_INLINE_VISIBILITY
22     __optional_move_base& operator=(__optional_move_base&&) = default;
23 };

```

```

1 template<class _Tp, bool = is_trivially_move_constructible<_Tp>::value>
2 struct __optional_move_base : __optional_copy_base<_Tp> {
3     using __optional_copy_base<_Tp>::__optional_copy_base;
4 };

```



__optional_copy_assign_base

```

1 template<class _Tp>
2 struct __optional_copy_assign_base<_Tp, false> : __optional_move_base<_Tp> {
3     using __optional_move_base<_Tp>::__optional_move_base;
4
5     _LIBCPP_INLINE_VISIBILITY
6     __optional_copy_assign_base() = default;
7     _LIBCPP_INLINE_VISIBILITY
8     __optional_copy_assign_base(const __optional_copy_assign_base&) = default;
9     _LIBCPP_INLINE_VISIBILITY
10    __optional_copy_assign_base(__optional_copy_assign_base&&) = default;
11
12    _LIBCPP_INLINE_VISIBILITY
13    _LIBCPP_CONSTEXPR_AFTER_CXX17 __optional_copy_assign_base&
14    operator=(const __optional_copy_assign_base& __opt)
15    {
16        this->__assign_from(__opt);
17        return *this;
18    }
19
20    _LIBCPP_INLINE_VISIBILITY
21    __optional_copy_assign_base& operator=(__optional_copy_assign_base&&) = default;
22 };

```

```

1 template<class _Tp,
2         bool =
3         is_trivially_destructible<_Tp>::value&& is_trivially_copy_constructible<
4         _Tp>::value&& is_trivially_copy_assignable<_Tp>::value>
5     > struct __optional_copy_assign_base : __optional_move_base<_Tp> {
6     using __optional_move_base<_Tp>::__optional_move_base;
7 };

```



__optional_move_assign_base

```

1 template<class Tp>
2 struct __optional_move_assign_base<Tp, false> : __optional_copy_assign_base<Tp> {
3     using value_type = Tp;
4     using __optional_copy_assign_base<Tp>::__optional_copy_assign_base;
5
6     _LIBCPP_INLINE_VISIBILITY
7     __optional_move_assign_base() = default;
8     _LIBCPP_INLINE_VISIBILITY
9     __optional_move_assign_base(const __optional_move_assign_base& __opt) = default;
10    _LIBCPP_INLINE_VISIBILITY
11    __optional_move_assign_base(__optional_move_assign_base&&) = default;
12    _LIBCPP_INLINE_VISIBILITY
13    __optional_move_assign_base&
14    operator=(const __optional_move_assign_base&) = default;
15
16    _LIBCPP_INLINE_VISIBILITY
17    _LIBCPP_CONSTEXPR_AFTER_CXX17 __optional_move_assign_base&
18    operator=(__optional_move_assign_base&& __opt) noexcept(
19        is_nothrow_move_assignable_v<value_type>&&
20        is_nothrow_move_constructible_v<value_type>)
21    {
22        this->__assign_from(_VSTD::move(__opt));
23        return *this;
24    }
25 };

```

```

1 template<class Tp,
2         bool =
3         is_trivially_destructible<Tp::value&& is_trivially_move_constructible<
4         Tp::value&& is_trivially_move_assignable<Tp::value>
5 struct __optional_move_assign_base : __optional_copy_assign_base<Tp> {
6     using __optional_move_assign_base<Tp>::__optional_copy_assign_base;
7 };

```



__sfinae_ctor_base

```

1 template<bool _CanCopy, bool _CanMove>
2 struct __sfinae_ctor_base {};
3
4 template<>
5 struct __sfinae_ctor_base<false, false> {
6     __sfinae_ctor_base() = default;
7     __sfinae_ctor_base(__sfinae_ctor_base const&) = delete;
8     __sfinae_ctor_base(__sfinae_ctor_base&&) = delete;
9     __sfinae_ctor_base& operator=(__sfinae_ctor_base const&) = default;
10    __sfinae_ctor_base& operator=(__sfinae_ctor_base&&) = default;
11 };
12
13 template<>
14 struct __sfinae_ctor_base<true, false> {
15     __sfinae_ctor_base() = default;
16     __sfinae_ctor_base(__sfinae_ctor_base const&) = default;
17     __sfinae_ctor_base(__sfinae_ctor_base&&) = delete;
18     __sfinae_ctor_base& operator=(__sfinae_ctor_base const&) = default;
19     __sfinae_ctor_base& operator=(__sfinae_ctor_base&&) = default;
20 };
21
22 template<>
23 struct __sfinae_ctor_base<false, true> {
24     __sfinae_ctor_base() = default;
25     __sfinae_ctor_base(__sfinae_ctor_base const&) = delete;
26     __sfinae_ctor_base(__sfinae_ctor_base&&) = default;
27     __sfinae_ctor_base& operator=(__sfinae_ctor_base const&) = default;
28     __sfinae_ctor_base& operator=(__sfinae_ctor_base&&) = default;
29 };

```



__sfinae_assign_base

```

1 template<bool _CanCopy, bool _CanMove>
2 struct __sfinae_assign_base {};
3
4 template<>
5 struct __sfinae_assign_base<false, false> {
6     __sfinae_assign_base() = default;
7     __sfinae_assign_base(__sfinae_assign_base const&) = default;
8     __sfinae_assign_base(__sfinae_assign_base&&) = default;
9     __sfinae_assign_base& operator=(__sfinae_assign_base const&) = delete;
10    __sfinae_assign_base& operator=(__sfinae_assign_base&&) = delete;
11 };
12
13 template<>
14 struct __sfinae_assign_base<true, false> {
15     __sfinae_assign_base() = default;
16     __sfinae_assign_base(__sfinae_assign_base const&) = default;
17     __sfinae_assign_base(__sfinae_assign_base&&) = default;
18     __sfinae_assign_base& operator=(__sfinae_assign_base const&) = default;
19     __sfinae_assign_base& operator=(__sfinae_assign_base&&) = delete;
20 };
21
22 template<>
23 struct __sfinae_assign_base<false, true> {
24     __sfinae_assign_base() = default;
25     __sfinae_assign_base(__sfinae_assign_base const&) = default;
26     __sfinae_assign_base(__sfinae_assign_base&&) = default;
27     __sfinae_assign_base& operator=(__sfinae_assign_base const&) = delete;
28     __sfinae_assign_base& operator=(__sfinae_assign_base&&) = default;
29 };

```



Finally... std::optional

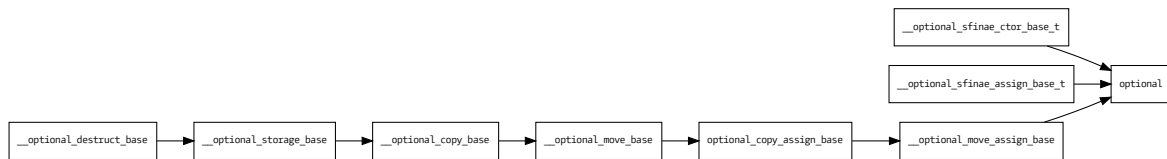
```

1 template<class _Tp>
2 class optional : private __optional_move_assign_base<_Tp>,
3                 private __optional_sfinae_ctor_base_t<_Tp>,
4                 private __optional_sfinae_assign_base_t<_Tp> {
5     using __base = __optional_move_assign_base<_Tp>;
6
7     // ...
8
9     constexpr optional() noexcept {}
10    constexpr optional(const optional&) = default;
11    constexpr optional(optional&&) = default;
12    constexpr optional(nullopt_t) noexcept {}
13
14    // ...
15
16    optional& operator=(const optional&) = default;
17    optional& operator=(optional&&) = default;
18
19    // other optional operations
20 }

```



The Power of Concepts - std::optional



std::optional with Concepts

```

1 template<class _Tp>
2 class optional {
3 public:
4     using value_type = _Tp;
5
6 private:
7     static_assert(
8         is_object_v<value_type>,
9         "instantiation of optional with a non-object type is undefined behavior");
10    union {
11        char __null_state_;
12        value_type __val_;
13    };
14    bool __engaged_{false};
15
16 public:
17     // ctor / dtor
18     // copy-operations
19     // move-operations
20     //
21     // other optional operations
22 };
  
```



std::optional with Concepts

```

1 constexpr ~optional() requires(not is_trivially_destructible_v<Tp>) { if (has_value()) __val_~value_type(); }
2 ~optional() = default;
3
4 constexpr void reset() noexcept
5 {
6     if (has_value()) {
7         if constexpr (not is_trivially_destructible_v<Tp>) { __val_~value_type(); }
8         __engaged_ = false;
9     }
10 }
11
12 constexpr optional() noexcept : __null_state_() {}

```



std::optional with Concepts

```

1 constexpr optional(const optional& __opt)
2 requires(not is_trivially_copy_constructible_v<Tp> and is_copy_constructible_v<Tp>)
3 {
4     this->__construct_from(__opt);
5 }
6
7 optional(const optional& __opt) = default;
8
9 constexpr optional& operator=(const optional& __opt)
10 requires(not is_trivially_copy_assignable_v<Tp> and is_copy_constructible_v<Tp> and is_copy_assignable_v<Tp>)
11 {
12     this->__assign_from(__opt);
13     return *this;
14 }
15
16 optional& operator=(const optional&)
17 requires(is_trivially_destructible_v<Tp> and
18         is_trivially_copy_constructible_v<Tp> and is_trivially_copy_assignable_v<Tp>) = default;

```



std::optional with Concepts

```

1 constexpr optional(optional&& __opt) noexcept(is_nothrow_move_constructible_v<value_type>)
2 requires(not is_trivially_move_constructible_v<_Tp> and is_move_constructible_v<_Tp>)
3 {
4     this->__construct_from(_VSTD::move(__opt));
5 }
6
7 optional(optional&&) = default;
8
9 constexpr optional& operator=(optional&& __opt)
10 noexcept(is_nothrow_move_assignable_v<value_type> and is_nothrow_move_constructible_v<value_type>)
11 requires(not is_trivially_move_assignable_v<_Tp> and is_move_constructible_v<_Tp> and is_move_assignable_v<_Tp>)
12 {
13     this->__assign_from(_VSTD::move(__opt));
14     return *this;
15 }
16
17 optional& operator=(optional&&)
18 requires(is_trivially_destructible_v<_Tp> and
19         is_trivially_move_constructible_v<_Tp> and is_trivially_move_assignable_v<_Tp>) = default;

```



The Power of Concepts - std::optional

	Before	After
Classes	22 ¹ (including optional)	1
Lines of code ²	1'187	1'068
Speed ³	463ms	391ms

```

1 #include <optional>
2
3 struct NotTriviallyDestructible {
4     ~NotTriviallyDestructible() {}
5 };
6
7 int main()
8 {
9     std::optional<int> o{};
10    std::optional<NotTriviallyDestructible> o2{};
11 }

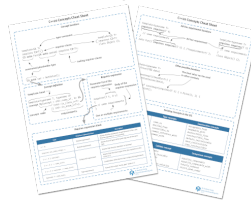
```

¹Not counting __optional_storage_base²Formatted with clang-format³MacMini 2018, 3,2 GHz 6-Core Intel Core i7, 64 GB

}

I am Fertig.

C++20 Concepts Cheat Sheet



fertig.to/subscribe



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

41

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 Concepts - We have to rethink what we did in the past

42



References

Images:

44: Franziska Panter



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

43

Upcoming Events

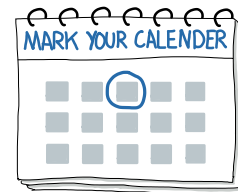
Training Classes

- *Programming with C++20*, Andreas Fertig, July 11 - 15
- *Programmieren mit C++20*, Andreas Fertig, October 05 - 07
- *C++1x for Embedded Systems*, QA Systems, October 18 - 21
- *Programming with C++11 to C++17*, Andreas Fertig, November 07 - 11

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

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

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



Andreas Fertig
v2.0

C++20 Concepts - We have to rethink what we did in the past

44

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 20.

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 to 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.

