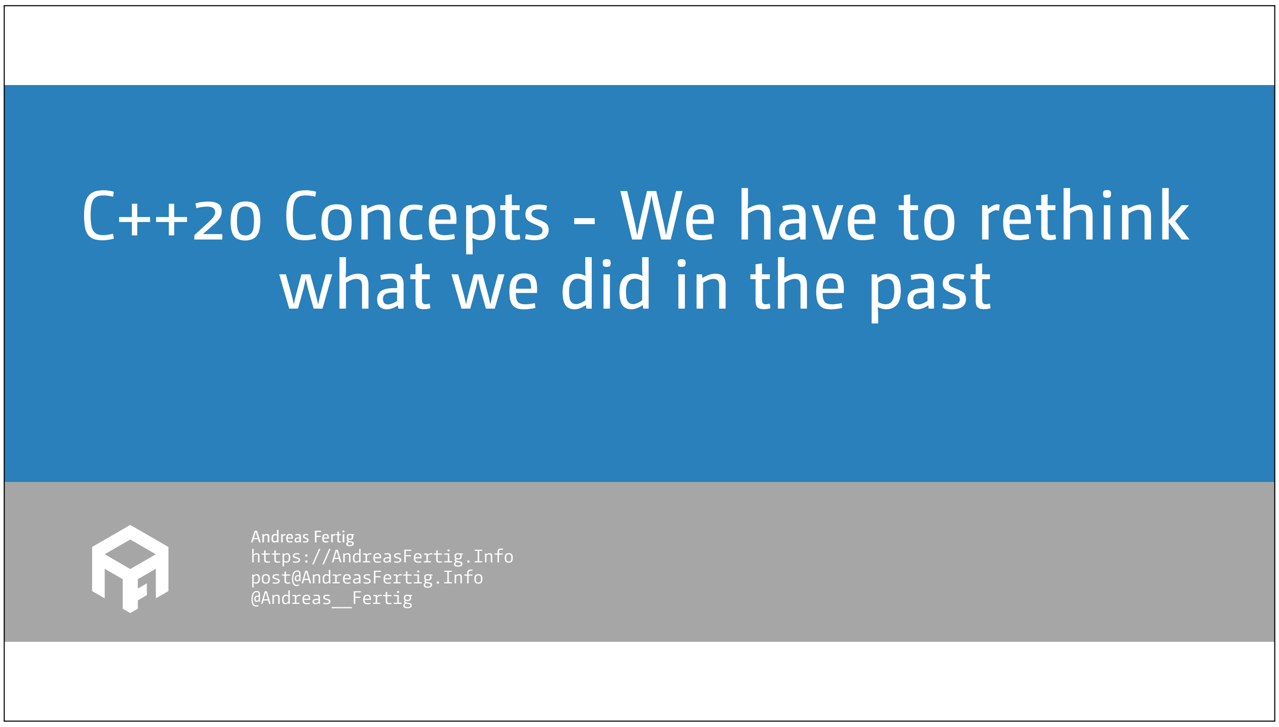
accu 2022

# C++20 CONCEPTS

*WE HAVE TO RETHINK WHAT WE DID IN THE PAST*

ANDREAS FERTIG

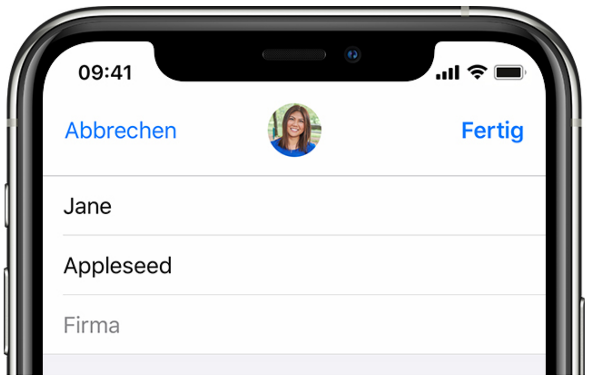

## C++20 Concepts - We have to rethink what we did in the past

Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info
@Andreas__Fertig

## fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed

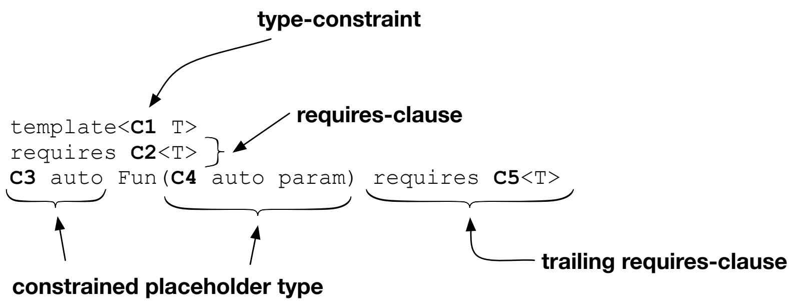


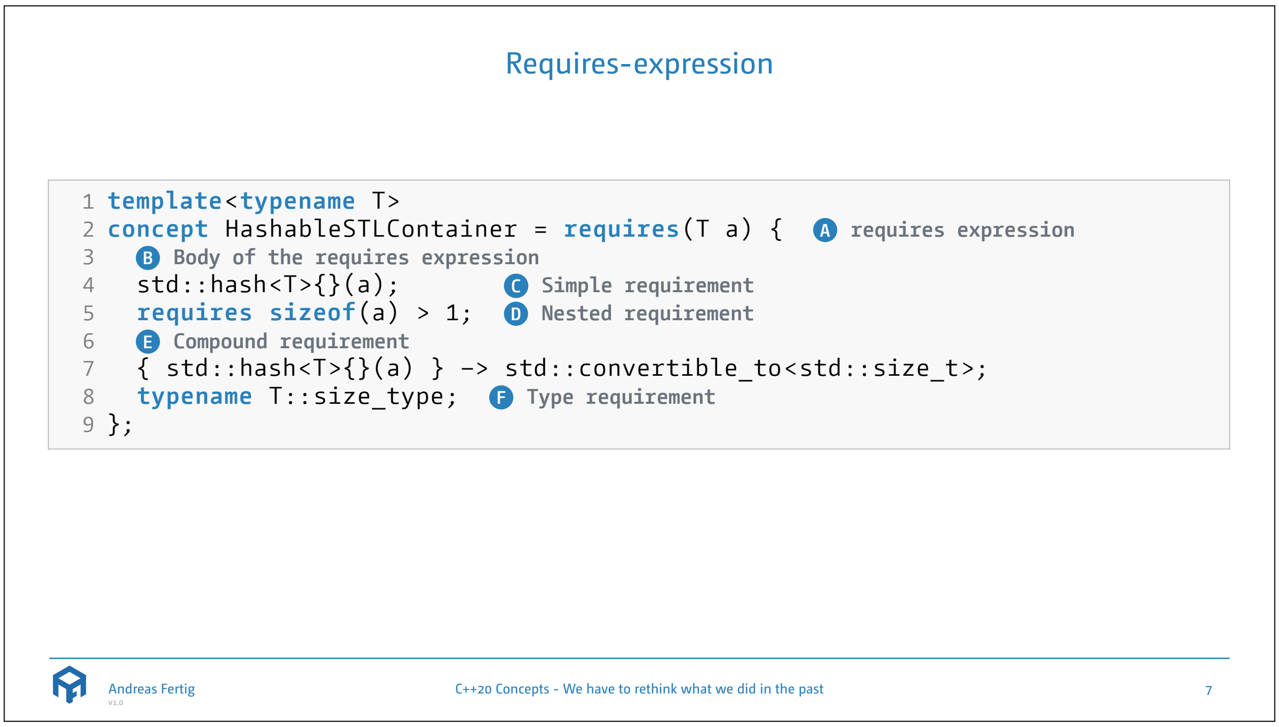
©Apple

## Concepts

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

## Application areas for Concepts

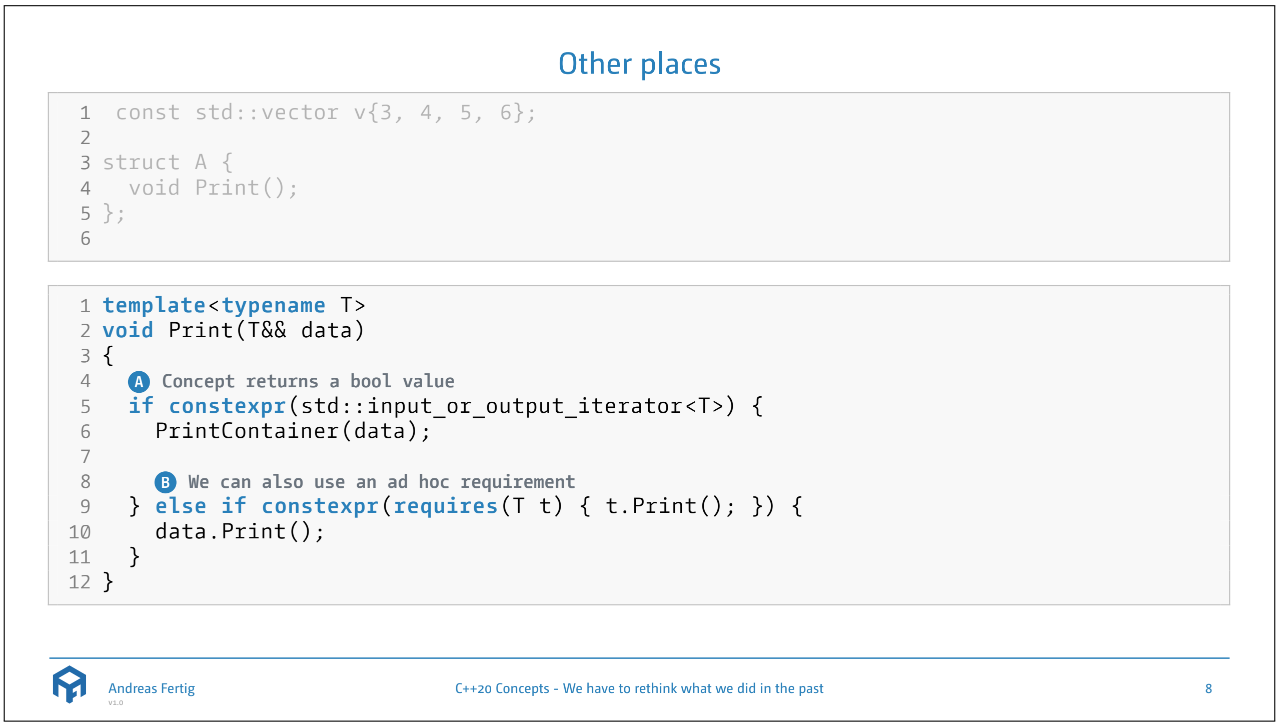

```cpp
template<C1 T>
requires C2<T>
C3 auto Fun(C4 auto param) requires C5<T>
```



## Requires-expression

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

Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 7

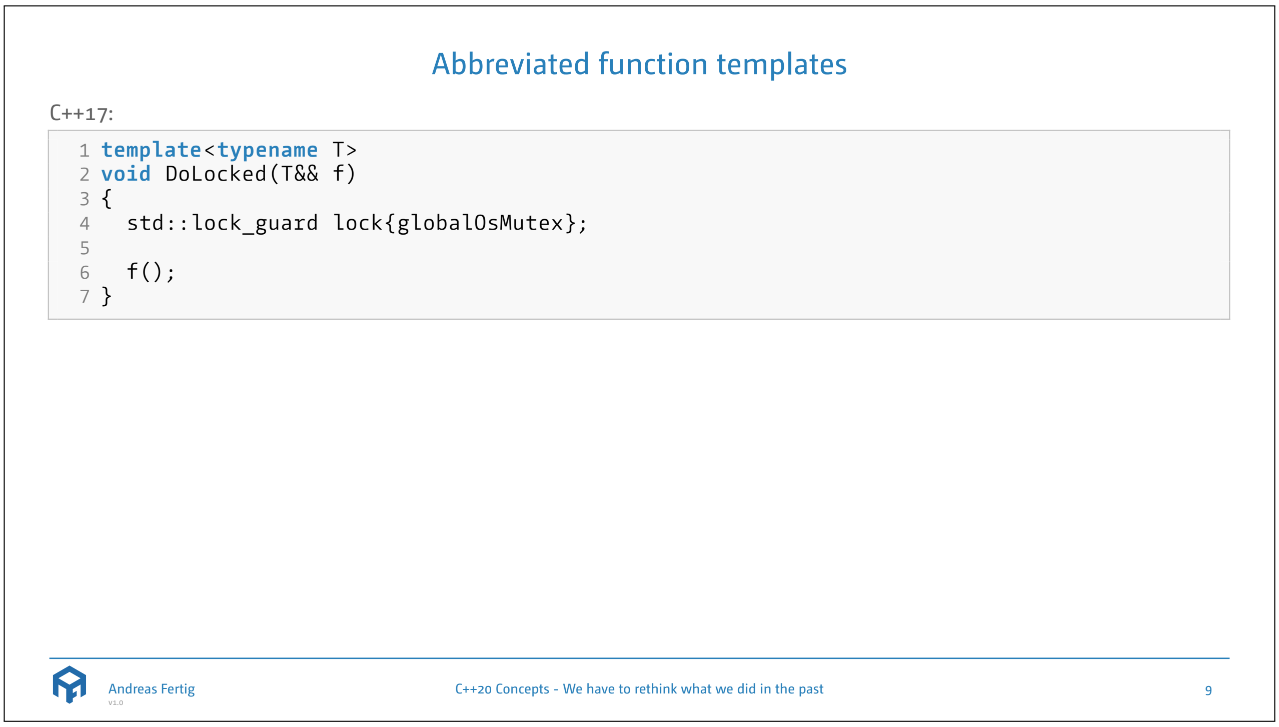


## Other places

```
 const std::vector v{3, 4, 5, 6};

struct A {
  void Print();
};

```

```
template<typename T>
void Print(T&& data)
{
  Ⓐ Concept returns a bool value
  if constexpr(std::input_or_output_iterator<T>) {
    PrintContainer(data);

    Ⓑ We can also use an ad hoc requirement
  } else if constexpr(requires(T t) { t.Print(); }) {
    data.Print();
  }
}
```

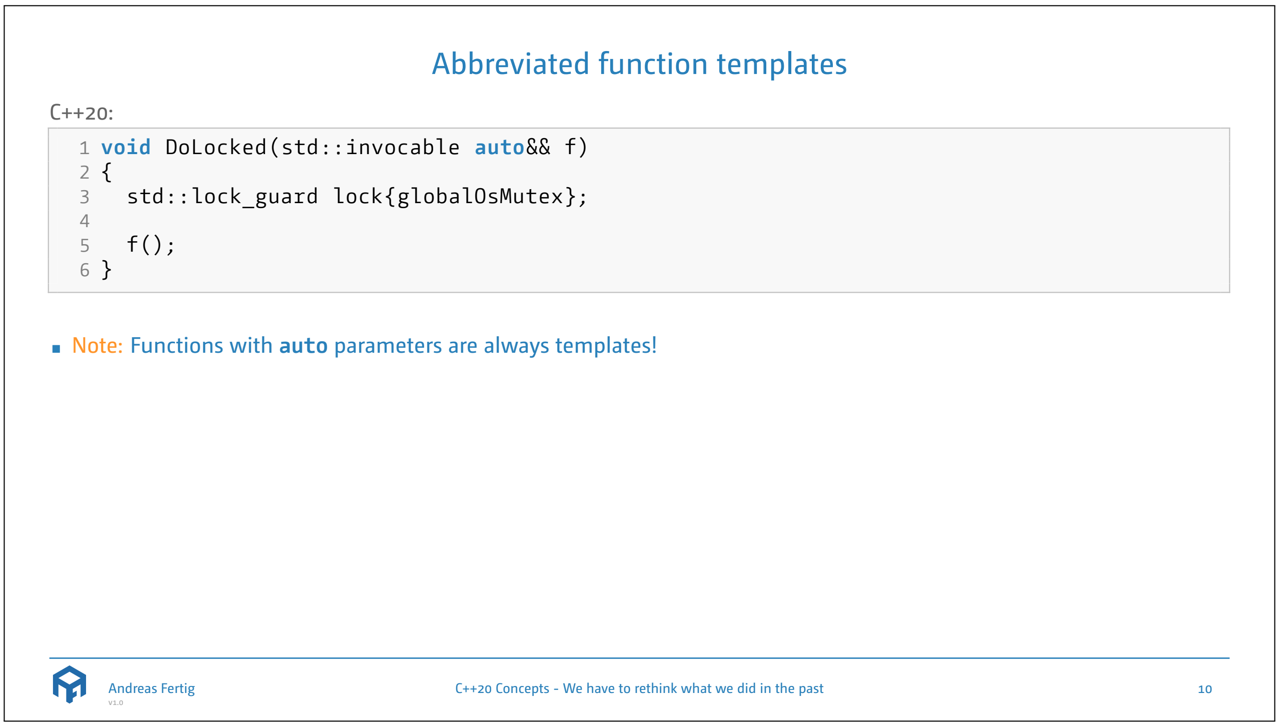
Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 8



## Abbreviated function templates

C++17:

```cpp
template<typename T>
void DoLocked(T&& f)
{
  std::lock_guard lock{globalOsMutex};

  f();
}
```

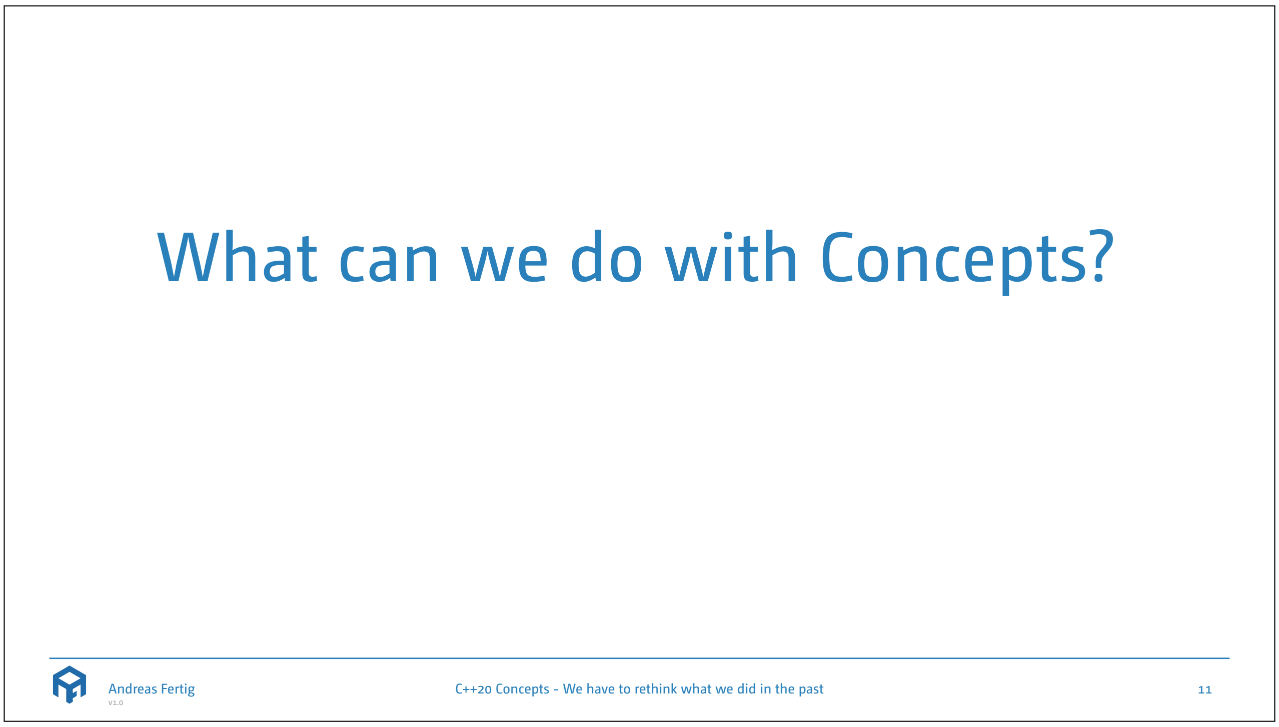


## Abbreviated function templates

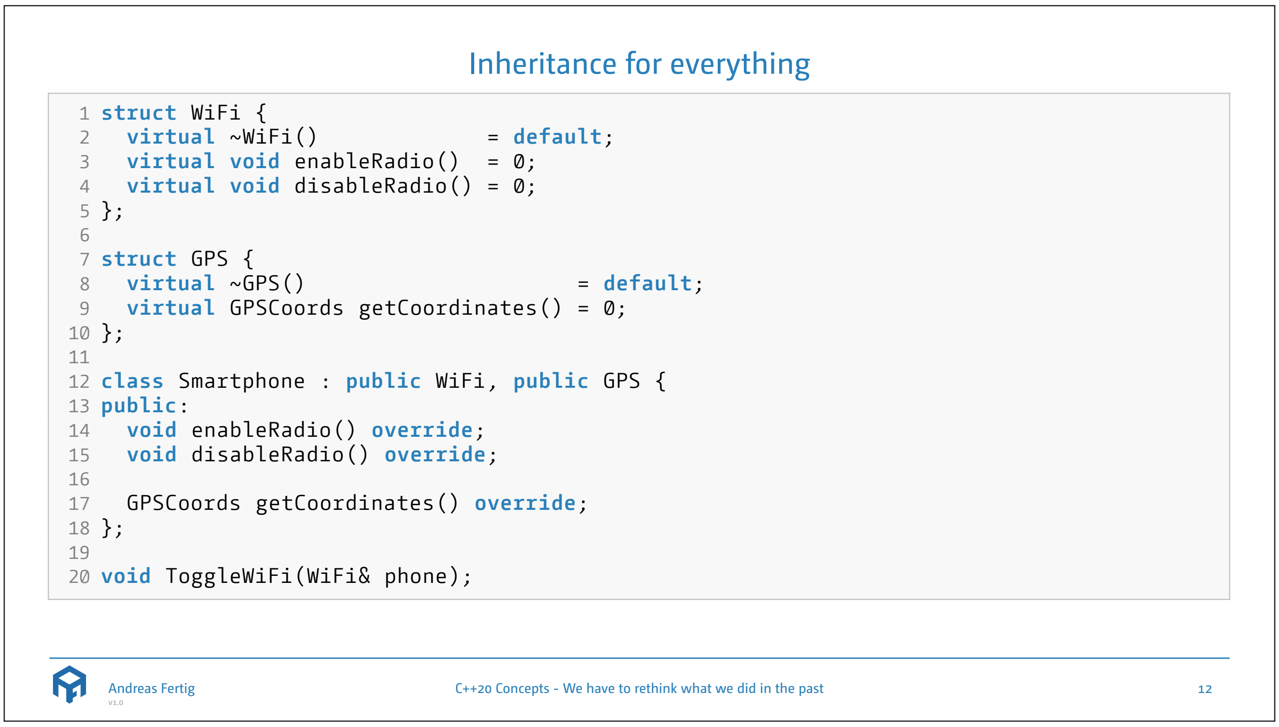
C++20:

```cpp
void DoLocked(std::invocable auto&& f)
{
  std::lock_guard lock{globalOsMutex};

  f();
}
```

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

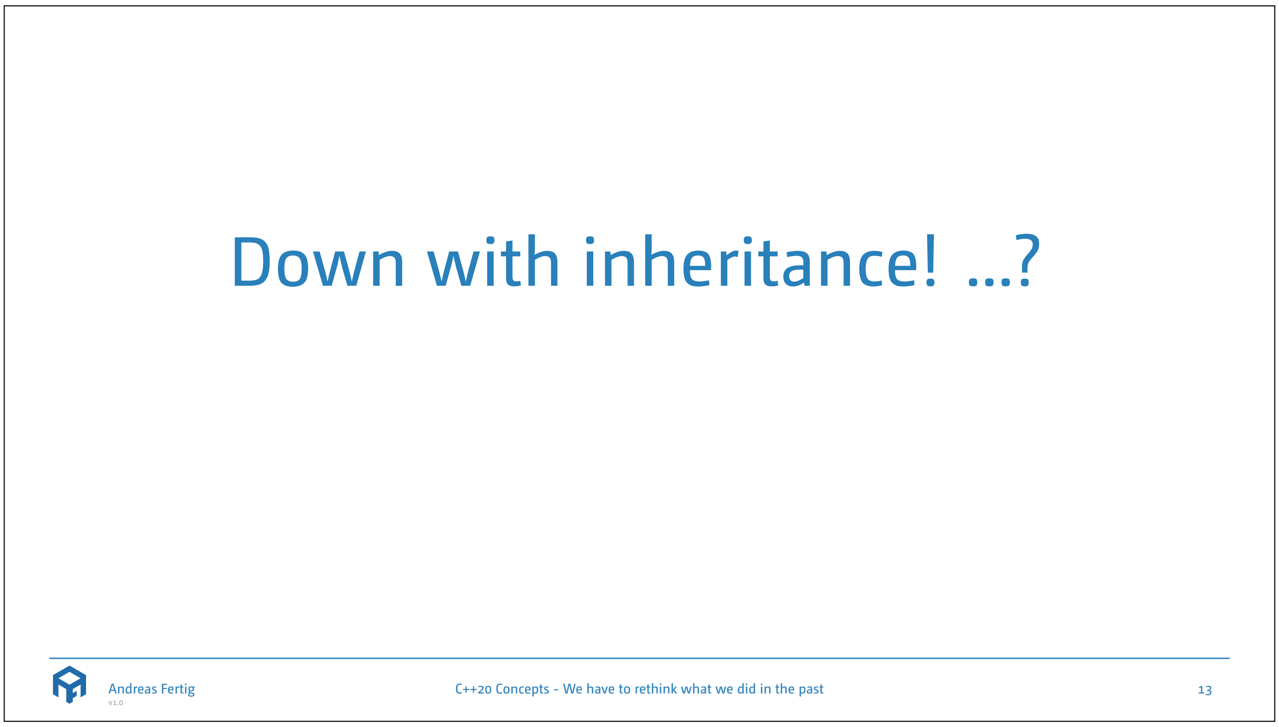


## What can we do with Concepts?



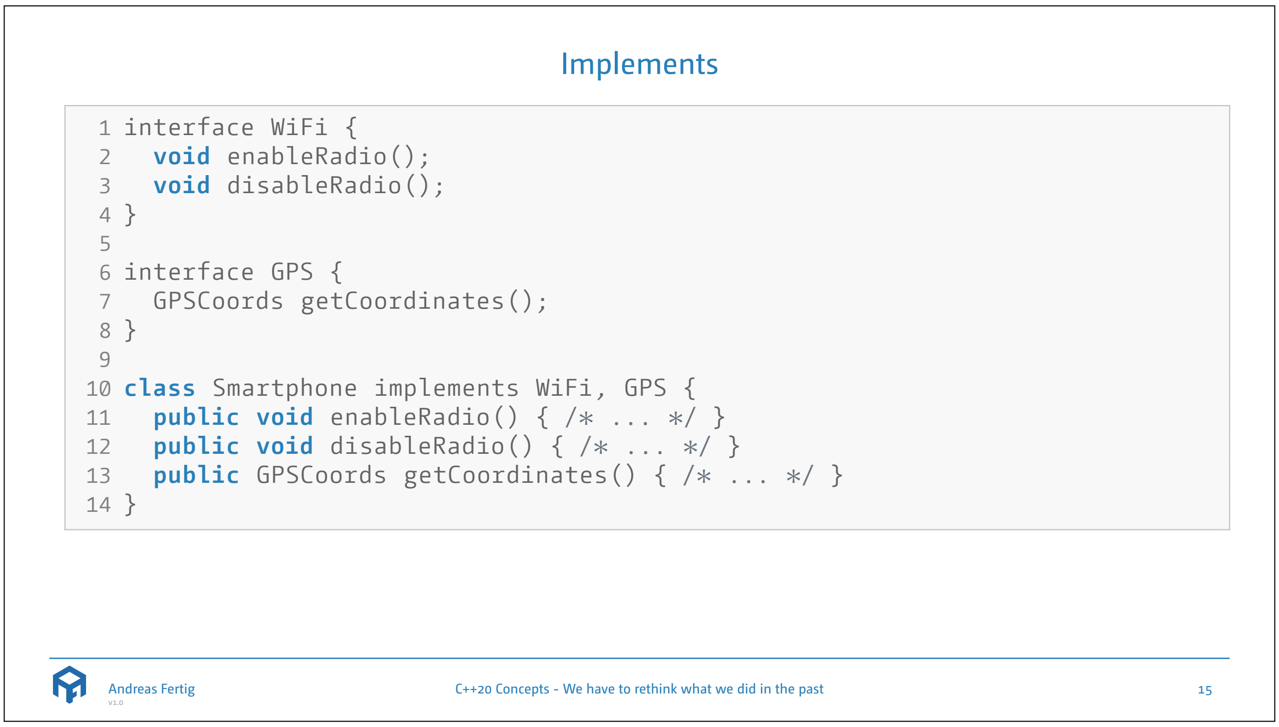
## Inheritance for everything

```cpp
struct WiFi {
  virtual ~WiFi()              = default;
  virtual void enableRadio()   = 0;
  virtual void disableRadio() = 0;
};

struct GPS {
  virtual ~GPS()                       = default;
  virtual GPSCoords getCoordinates() = 0;
};

class Smartphone : public WiFi, public GPS {
public:
  void enableRadio() override;
  void disableRadio() override;

  GPSCoords getCoordinates() override;
};

void ToggleWiFi(WiFi& phone);
```



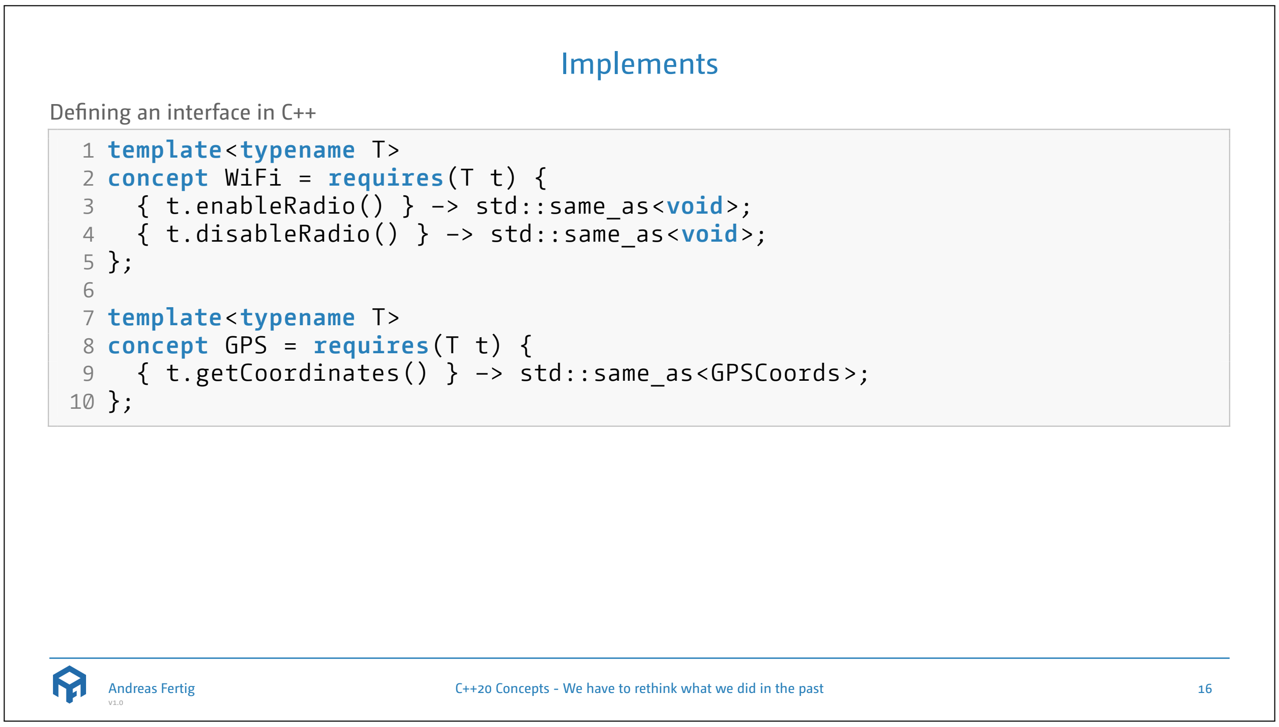
## Down with inheritance! ...?

## We want abstraction not inheritance!



## Implements

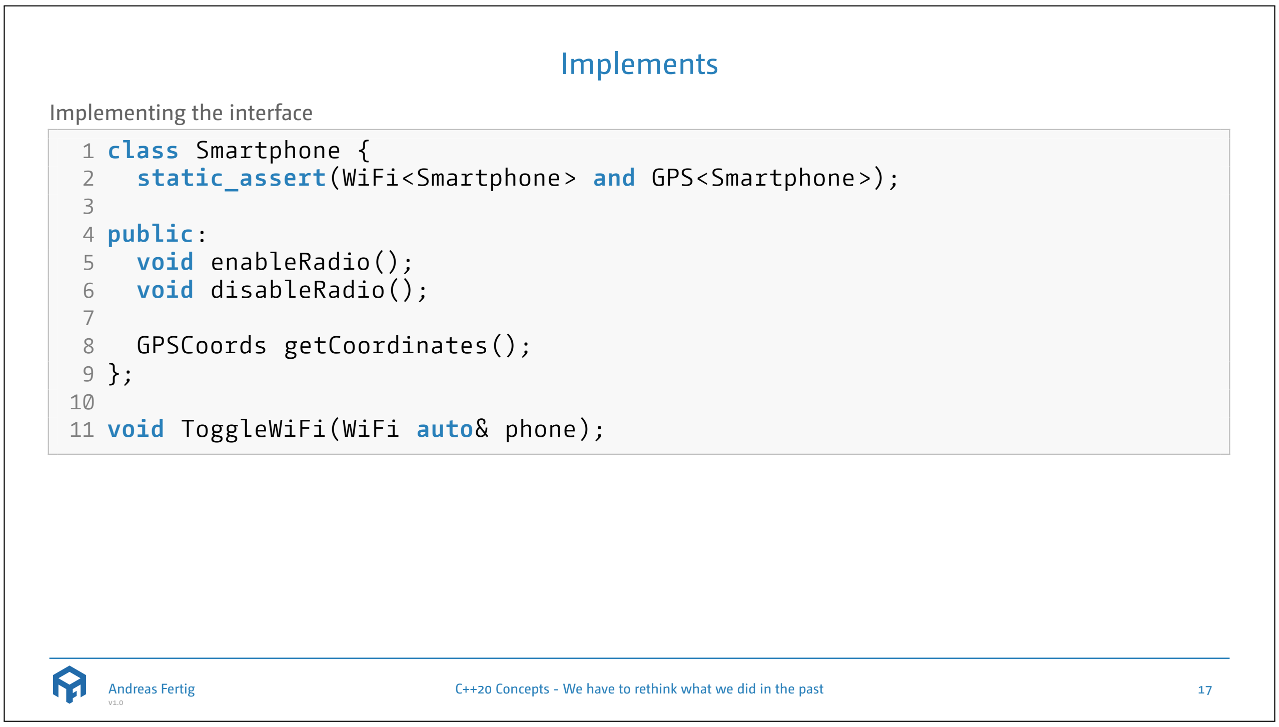
```
interface WiFi {
  void enableRadio();
  void disableRadio();
}

interface GPS {
  GPSCoords getCoordinates();
}

class Smartphone implements WiFi, GPS {
  public void enableRadio() { /* ... */ }
  public void disableRadio() { /* ... */ }
  public GPSCoords getCoordinates() { /* ... */ }
}
```



## Implements

Defining an interface in C++

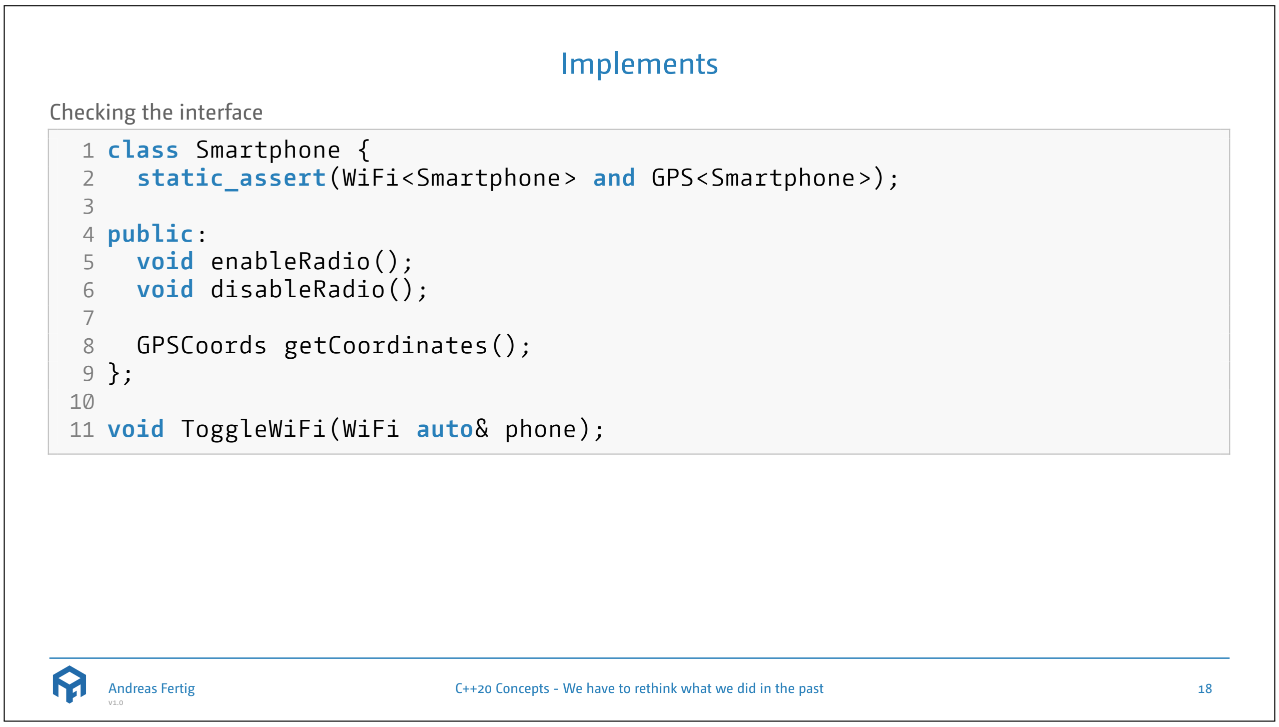
```
template<typename T>
concept WiFi = requires(T t) {
  { t.enableRadio() } -> std::same_as<void>;
  { t.disableRadio() } -> std::same_as<void>;
};

template<typename T>
concept GPS = requires(T t) {
  { t.getCoordinates() } -> std::same_as<GPSCoords>;
};
```



## Implements

Implementing the interface

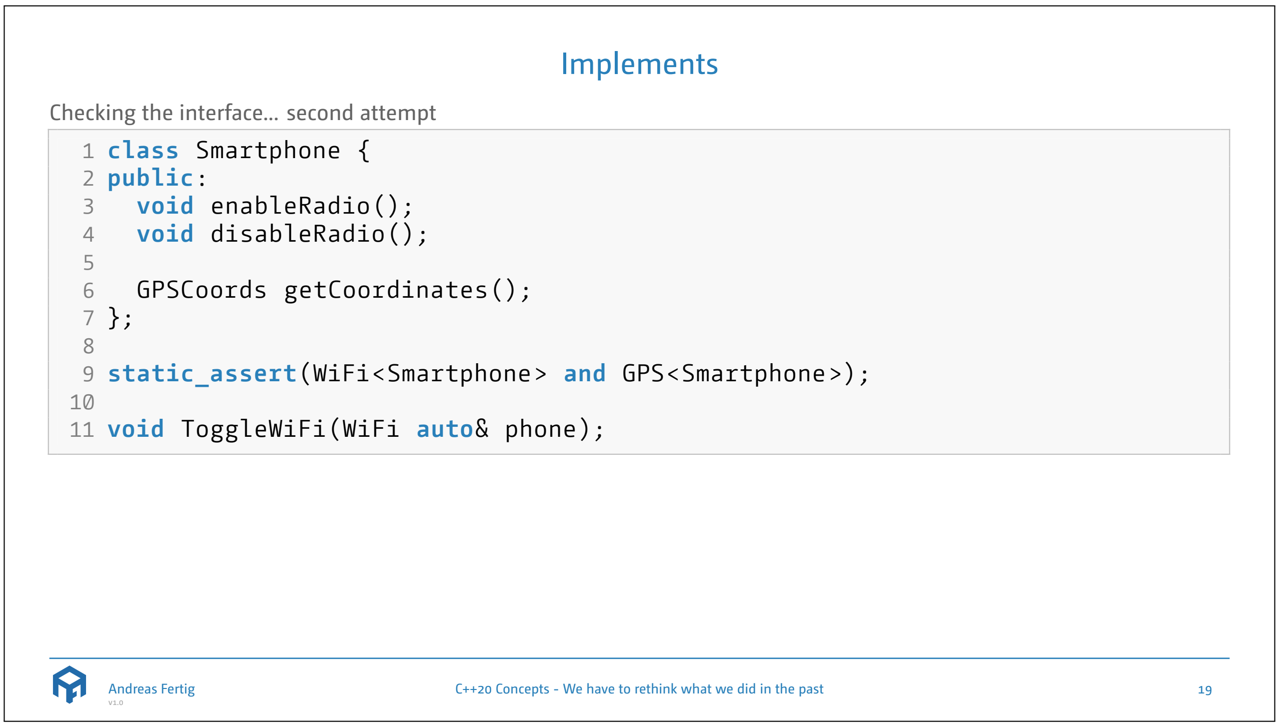
```
class Smartphone {
  static_assert(WiFi<Smartphone> and GPS<Smartphone>);

public:
  void enableRadio();
  void disableRadio();

  GPSCoords getCoordinates();
};

void ToggleWiFi(WiFi auto& phone);
```



## Implements

Checking the interface

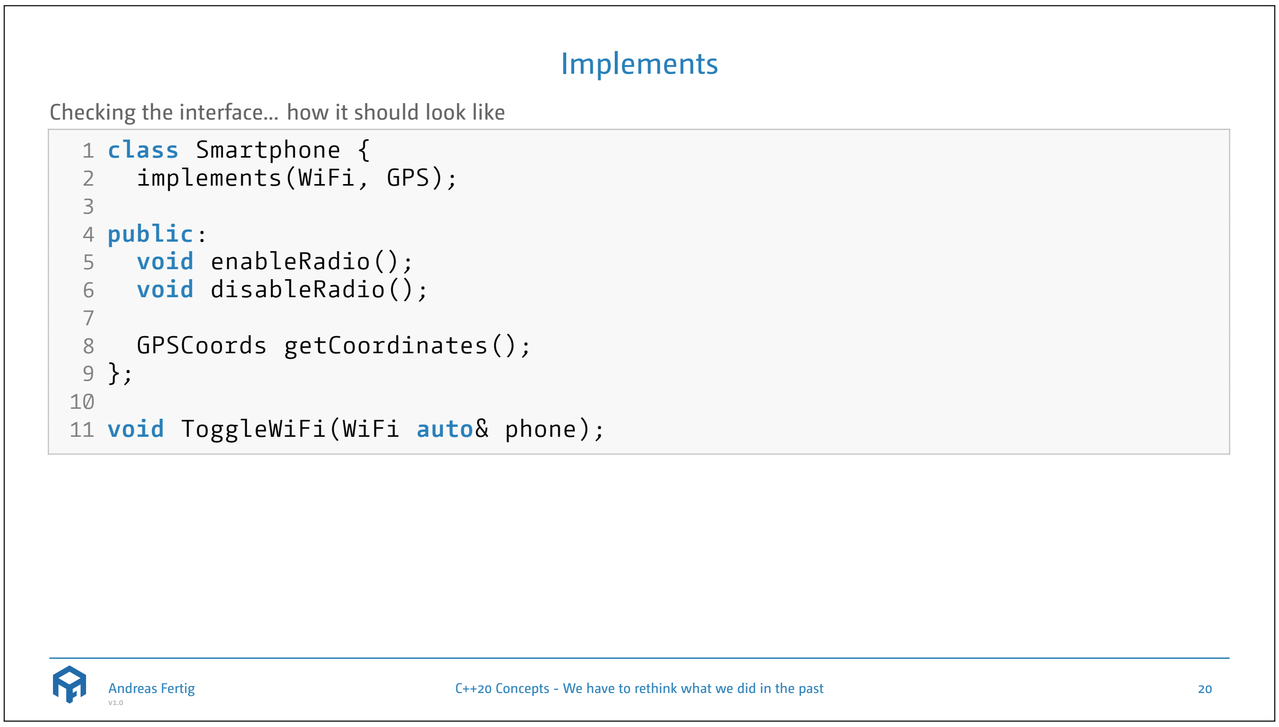
```
class Smartphone {
  static_assert(WiFi<Smartphone> and GPS<Smartphone>);

public:
  void enableRadio();
  void disableRadio();

  GPSCoords getCoordinates();
};

void ToggleWiFi(WiFi auto& phone);
```



## Implements

Checking the interface... second attempt

```cpp
class Smartphone {
public:
  void enableRadio();
  void disableRadio();

  GPSCoords getCoordinates();
};

static_assert(WiFi<Smartphone> and GPS<Smartphone>);

void ToggleWiFi(WiFi auto& phone);
```



## Implements

Checking the interface... how it should look like

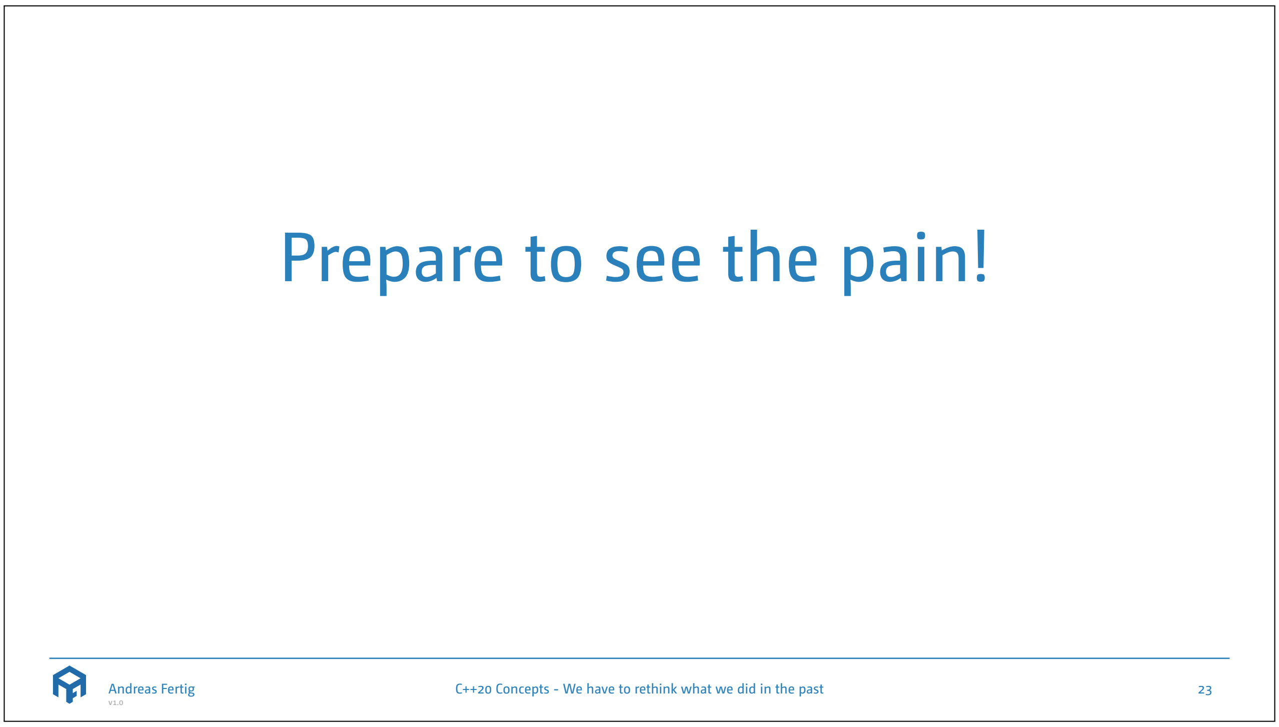
```cpp
class Smartphone {
  implements(WiFi, GPS);

public:
  void enableRadio();
  void disableRadio();

  GPSCoords getCoordinates();
};

void ToggleWiFi(WiFi auto& phone);
```

## Implements

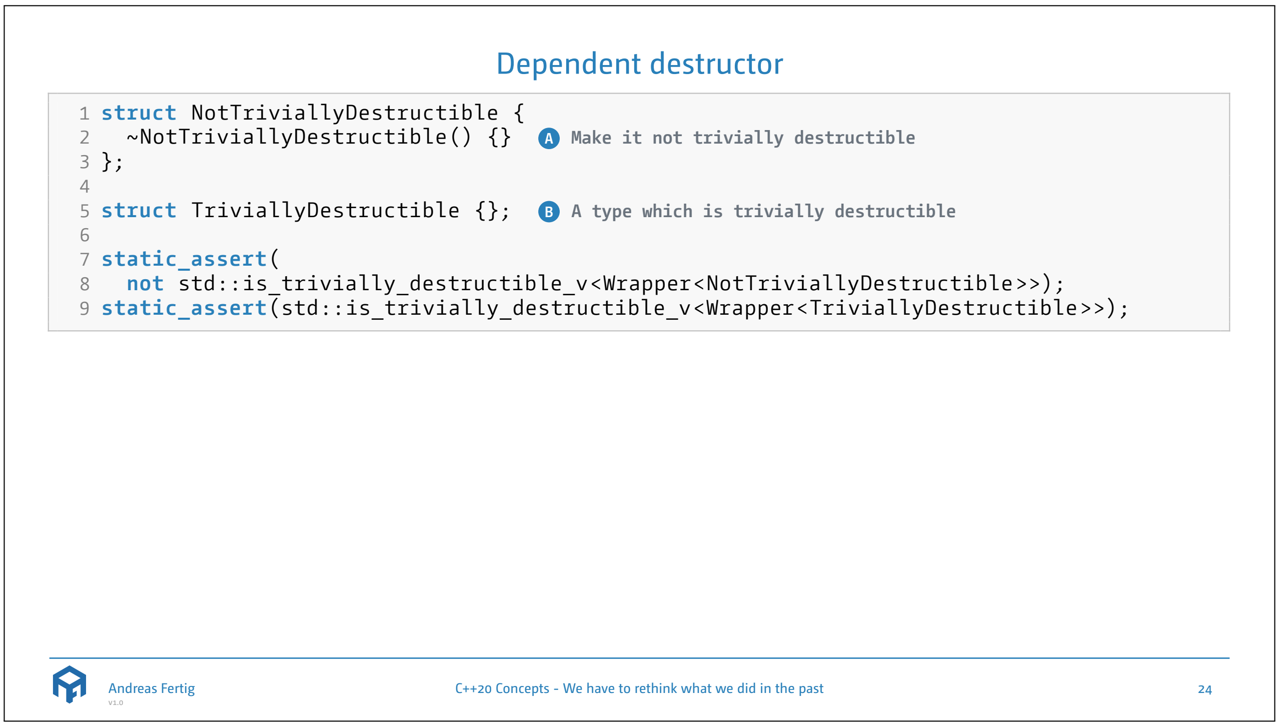
Checking the interface... a solution... appologies for the macro

```
#define implements(name, ...)                                                        \
  consteval void __check_implements() {                                              \
    enum class Helper { name, __VA_ARGS__ __VA_OPT__(, ) Last }; A      \
    B                                                                    \
    auto impl = []<auto N>(auto& self, auto* t, auto*... ts) {                       \
      if constexpr ((sizeof...(ts) + 1) < N) {           C               \
        self.template operator()<N>(self, t, t, ts...);  D               \
      } else {                                                                       \
        E                                                                \
        []<name __VA_OPT__(, ) __VA_ARGS__>() {                                      \
        }.template operator()<std::remove_pointer_t<decltype(t)>,                    \
                              std::remove_pointer_t<decltype(ts)>...>();             \
      }                                                                              \
    };                                                                               \
    F                                                                    \
    impl.template operator()<static_cast<int>(Helper::Last)>(impl, this);            \
  }
```

## You still think inheritance is good?



## Prepare to see the pain!



## Dependent destructor

```
struct NotTriviallyDestructible {
  ~NotTriviallyDestructible() {}   A Make it not trivially destructible
};

struct TriviallyDestructible {};   B A type which is trivially destructible

static_assert(
  not std::is_trivially_destructible_v<Wrapper<NotTriviallyDestructible>>);
static_assert(std::is_trivially_destructible_v<Wrapper<TriviallyDestructible>>);
```

## Dependent destructor

```cpp
template<typename T>
class Wrapper {
  union {
    char __null_state_;
    T    mData;
  };

public:
  ~Wrapper() requires(not std::is_trivially_destructible_v<T>) { mData.~T(); }
  ~Wrapper() = default;
};
```

## 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

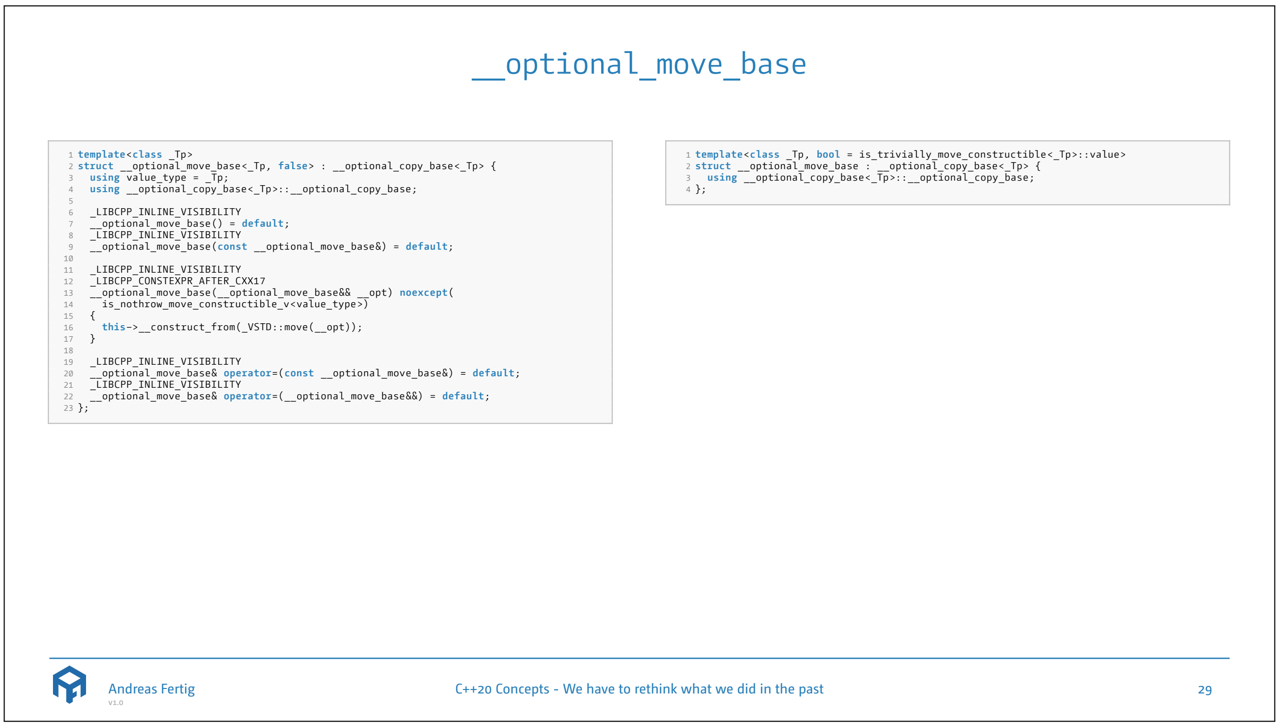
```cpp
template<class _Tp>
struct __optional_destruct_base<_Tp, false> {
  typedef _Tp value_type;
  static_assert(is_object_v<value_type>,
                "instantiation of optional with a non-object type "
                "is undefined behavior");
  union {
    char        __null_state_;
    value_type __val_;
  };
  bool __engaged_;

  _LIBCPP_CONSTEXPR_AFTER_CXX17 ~__optional_destruct_base()
  {
    if (__engaged_) __val_.~value_type();
  }

  constexpr __optional_destruct_base() noexcept : __null_state_(), __engaged_(false)
  {}

  template<class... _Args>
  constexpr explicit __optional_destruct_base(in_place_t, _Args&&... __args)
  : __val_(_VSTD::forward<_Args>(__args)...), __engaged_(true)
  {}

  template<class _Fp, class... _Args>
  _LIBCPP_HIDE_FROM_ABI constexpr __optional_destruct_base(
    __optional_construct_from_invoke_tag,
    _Fp&& __f,
    _Args&&... __args)
  : __val_(_VSTD::invoke(_VSTD::forward<_Fp>(__f), _VSTD::forward<_Args>(__args)...))
  , __engaged_(true)
  {}

  _LIBCPP_CONSTEXPR_AFTER_CXX17 void reset() noexcept
  {
    if (__engaged_) {
      __val_.~value_type();
      __engaged_ = false;
    }
  }
};
```

```cpp
template<class _Tp>
struct __optional_destruct_base<_Tp, true> {
  typedef _Tp value_type;
  static_assert(is_object_v<value_type>,
                "instantiation of optional with a non-object type "
                "is undefined behavior");
  union {
    char        __null_state_;
    value_type __val_;
  };
  bool __engaged_;

  constexpr __optional_destruct_base() noexcept : __null_state_(), __engaged_(false)
  {}

  template<class... _Args>
  constexpr explicit __optional_destruct_base(in_place_t, _Args&&... __args)
  : __val_(_VSTD::forward<_Args>(__args)...), __engaged_(true)
  {}

  template<class _Fp, class... _Args>
  _LIBCPP_HIDE_FROM_ABI constexpr __optional_destruct_base(
    __optional_construct_from_invoke_tag,
    _Fp&& __f,
    _Args&&... __args)
  : __val_(_VSTD::invoke(_VSTD::forward<_Fp>(__f), _VSTD::forward<_Args>(__args)...))
  , __engaged_(true)
  {}

  _LIBCPP_CONSTEXPR_AFTER_CXX17 void reset() noexcept
  {
    if (__engaged_) { __engaged_ = false; }
  }
};
```

## __optional_copy_base

```cpp
template<class _Tp>
struct __optional_copy_base<_Tp, false> : __optional_storage_base<_Tp> {
  using __optional_storage_base<_Tp>::__optional_storage_base;

  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_base() = default;

  _LIBCPP_INLINE_VISIBILITY
  _LIBCPP_CONSTEXPR_AFTER_CXX17
  __optional_copy_base(const __optional_copy_base& __opt)
  {
    this->__construct_from(__opt);
  }

  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_base(__optional_copy_base&&) = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_base& operator=(const __optional_copy_base&) = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_base& operator=(__optional_copy_base&&) = default;
};
```

```cpp
template<class _Tp, bool = is_trivially_copy_constructible<_Tp>::value>
struct __optional_copy_base : __optional_storage_base<_Tp> {
  using __optional_storage_base<_Tp>::__optional_storage_base;
};
```



## __optional_move_base

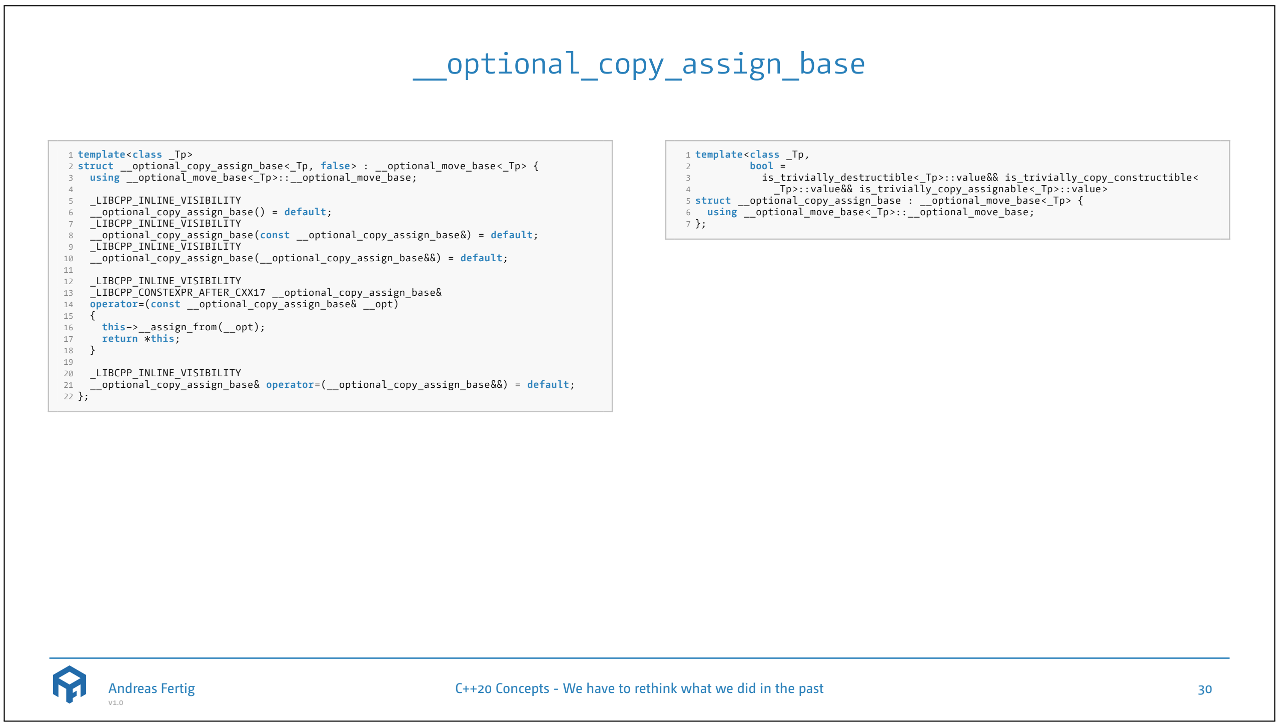
```cpp
template<class _Tp>
struct __optional_move_base<_Tp, false> : __optional_copy_base<_Tp> {
  using value_type = _Tp;
  using __optional_copy_base<_Tp>::__optional_copy_base;

  _LIBCPP_INLINE_VISIBILITY
  __optional_move_base() = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_move_base(const __optional_move_base&) = default;

  _LIBCPP_INLINE_VISIBILITY
  _LIBCPP_CONSTEXPR_AFTER_CXX17
  __optional_move_base(__optional_move_base&& __opt) noexcept(
    is_nothrow_move_constructible_v<value_type>)
  {
    this->__construct_from(_VSTD::move(__opt));
  }

  _LIBCPP_INLINE_VISIBILITY
  __optional_move_base& operator=(const __optional_move_base&) = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_move_base& operator=(__optional_move_base&&) = default;
};
```

```cpp
template<class _Tp, bool = is_trivially_move_constructible<_Tp>::value>
struct __optional_move_base : __optional_copy_base<_Tp> {
  using __optional_copy_base<_Tp>::__optional_copy_base;
};
```

Andreas Fertig
v1.0

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

29



## __optional_copy_assign_base

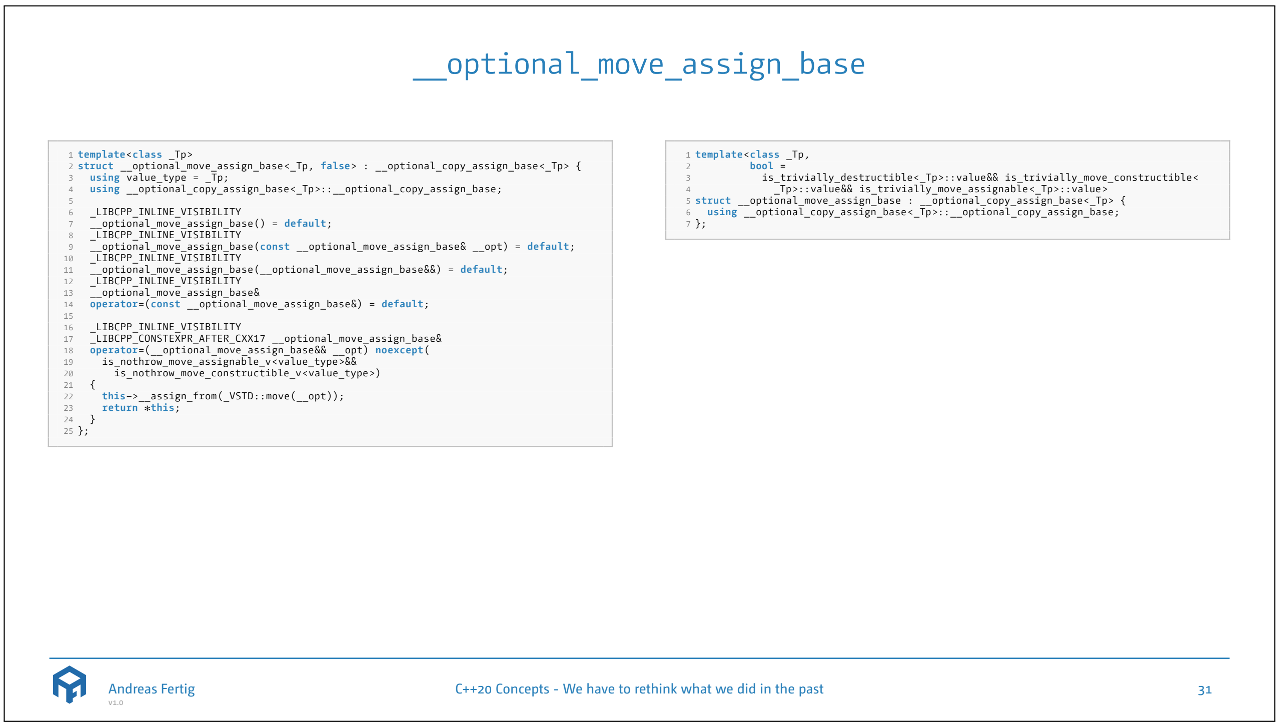
```cpp
template<class _Tp>
struct __optional_copy_assign_base<_Tp, false> : __optional_move_base<_Tp> {
  using __optional_move_base<_Tp>::__optional_move_base;

  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_assign_base() = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_assign_base(const __optional_copy_assign_base&) = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_assign_base(__optional_copy_assign_base&&) = default;

  _LIBCPP_INLINE_VISIBILITY
  _LIBCPP_CONSTEXPR_AFTER_CXX17 __optional_copy_assign_base&
  operator=(const __optional_copy_assign_base& __opt)
  {
    this->__assign_from(__opt);
    return *this;
  }

  _LIBCPP_INLINE_VISIBILITY
  __optional_copy_assign_base& operator=(__optional_copy_assign_base&&) = default;
};
```

```cpp
template<class _Tp,
         bool =
           is_trivially_destructible<_Tp>::value&& is_trivially_copy_constructible<
             _Tp>::value&& is_trivially_copy_assignable<_Tp>::value>
struct __optional_copy_assign_base : __optional_move_base<_Tp> {
  using __optional_move_base<_Tp>::__optional_move_base;
};
```

Andreas Fertig
v1.0

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

30

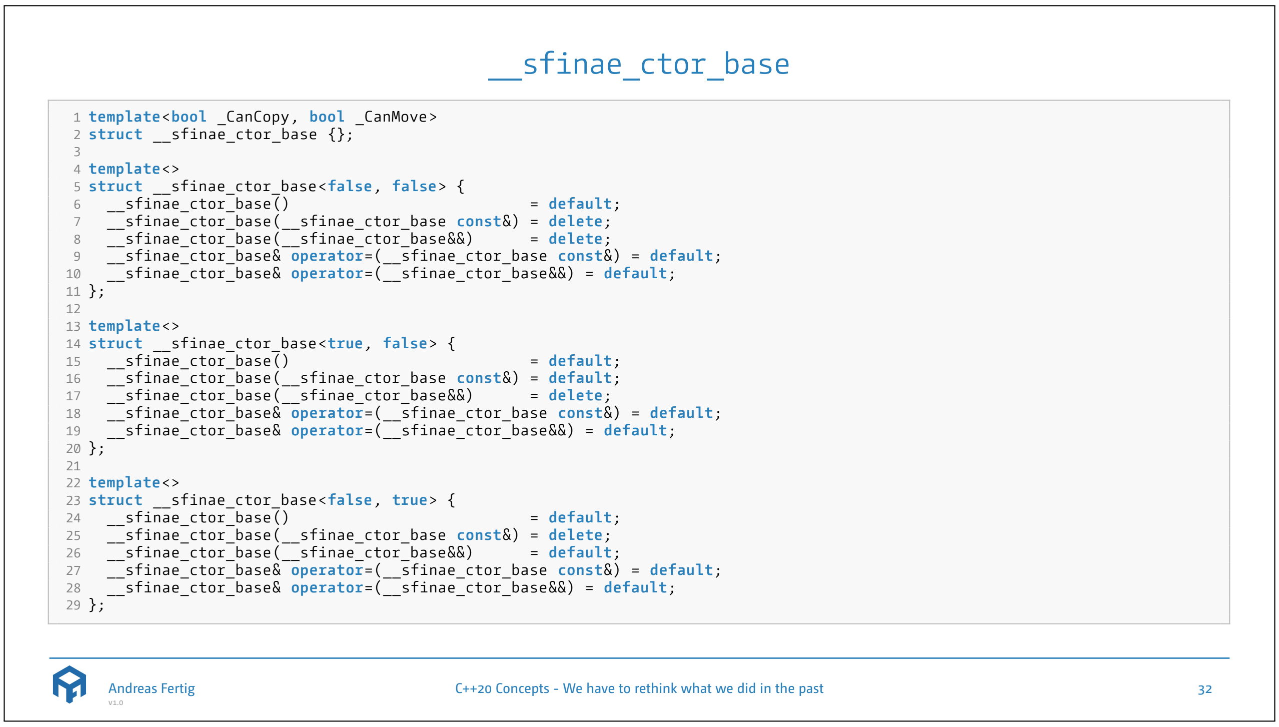


## __optional_move_assign_base

```cpp
template<class _Tp>
struct __optional_move_assign_base<_Tp, false> : __optional_copy_assign_base<_Tp> {
  using value_type = _Tp;
  using __optional_copy_assign_base<_Tp>::__optional_copy_assign_base;

  _LIBCPP_INLINE_VISIBILITY
  __optional_move_assign_base() = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_move_assign_base(const __optional_move_assign_base& __opt) = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_move_assign_base(__optional_move_assign_base&&) = default;
  _LIBCPP_INLINE_VISIBILITY
  __optional_move_assign_base&
  operator=(const __optional_move_assign_base&) = default;

  _LIBCPP_INLINE_VISIBILITY
  _LIBCPP_CONSTEXPR_AFTER_CXX17 __optional_move_assign_base&
  operator=(__optional_move_assign_base&& __opt) noexcept(
    is_nothrow_move_assignable_v<value_type>&&
      is_nothrow_move_constructible_v<value_type>)
  {
    this->__assign_from(_VSTD::move(__opt));
    return *this;
  }
};
```

```cpp
template<class _Tp,
         bool =
           is_trivially_destructible<_Tp>::value&& is_trivially_move_constructible<
             _Tp>::value&& is_trivially_move_assignable<_Tp>::value>
struct __optional_move_assign_base : __optional_copy_assign_base<_Tp> {
  using __optional_copy_assign_base<_Tp>::__optional_copy_assign_base;
};
```

Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 31



## __sfinae_ctor_base

```cpp
template<bool _CanCopy, bool _CanMove>
struct __sfinae_ctor_base {};

template<>
struct __sfinae_ctor_base<false, false> {
  __sfinae_ctor_base()                             = default;
  __sfinae_ctor_base(__sfinae_ctor_base const&) = delete;
  __sfinae_ctor_base(__sfinae_ctor_base&&)      = delete;
  __sfinae_ctor_base& operator=(__sfinae_ctor_base const&) = default;
  __sfinae_ctor_base& operator=(__sfinae_ctor_base&&) = default;
};

template<>
struct __sfinae_ctor_base<true, false> {
  __sfinae_ctor_base()                             = default;
  __sfinae_ctor_base(__sfinae_ctor_base const&) = default;
  __sfinae_ctor_base(__sfinae_ctor_base&&)      = delete;
  __sfinae_ctor_base& operator=(__sfinae_ctor_base const&) = default;
  __sfinae_ctor_base& operator=(__sfinae_ctor_base&&) = default;
};

template<>
struct __sfinae_ctor_base<false, true> {
  __sfinae_ctor_base()                             = default;
  __sfinae_ctor_base(__sfinae_ctor_base const&) = delete;
  __sfinae_ctor_base(__sfinae_ctor_base&&)      = default;
  __sfinae_ctor_base& operator=(__sfinae_ctor_base const&) = default;
  __sfinae_ctor_base& operator=(__sfinae_ctor_base&&) = default;
};
```

Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 32

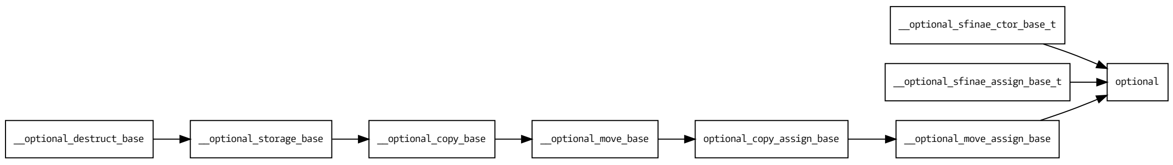
## `__sfinae_assign_base`

```cpp
template<bool _CanCopy, bool _CanMove>
struct __sfinae_assign_base {};

template<>
struct __sfinae_assign_base<false, false> {
  __sfinae_assign_base()                            = default;
  __sfinae_assign_base(__sfinae_assign_base const&) = default;
  __sfinae_assign_base(__sfinae_assign_base&&)      = default;
  __sfinae_assign_base& operator=(__sfinae_assign_base const&) = delete;
  __sfinae_assign_base& operator=(__sfinae_assign_base&&) = delete;
};

template<>
struct __sfinae_assign_base<true, false> {
  __sfinae_assign_base()                            = default;
  __sfinae_assign_base(__sfinae_assign_base const&) = default;
  __sfinae_assign_base(__sfinae_assign_base&&)      = default;
  __sfinae_assign_base& operator=(__sfinae_assign_base const&) = default;
  __sfinae_assign_base& operator=(__sfinae_assign_base&&) = delete;
};

template<>
struct __sfinae_assign_base<false, true> {
  __sfinae_assign_base()                            = default;
  __sfinae_assign_base(__sfinae_assign_base const&) = default;
  __sfinae_assign_base(__sfinae_assign_base&&)      = default;
  __sfinae_assign_base& operator=(__sfinae_assign_base const&) = delete;
  __sfinae_assign_base& operator=(__sfinae_assign_base&&) = default;
};
```

## Finally... `std::optional`

```cpp
template<class _Tp>
class optional : private __optional_move_assign_base<_Tp>,
                 private __optional_sfinae_ctor_base_t<_Tp>,
                 private __optional_sfinae_assign_base_t<_Tp> {
  using __base = __optional_move_assign_base<_Tp>;

  // ...

  constexpr optional() noexcept {}
  constexpr optional(const optional&) = default;
  constexpr optional(optional&&)      = default;
  constexpr optional(nullopt_t) noexcept {}

  // ...

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

  // other optional operations
}
```

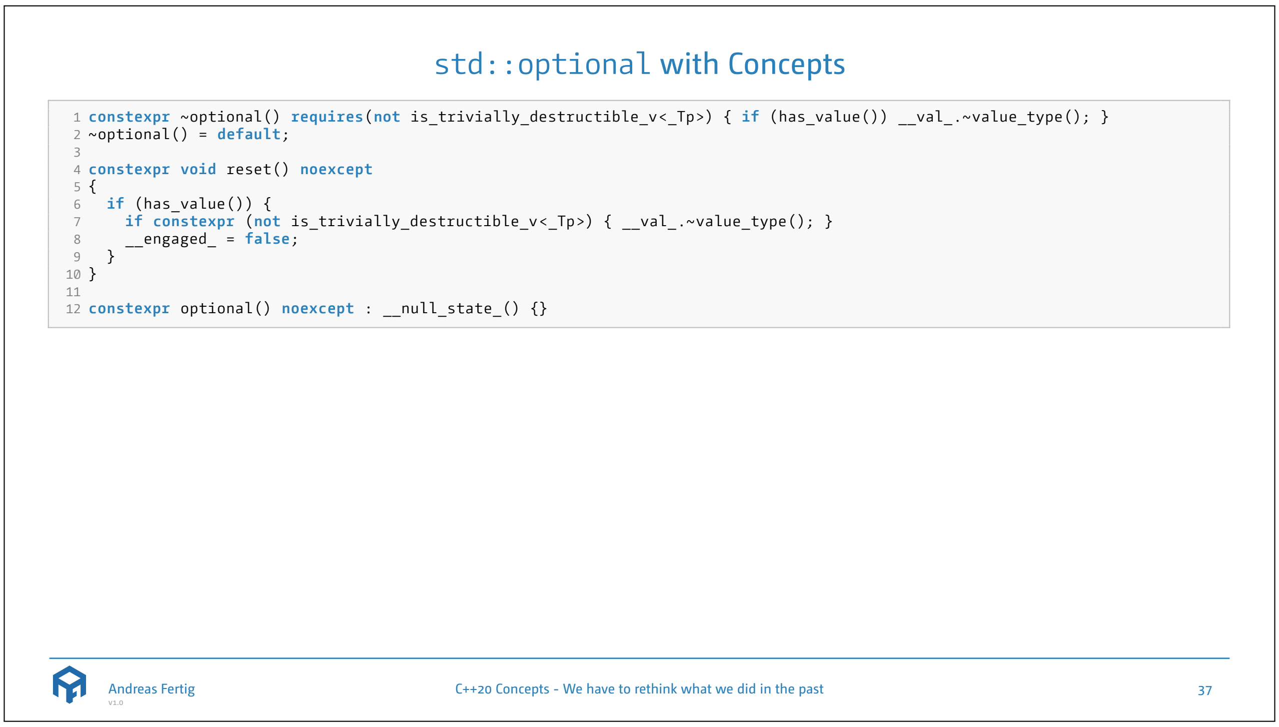
## The Power of Concepts - `std::optional`



__optional_sfinae_ctor_base_t
__optional_sfinae_assign_base_t
optional
__optional_destruct_base
__optional_storage_base
__optional_copy_base
__optional_move_base
optional_copy_assign_base
__optional_move_assign_base

## `std::optional` with Concepts

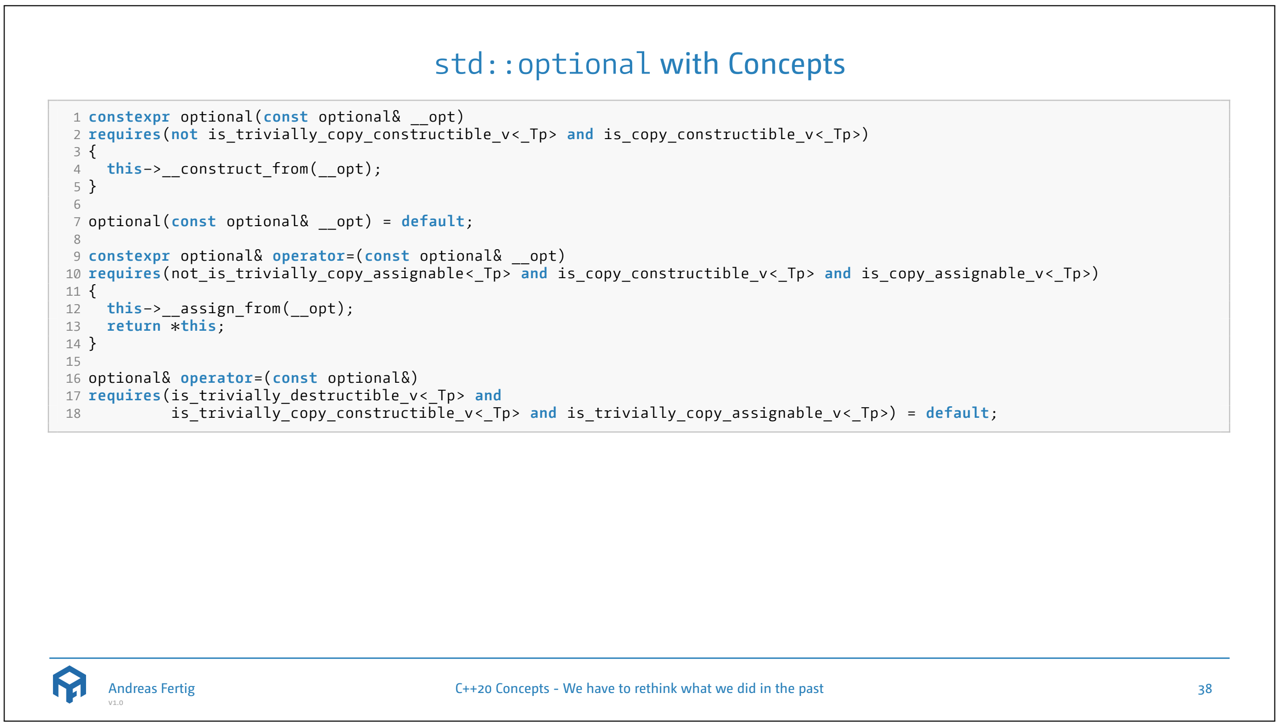
```cpp
template<class _Tp>
class optional {
public:
  using value_type = _Tp;

private:
  static_assert(
    is_object_v<value_type>,
    "instantiation of optional with a non-object type is undefined behavior");
  union {
    char        __null_state_;
    value_type __val_;
  };
  bool __engaged_{false};

public:
  // ctor / dtor
  // copy-operations
  // move-operations
  //
  // other optional operations
};
```



## std::optional with Concepts

```cpp
constexpr ~optional() requires(not is_trivially_destructible_v<_Tp>) { if (has_value()) __val_.~value_type(); }
~optional() = default;

constexpr void reset() noexcept
{
  if (has_value()) {
    if constexpr (not is_trivially_destructible_v<_Tp>) { __val_.~value_type(); }
    __engaged_ = false;
  }
}

constexpr optional() noexcept : __null_state_() {}
```

Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 37



## std::optional with Concepts

```cpp
constexpr optional(const optional& __opt)
requires(not is_trivially_copy_constructible_v<_Tp> and is_copy_constructible_v<_Tp>)
{
  this->__construct_from(__opt);
}

optional(const optional& __opt) = default;

constexpr optional& operator=(const optional& __opt)
requires(not_is_trivially_copy_assignable<_Tp> and is_copy_constructible_v<_Tp> and is_copy_assignable_v<_Tp>)
{
  this->__assign_from(__opt);
  return *this;
}

optional& operator=(const optional&)
requires(is_trivially_destructible_v<_Tp> and
         is_trivially_copy_constructible_v<_Tp> and is_trivially_copy_assignable_v<_Tp>) = default;
```

Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 38

## `std::optional` with Concepts

```
constexpr optional(optional&& __opt) noexcept(is_nothrow_move_constructible_v<value_type>)
requires(not is_trivially_move_constructible_v<_Tp> and is_move_constructible_v<_Tp>)
{
  this->__construct_from(_VSTD::move(__opt));
}

optional(optional&&) = default;

constexpr optional& operator=(optional&& __opt)
noexcept(is_nothrow_move_assignable_v<value_type> and is_nothrow_move_constructible_v<value_type>)
requires(not_is_trivially_move_assignable<_Tp> and is_move_constructible_v<_Tp> and is_move_assignable_v<_Tp>)
{
  this->__assign_from(_VSTD::move(__opt));
  return *this;
}

optional& operator=(optional&&)
requires(is_trivially_destructible_v<_Tp> and
         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] | $463ms$ | $391ms$ |

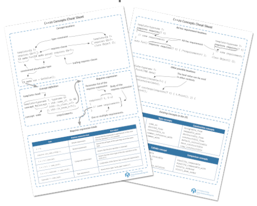
```
#include <optional>

struct NotTriviallyDestructible {
  ~NotTriviallyDestructible() {}
};

int main()
{
  std::optional<int>                      o{};
  std::optional<NotTriviallyDestructible> o2{};
}
```

[1] Not counting `__optional_storage_base`
[2] Formatted with clang-format
[3] MacMini 2018, 3,2 GHz 6-Core Intel Core i7, 64 GB

}

I am Fertig.

C++20 Concepts Cheat Sheet



fertig.to/subscribe

## 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/

## References

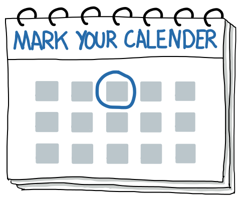
**Images:**
44: Franziska Panter

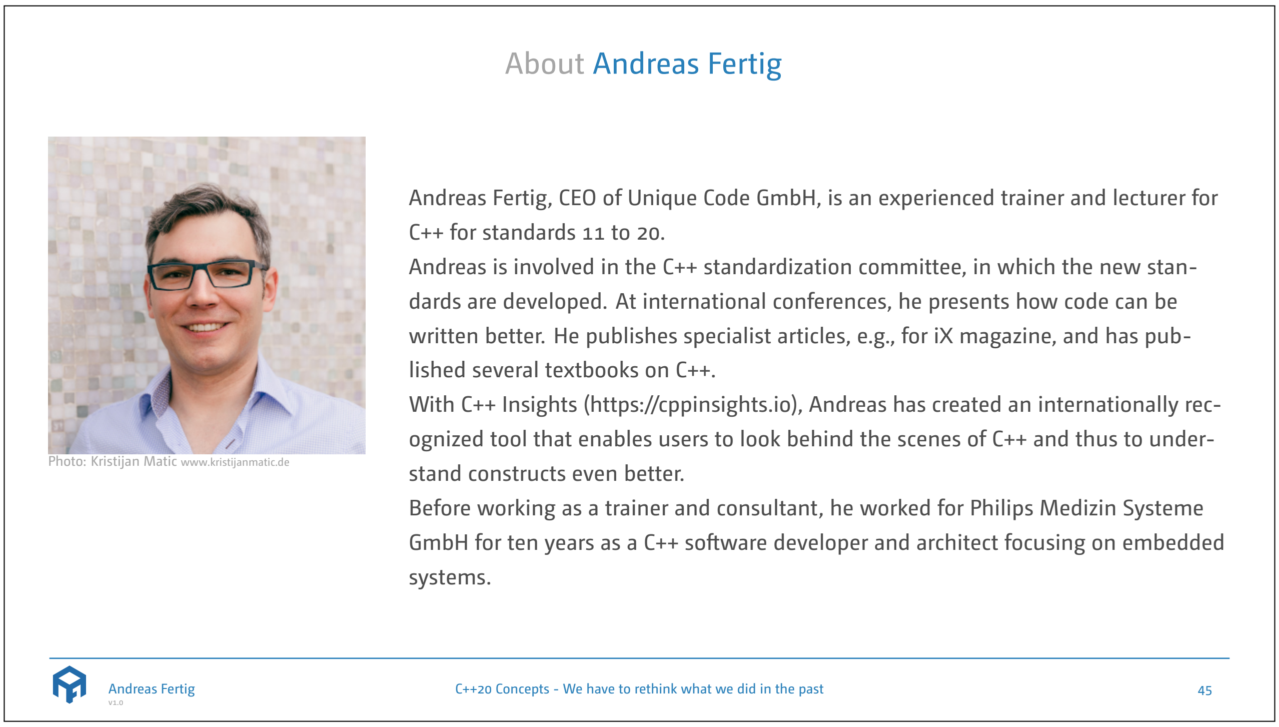
## Upcoming Events

**Training Classes**

- *Programming with C++20*, Andreas Fertig, July 11 - 15
- *Programmieren mit C++20*, Andreas Fertig, October 05 - 07
- *C++1x for Emebedded 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/.





## 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.

Andreas Fertig v1.0 C++20 Concepts - We have to rethink what we did in the past 45