Templates Made Easy with C++ 20

Meeting C++ 2024

Roth Michaels
Principal Software Engineer
Native Instruments

#include <c++>

https://www.includecpp.org

You can do it!

NATIVE INSTRUMENTS

iZOTOPE

Plugin Alliance

BRAINWORX

Bearbeiten
Funktionen
Ansicht
Varispeed
±0.00%
120,0000
4/4
/16
Kein In
Kein Out
Einrasten: Intelligent
Verschieben: X-Fade
Norweg...d_audio
Inst 1
Vögel
Wind_Audio
Dub chords
Inst 2
Inst 3
Inst 4
Dub ch...alftime
Molek Melodies
Bongos...Merged
Bongos plus 100
Low Ris...ls pings
drums
Big 808
808 Bass
mm wave weaver
norwegi...S JAZZ
norwegi...o spirit
pads
Inst 3
Manuell
Vergleichen
Widerrufen
Wiederholen
Ansicht:
Editor
7 SKIES, Magnificence - THE DRILL
FILTER
CUTOFF
RESO
ENV AMT
THE DRILL
FLT ENV
ATTACK
DECAY
SUSTAIN
RELEASE
AMP
LAYER 1
LAYER 2
AMP ENV
Inst 4
gr808
808 Center
Mode
[mono]
808 Sides
Panning
[attack: fast]
[releaseeeeee]
Reverb: [Cloud Grain]
Size
Amount:
Rev Lo-Cut:
Rev Hi-Cut:
Presets:
1
2
3
Clay and Kelsy
Bouncy FX: [off]
Speed ModW:
gr808
kick
Kontakt
Inst 2
MS20 Drums
Cutoff
Saturation
Reverb
Delay
Delay Time
Peter Flint:
MS20 Drums
Kontakt

Why I wanted to give this talk

Why I wanted to give this talk

My favorite interview question...

I could learn more about...

- Templates
- Template metaprogramming
- Partial Template Specialization
- SFINAE (substitution failure is not an error)

Why I wanted to give this talk

Excited for what I don’t need to teach anymore…

Why I wanted to give this talk

Update your compilers!

Generic programming with templates...

...and the journey to **`constexpr`**

Generic programming with templates

```
auto msg = std::string{"hello, world"};
auto it = std::find(begin(msg), end(msg), ',');
```

Generic programming with templates

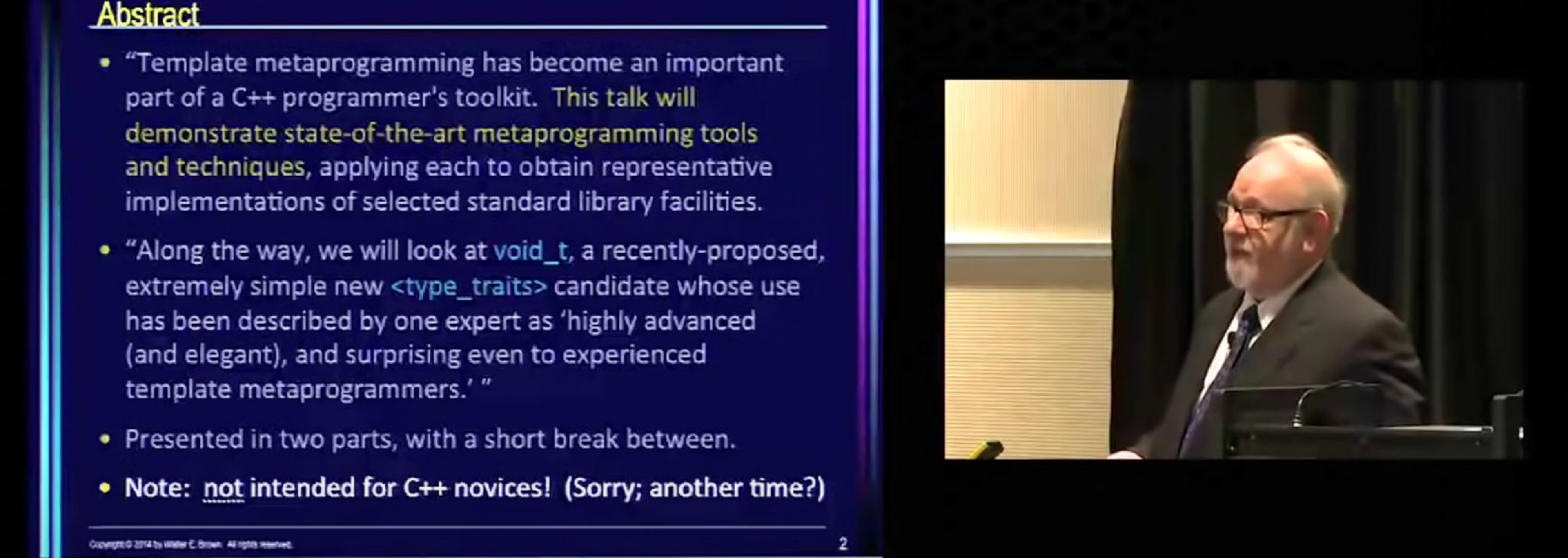
```
auto msg = std::string{"hello, world"};
auto it = std::find(begin(msg), end(msg), ',');

auto v = std::vector{1, 3, 42, 0, 50};
auto it = std::find(begin(v), end(v), 42);
```

Generic programming with templates

```
auto msg = std::string{"hello, world"};
auto it = std::find<char>(begin(msg), end(msg),',');

auto v = std::vector{1, 3, 42, 0, 50};
auto it = std::find<int>(begin(v), end(v), 42);
```

Turing complete templates

Discovered by accident



https://www.youtube.com/watch?v=Am2is2QCvxY

Modern Template Metaprogramming (Walter Brown)

Fibonacci Metaprogram

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

```
template <unsigned N> struct Fibonacci {
    static_assert(N > 1);
    static constexpr auto value =
        Fibonacci<N - 2>::value + Fibonacci<N - 1>::value;
};
template <> struct Fibonacci<1> {
    static constexpr unsigned value = 1u;
};
template <> struct Fibonacci<0> {
    static constexpr unsigned value = 0u;
};
static_assert(Fibonacci<6u>::value == 8u);
```

```cpp
template <unsigned N> struct Fibonacci {
    static_assert(N > 1);
    static constexpr auto value =
        Fibonacci<N - 2>::value + Fibonacci<N - 1>::value;
};
template <> struct Fibonacci<1> {
    static constexpr unsigned value = 1u;
};
template <> struct Fibonacci<0> {
    static constexpr unsigned value = 0u;
};
static_assert(Fibonacci<6u>::value == 8u);
```

```cpp
template <unsigned N> struct Fibonacci {
    static_assert(N > 1);
    static constexpr auto value =
        Fibonacci<N - 2>::value + Fibonacci<N - 1>::value;
};
template <> struct Fibonacci<1> {
    static constexpr unsigned value = 1u;
};
template <> struct Fibonacci<0> {
    static constexpr unsigned value = 0u;
};
static_assert(Fibonacci<6u>::value == 8u);
```

```
template <class T>
constexpr auto Fibonacci(T n) {
    static_assert(n >= 0);
    if (n == 1)
        return 1;
    if (n == 0)
        return 0;
    return Fibonacci(n - 2) + Fibonacci(n - 1);
}
static_assert(Fibonacci(6u) == 8u);
```

```cpp
constexpr auto Fibonacci(auto n) {
    static_assert(n >= 0);
    if (n == 1)
        return 1;
    if (n == 0)
        return 0;
    return Fibonacci(n - 2) + Fibonacci(n - 1);
}
static_assert(Fibonacci(6u) == 8u);
```

```
constexpr
auto Fibonacci(std::unsigned_integral auto n) {

    if (n == 1)
        return 1;
    if (n == 0)
        return 0;
    return Fibonacci(n - 2) + Fibonacci(n - 1);
}
static_assert(Fibonacci(6u) == 8u);
```

C++20 `constexpr`

- allocations (don't leave function)
- `constexpr std::vector` / `std::string`
- `consteval`

Fold expressions

Fold expressions

```
template <class... Args>
constexpr auto Sum(Args... args) {
    return (... + args);
}

static_assert(Sum(1, 2, 3, 4) == 10);
```

Fold expressions

```
template <class... Args>
constexpr auto Sum(Args... args) {
    return (... + args);
}

static_assert(Sum(1, 2, 3, 4) == 10);
```

Fold expressions

```
constexpr int Sum(int a, int b, int c, int d) {
    return (((a + b) + c) + d);
}

static_assert(Sum(1, 2, 3, 4) == 10);
```

Fold expressions

```
template <class... Args>
constexpr auto Sum(Args... args) {
    return (args + ...);
}

static_assert(Sum(1, 2, 3, 4) == 10);
```

Fold expressions

```
constexpr int Sum(int a, int b, int c, int d) {
    return (a + (b + (c + d)));
}

static_assert(Sum(1, 2, 3, 4) == 10);
```

Fold expressions

```
template <class T>
constexpr auto Sum(T first) {
    return first;
}
template <class T, class... Args>
constexpr auto Sum(T first, Args... rest) {
    return first + Sum(rest...);
}
static_assert(Sum(1,2,3,4) == 10);
```

C++20 Concepts

```
template <class T, class U>
U Foo(T x);
```

```
template <typename T, typename U>
U Foo(T x);
```

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>;
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

Using Concepts to create overload sets...

...and constrain types used in templates

```cpp
template <class T> constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
constexpr float Half(float x) {
    return x / 2.f;
}
constexpr double Half(double x) {
    return x / 2.0;
}
constexpr long double Half(long double x) {
    return x / 2.0;
}
```

```cpp
template <class T> constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
constexpr float Half(float x) {
    return x / 2.f;
}
constexpr double Half(double x) {
    return x / 2.0;
}
constexpr long double Half(long double x) {
    return x / 2.0;
}
```

```cpp
template <class T> constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
constexpr float Half(float x) {
    return x / 2.f;
}
constexpr double Half(double x) {
    return x / 2.0;
}
constexpr long double Half(long double x) {
    return x / 2.0;
}
```

```cpp
template <class T> constexpr auto Half(T x) {
    if constexpr (std::is_integral_v<T>) {
        return (x + T(1)) / T(2);
    } else {
        return x / T(2);
    }
}
```

```cpp
constexpr auto Half(std::integral auto x) {
    return (x + T(1)) / T(2);
}

constexpr auto Half(std::floating_point auto x) {
    return x / T(2);
}
```

```cpp
template <class T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T> requires std::integral<T>
constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
template <CanHalf T> requires std::floating_point<T>
constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <class T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T> requires std::integral<T>
constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
template <CanHalf T> requires std::floating_point<T>
constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <class T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T> requires std::integral<T>
constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
template <CanHalf T> requires std::floating_point<T>
constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <class T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T> requires std::integral<T>
constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
template <CanHalf T> requires std::floating_point<T>
constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <class T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T> requires std::integral<T>
constexpr auto Half(T x) {
    return (x + T(1)) / T(2);
}
template <CanHalf T> requires std::floating_point<T>
constexpr auto Half(T x) {
    return x / T(2);
}
```

Achtung!

Watch out for this easy mistake...

```
template <class T, class U>
concept ConcatPlus = requires(T a, U b) {
    a + b;
    requires std::same_as<std::decay_t<T>,
                          std::decay_t<U>>;
    requires !std::is_arithmetic_v<T>;
    requires !std::ranges::range<U>;
};
```

```
template <class T, class U>
concept ConcatPlus = requires(T a, U b) {
    a + b;
    requires std::same_as<std::decay_t<T>,
                          std::decay_t<U>>;
    requires !std::is_arithmetic_v<T>;
    requires !std::ranges::range<U>;
};
```

```
template <class T, class U>
concept ConcatPlus = requires(T a, U b) {
    a + b;
    std::same_as<std::decay_t<T>,
                 std::decay_t<U>>;
    !std::is_arithmetic_v<T>;
    !std::ranges::range<U>;
};
```

```
template <class T, class U>
concept ConcatPlus = requires(T a, U b) {
    a + b;
    requires std::same_as<std::decay_t<T>,
                          std::decay_t<U>>;
    requires !std::is_arithmetic_v<T>;
    requires !std::ranges::range<U>;
};
```

```
template <class T, class U>
concept ConcatPlus =
    std::same_as<std::decay_t<T>,
                 std::decay_t<U>> &&
    !std::is_arithmetic_v<T> &&
    !std::ranges::range<U> &&
    requires(T a, U b) {
        a + b;
    };
```

Concepts as API documentation

Concepts as API documentation

InPlaceProcessor

```
struct Inverse {
    void Process(std::span<float> buf) {
        std::transform(begin(buf), end(buf), begin(buf),
                       std::negate{});
    }
};
```

```
class OnePoleFilter {
public:
    void Process(std::span<float> buf) {
        std::transform(begin(buf), end(buf), begin(buf),
                       [this](float x) {
            m_prev = 0.01f * x + 0.99f * m_prev;
            return m_prev;
        });
    }
    void Reset() {
        m_prev = 0.f;
    }
private:
    float m_prev{0.f};
};
```

```cpp
//! ProcessorType must have a Process method taking a span<float>
template <class ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {
    }
    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        m_processor->Reset();
    } // …
```

```cpp
template <class ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {
        static_assert(IsInPlaceProcessor_v<ProcessorType>);
    }
    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        m_processor->Reset();
    } // …
```

```cpp
template <class T, class = void>
struct IsInPlaceProcessor : std::false_type {};

template <class T>
struct IsInPlaceProcessor<T, std::void_t<
                       decltype(std::declval<T>().Process(std::span<float>{}))>
                                        > : std::true_type {};

template <class T>
inline constexpr bool IsInPlaceProcessor_v = IsInPlaceProcessor<T>::value;
```

```
template <class T>
concept InPlaceProcessor = requires (T p) {
    p.Process(std::span<float>{});
};
```

```
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        m_processor->Reset();
    } // …
```

```
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        m_processor->Reset();
    } // …
```

```
template <class T>
concept InPlaceProcessor = requires (T p) {
    p.Process(std::span<float>{});
};
```

```
template <InPlaceProcessor ProcessorType>
class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor);

    void Process(std::span<float> buf);

    void Reset() {
        if constexpr (requires (ProcessorType p){ p.Reset(); }) {
            m_processor->Reset();
        }
    }
```

```
template <InPlaceProcessor ProcessorType>
class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor);

    void Process(std::span<float> buf);

    void Reset() requires requires (ProcessorType p){ p.Reset(); } {
        m_processor->Reset();
    }
```

Concepts as API documentation

BlockProcessor

```
class BlockProcessor {
public:

    void Process(std::span<const float> input,
                 std::span<float> output,
                 std::function<void(std::span<float>)>&&);
};
```

```
class BlockProcessor {
public:
    template <class ProcessFn>
    void Process(std::span<const float> input,
                 std::span<float> output,
                 ProcessFn&&);
};
```

```
class BlockProcessor {
public:
    template <std::invocable<std::span<float>> ProcessFn>
    void Process(std::span<const float> input,
                 std::span<float> output,
                 ProcessFn&&);
};
```

```
class BlockProcessor {
public:

    void Process(std::span<const float> input,
                 std::span<float> output,
                 std::invocable<std::span<float>> auto&&);
};
```

Metaprogramming

“All together now…”

```
template <typename Derived, typename Ps>
class HasProperties {
private:
  template <typename P>
  void createProperty() {
    auto success = m_propertyHolder.CreateProperty(
                     getName<P>(),
                     getName<typename P::property_type>(),
                     T::defaultValue);
    assert(success);
  }
};
```

Roth Michaels

https://www.youtube.com/watch?v=90I0hH5-r5A

A Case-study in Rewriting a Legacy GUI Library for Real-time Audio Software in Modern C++

```
struct Size {
    using type = float;
    static constexpr auto name = "Size";
    static constexpr type defaultValue = 100.f;
};
```

```cpp
template <class... Ts> class HasProperties {
public:
    HasProperties() {
        (addProperty<Ts>(), ...);
    };

private:
    template <class T>
    void addProperty() {
        m_defaults.emplace(
            std::pair<std::string, std::any>(T::name,
                                             T::defaultValue));
    }

    std::map<std::string, std::any> m_defaults;
};
```

```cpp
template <class... Ts> class HasProperties {
public:
    HasProperties() {
        (addProperty<Ts>(), ...);
    };

private:
    template <class T>
    void addProperty() {
        m_defaults.emplace(
            std::pair<std::string, std::any>(T::name,
                                             T::defaultValue));
    }

    std::map<std::string, std::any> m_defaults;
};
```

```
template <class... Ts> class HasProperties {
public:
    HasProperties() {
        (addProperty<Ts>(), ...);
    };

private:
    template <class T>
    void addProperty() {
        m_defaults.emplace(
            std::pair<std::string, std::any>(T::name,
                                             T::defaultValue));
    }

    std::map<std::string, std::any> m_defaults;
};
```

```
template <class... Ts> class HasProperties {
public:
    HasProperties() {
        (addProperty<Ts>(), ...);
    };

private:
    template <class T>
    void addProperty() {
        m_defaults.emplace(
            std::pair<std::string, std::any>(T::name,
                                             T::defaultValue));
    }

    std::map<std::string, std::any> m_defaults;
};
```

```
struct Size {
    using type = float;
    static constexpr auto name = "Size";
    static constexpr type defaultValue = 100.f;
};
```

```
struct Title {
    using type = std::string;
    static constexpr auto name = "Title";
    static type defaultValue() { return "My Button"; };
};
```

```
template <class T> struct OptionalProperty {
    using type = typename T::type;
    static auto name() {
        return std::format("Optional: {}", T::name);
    };
    static constexpr type defaultValue{};
};
```

```
template <class T> auto GetName() {
    if constexpr (requires { T::name(); }) {
        return T::name();
    } else {
        return T::name;
    }
};
```

```cpp
template <class T>
concept HasDefaultValueFunction = requires {
    T::defaultValue();
};

template <HasDefaultValueFunction T>
auto GetDefaultValue() {
    return T::defaultValue();
}

template <class T>
concept HasDefaultValue =
    std::same_as<decltype(T::defaultValue), typename T::type>;

template <HasDefaultValue T>
auto GetDefaultValue() {
    return T::defaultValue;
}
```

```cpp
template <class... Ts> class HasProperties {
public:
    HasProperties() {
        (addProperty<Ts>(), ...);
    };

private:
    template <class T>
    void addProperty() {
        m_defaults.emplace(
            std::pair<std::string, std::any>(
                                            GetName<T>(),
                                            GetDefaultValue<T>()));
    }

    std::map<std::string, std::any> m_defaults;
};
```

```
struct BorderWidth {
    using type = float;
    static constexpr auto name = "Border Width";
};
```

```
glass.cpp:108:44: error: no matching function for call to 'GetDefaultValue'
        std::cout << GetName<Size>() << " -- " << GetDefaultValue<Size>() << '\n';
                                                  ^~~~~~~~~~~~~~~~~~~~~
glass.cpp:54:43: note: candidate template ignored: constraints not satisfied [with T =
CppOnSea::Size]
template <HasDefaultValueFunction T> auto GetDefaultValue() {
                                          ^
glass.cpp:54:11: note: because 'CppOnSea::Size' does not satisfy 'HasDefaultValueFunction'
template <HasDefaultValueFunction T> auto GetDefaultValue() {
          ^
glass.cpp:51:2: note: because 'T::defaultValue()' would be invalid: called object type
'CppOnSea::Size::type' (aka 'float') is not a function or function
      pointer
        T::defaultValue();
        ^
glass.cpp:61:35: note: candidate template ignored: constraints not satisfied [with T =
CppOnSea::Size]
template <HasDefaultValue T> auto GetDefaultValue() {
                                  ^
glass.cpp:61:11: note: because 'CppOnSea::Size' does not satisfy 'HasDefaultValue'
template <HasDefaultValue T> auto GetDefaultValue() {
          ^
glass.cpp:59:27: note: because 'std::same_as<decltype(Size::defaultValue), typename Size::type>'
evaluated to false
concept HasDefaultValue = std::same_as<decltype(T::defaultValue), typename T::type>;
```

```
glass.cpp:108:44: error: no matching function for call to 'GetDefaultValue'
        std::cout << GetName<Size>() << " -- " << GetDefaultValue<Size>() << '\n';
                                                  ^~~~~~~~~~~~~~~~~~~~~
glass.cpp:54:43: note: candidate template ignored: constraints not satisfied [with T =
CppOnSea::Size]
template <HasDefaultValueFunction T> auto GetDefaultValue() {
                                          ^
glass.cpp:54:11: note: because 'CppOnSea::Size' does not satisfy 'HasDefaultValueFunction'
template <HasDefaultValueFunction T> auto GetDefaultValue() {
          ^
glass.cpp:51:2: note: because 'T::defaultValue()' would be invalid: called object type
'CppOnSea::Size::type' (aka 'float') is not a function or function
      pointer
        T::defaultValue();
        ^
glass.cpp:61:35: note: candidate template ignored: constraints not satisfied [with T =
CppOnSea::Size]
template <HasDefaultValue T> auto GetDefaultValue() {
                                  ^
glass.cpp:61:11: note: because 'CppOnSea::Size' does not satisfy 'HasDefaultValue'
template <HasDefaultValue T> auto GetDefaultValue() {
          ^
glass.cpp:59:27: note: because 'std::same_as<decltype(Size::defaultValue), typename Size::type>'
evaluated to false
concept HasDefaultValue = std::same_as<decltype(T::defaultValue), typename T::type>;
```

```
struct BorderWidth {
    using type = float;
    static constexpr auto name = "Border Width";
};


template <std::default_initializable T>
auto GetDefaultValue() {
    return typename T::type{};
}
```

```
struct Size {
    using type = float;
    static constexpr auto name = "Size";
    static constexpr type defaultValue = 100.f;
};

template <std::default_initializable T>
auto GetDefaultValue() {
    return typename T::type{};
}
```

```
struct BorderWidth {
    using type = float;
    static constexpr auto name = "Border Width";
};


template <std::default_initializable T>
requires(!HasDefaultValue<T> && !HasDefaultValueFunction<T>)
auto GetDefaultValue() {
    return typename T::type{};
}
```

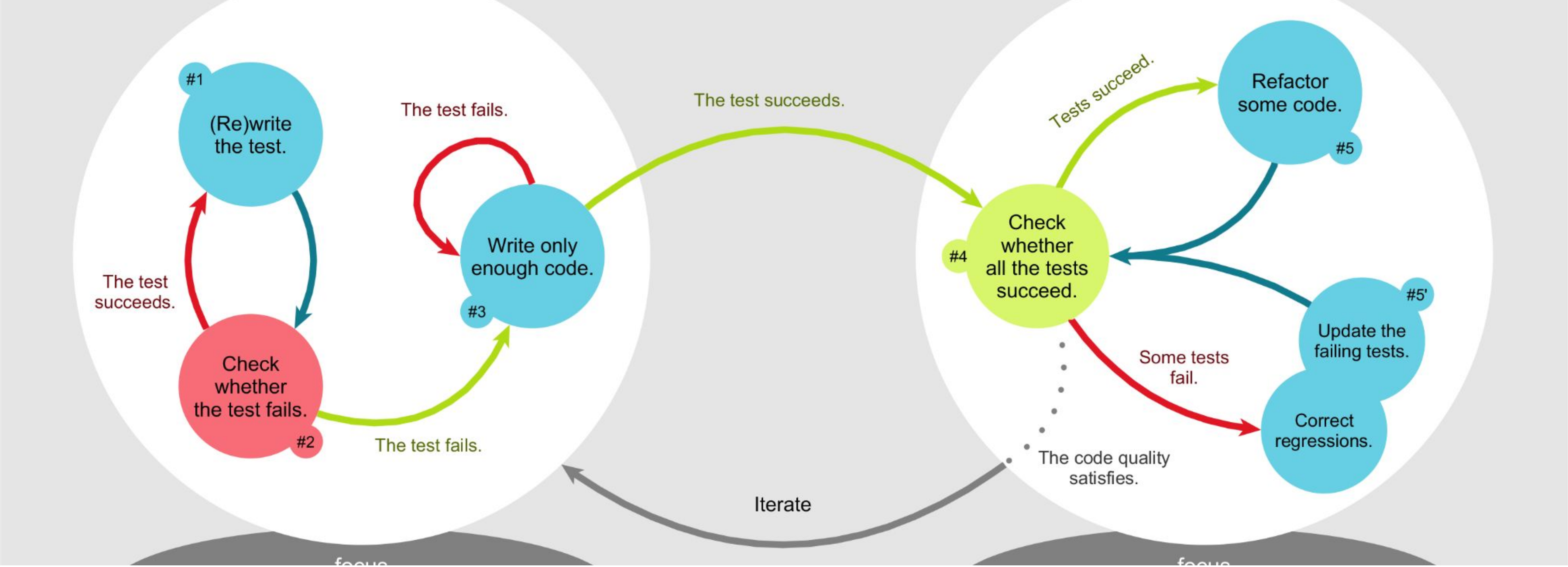
What does it look like with SFINAE?

```cpp
template <class T, class = void>
struct HasNameFunction : std::false_type {};

template <class T>
struct HasNameFunction<T, std::void_t<decltype(T::name())>> :
std::true_type {};

template <class T>
inline constexpr bool HasNameFunction_v =
HasNameFunction<T>::value;

template <class T> auto GetName() {
    if constexpr (HasNameFunction_v<T>) {
        return T::name();
    } else {
        return T::name;
    }
};
```

```
template <class T, class = void>
struct HasDefaultValueFunction : std::false_type {};

template <class T>
struct HasDefaultValueFunction<T, std::void_t<decltype(T::defaultValue())>> : std::true_type {};

template <class T>
inline constexpr bool HasDefaultValueFunction_v = HasDefaultValueFunction<T>::value;

template <class T, class = void>
struct HasDefaultValue : std::false_type {};

template <class T>
struct HasDefaultValue<T, std::void_t<decltype(T::defaultValue)>> : std::true_type {};

template <class T>
inline constexpr bool HasDefaultValue_v = HasDefaultValue<T>::value;

template <class T>
auto GetDefaultValue() {
      if constexpr (HasDefaultValue_v<T>) {
            if constexpr (HasDefaultValueFunction_v<T>) {
                  return T::defaultValue();
            } else {
                  return T::defaultValue;
            }
      } else if constexpr (std::is_default_constructible_v<T>) {
            return typename T::type{};
      }
}
```

When do we still need SFINAE?

Concepts and TDD



By Xarawn - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=44782343

```
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        m_processor->Reset();
    } // …
```

Concepts and TDD

1. Write a minimal concept (always true?)
2. static_assert that a test T matches that concept
3. Write a failing test
4. Start writing just enough code; if you will use T:
    a. Add use to the concept if needed
    b. See static_assert fail for the test T
    c. Make T conform to concept
5. Now use T, if test still fails repeat 4

```cpp
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {}


}
static_assert(InPlaceProcessor<T>);
```

```
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        // m_processor->Reset();
    }
}
static_assert(InPlaceProcessor<T>);
```

```
template <class T>
concept InPlaceProcessor = requires (T p) {
    p.Process(std::span<float>{});
    p.Reset();
};
```

```
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        // m_processor->Reset();
    }
}
static_assert(InPlaceProcessor<T>);
```

```cpp
template <InPlaceProcessor ProcessorType> class WrappedProcessor {
public:
    WrappedProcessor(std::shared_ptr<ProcessorType> processor)
    : m_processor{std::move(processor)} {}

    void Process(std::span<float> buf) {
        preProcess(buf);
        m_processor->Process(buf);
        postProcess(buf);
    }
    void Reset() {
        m_processor->Reset();
    }
}
static_assert(InPlaceProcessor<T>);
```

Conclusion

Our code looks like normal C++

Thank you!

Roth Michaels
Principal Software Engineer
roth.michaels@native-instruments.com
@thevibesman