



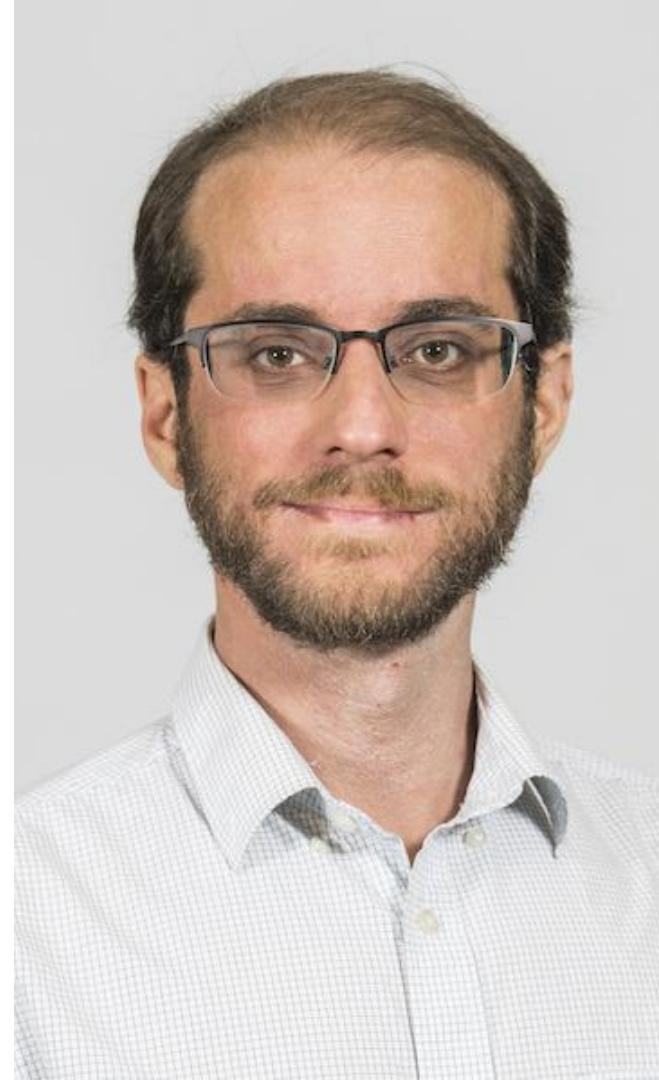
ROTH MICHAELS

TEMPLATES MADE EASY WITH C++20

HOW CONSTEXPR/CONSTEVAL,
FOLD EXPRESSIONS, AND
CONCEPTS CHANGE HOW WE
WRITE CODE

2024

Roth Michaels
Principal Software Engineer
Native Instruments





NATIVE INSTRUMENTS[®]



iZOTYPE



Plugin Alliance



BRAINWORX

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...

Abstract

- “Template metaprogramming has become an important part of a C++ programmer’s toolkit. This talk will demonstrate state-of-the-art metaprogramming tools and techniques, applying each to obtain representative implementations of selected standard library facilities.
- “Along the way, we will look at `void_t`, a recently-proposed, extremely simple new `<type_traits>` candidate whose use has been described by one expert as ‘highly advanced (and elegant), and surprising even to experienced template metaprogrammers.’ ”
- Presented in two parts, with a short break between.
- Note: not intended for C++ novices! (Sorry; another time?)

Copyright © 2014 by Walter E. Brown. All rights reserved.

2



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

Modern Template Metaprogramming
(Walter Brown)



NATIVE INSTRUMENTS[®]



iZOTYPE



Plugin Alliance



BRAINWORX

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);
```

Turing complete templates

Discovered by accident

```
template <unsigned N> struct Fibonacci {  
    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 <unsigned N> struct Fibonacci {  
    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 <unsigned N> struct Fibonacci {  
    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) {
    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(auto n) {  
    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**
- **constexpr**

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 auto 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 auto 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

C++20 Concepts

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

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))>
```

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))>
```

Using Concepts to create overload sets...

...and constrain types used in templates

```
template <class T> constexpr auto Half(T x) {
    return x / T(2) + (x % 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;
}
```

```
template <class T> constexpr auto Half(T x) {
    return x / T(2) + (x % 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;
}
```

```
template <class T> constexpr auto Half(T x) {
    return x / T(2) + (x % 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;
}
```

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

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

constexpr auto Half(std::floating_point auto 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(2) + (x % T(2));
}

template <CanHalf 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(2) + (x % T(2));
}

template <CanHalf 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(2) + (x % T(2));
}

template <CanHalf 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(2) + (x % T(2));
}

template <CanHalf T> constexpr auto Half(T x) {
    return x / T(2);
}
```

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::is_arithmetic_v<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::is_arithmetic_v<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::is_arithmetic_v<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::is_arithmetic_v<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::is_arithmetic_v<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};  
};
```

```
//! 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();
    } // ...
}
```

```
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();
    } // ...
}
```

```
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);
    }
};
```

<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++



Roth Michaels



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

```
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;
};
```

```
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;
    }
};
```

```
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;
}
```

```
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 'CppType' does not satisfy 'HasDefaultValueFunction'
template <HasDefaultValueFunction T> auto GetDefaultValue() {
                                         ^
glass.cpp:51:2: note: because 'T::defaultValue()' would be invalid: called object type
'CppType' (aka 'float') is not a function or function
pointer
    T::defaultValue();
    ^
glass.cpp:61:35: note: candidate template ignored: constraints not satisfied [with T =
CppType]
template <HasDefaultValue T> auto GetDefaultValue() {
                                         ^
glass.cpp:61:11: note: because 'CppType' 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 'CppType' does not satisfy 'HasDefaultValueFunction'
template <HasDefaultValueFunction T> auto GetDefaultValue() {
                                         ^
glass.cpp:51:2: note: because 'T::defaultValue()' would be invalid: called object type
'CppType' (aka 'float') is not a function or function
pointer
    T::defaultValue();
    ^
glass.cpp:61:35: note: candidate template ignored: constraints not satisfied [with T =
CppType]
template <HasDefaultValue T> auto GetDefaultValue() {
                                         ^
glass.cpp:61:11: note: because 'CppType' 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?

```
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 {};

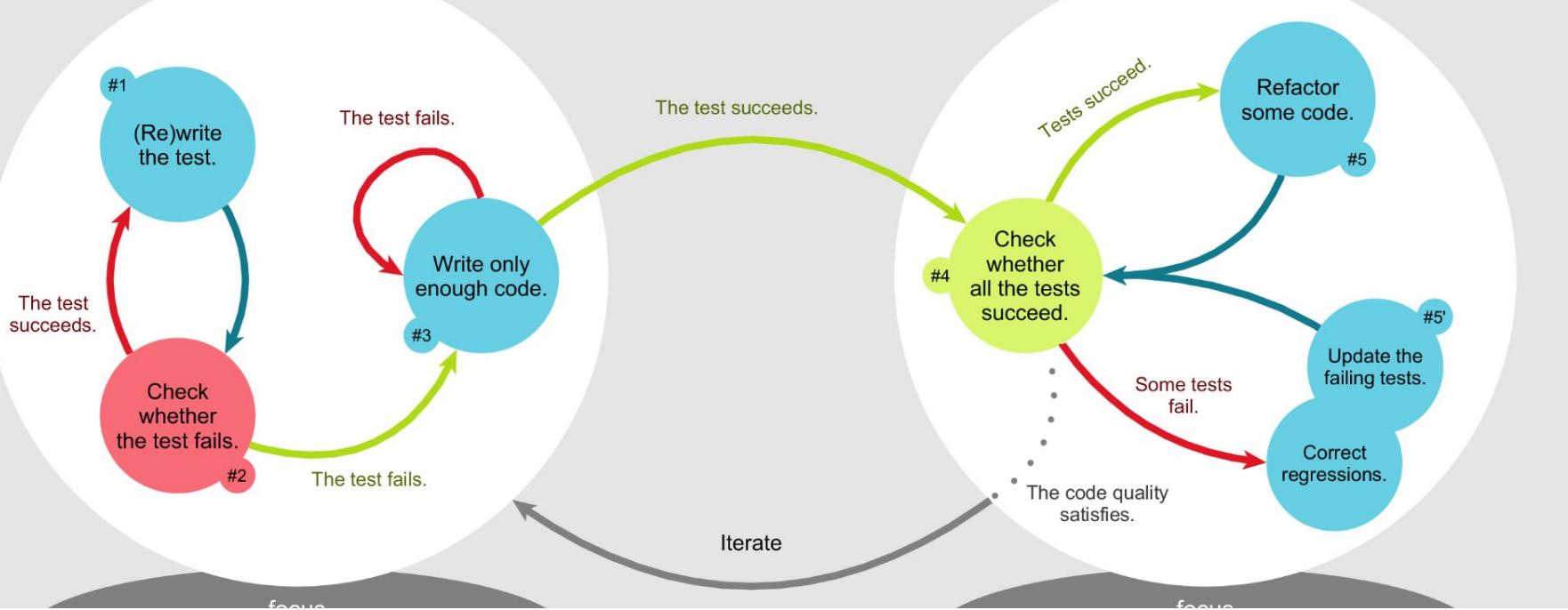
template <class T>
struct HasDefaultValue<T, std::void_t<decltype(T::defaultValue())>> : std::true_type {};

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{};
    }
}
```

Concepts and TDD

Live demo time! 😊



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

FIRE SEEKER



Master

JAZZ AMP



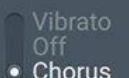
Bass



Mid



Treble



Vibrato
Off
• Chorus



Rate



Depth



Vo

Concepts and TDD

Live demo time! 😊

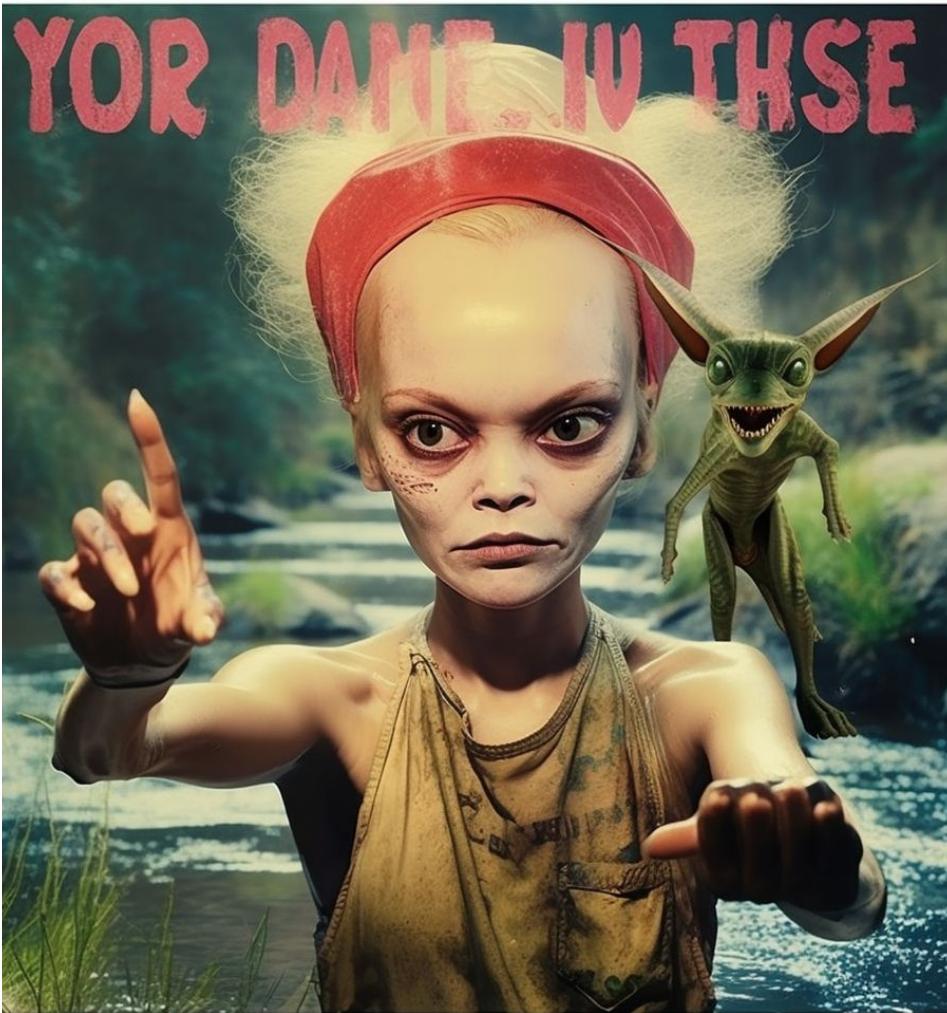


<https://www.youtube.com/watch?v=54gycqyHCPw>

Lightning Talk: What Does ChatGPT
Know About C++ and Why Fear the
Global Namespace?



You can do it!



#include <c++>

<https://www.includercpp.org>



Thank you!

Roth Michaels

Principal Software Engineer

roth.michaels@native-instruments.com

@thevibesman