

CppCon 2020

Template Metaprogramming:

Type Traits

Part 1

Jody Hagins
jhagins@maystreet.com
coachhagins@gmail.com

CppCon 2020

Template Metaprogramming: Type Traits

Introduction

Intended Audience

Intended Audience

- Beginner/Intermediate

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current
 - Not necessarily beginner to C++, but beginner to traditional template metaprogramming techniques

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current
 - Not necessarily beginner to C++, but beginner to traditional template metaprogramming techniques
 - Type traits part of standard library for ~10 years

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current
 - Not necessarily beginner to C++, but beginner to traditional template metaprogramming techniques
 - Type traits part of standard library for ~10 years
 - Fundamentals have been in use for ~20 years

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current
 - Not necessarily beginner to C++, but beginner to traditional template metaprogramming techniques
 - Type traits part of standard library for ~10 years
 - Fundamentals have been in use for ~20 years
 - Some have avoided more modern techniques because of a perceived large barrier to entry

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current
 - Not necessarily beginner to C++, but beginner to traditional template metaprogramming techniques
 - Type traits part of standard library for ~10 years
 - Fundamentals have been in use for ~20 years
 - Some have avoided more modern techniques because of a perceived large barrier to entry
- Part 2 is a continuation but with a bit more depth with swifter current

Intended Audience

- Beginner/Intermediate
 - Gentle entry: swimming pool to river
- Part 1 is beginner oriented: shallow depth, slow current
 - Not necessarily beginner to C++, but beginner to traditional template metaprogramming techniques
 - Type traits part of standard library for ~10 years
 - Fundamentals have been in use for ~20 years
 - Some have avoided more modern techniques because of a perceived large barrier to entry
- Part 2 is a continuation but with a bit more depth with swifter current
- Back to Basics: Templates - Andreas Fertig, Tuesday

Why This Talk?

Why This Talk?



Why This Talk?

cppcon +

Proposed new type trait `void_t`

- Near-trivial to specify:
 - `template< class ... >`
`using void_t = void;`
 - May need a workaround (sigh), pending resolution (at next meeting?) of CWG issue 1558 clarifying “treatment of unused arguments in an alias template specialization”:
 - `template< class ... > // helper to step around CWG 1558`
`struct voider { using type = void; };`
 - `template< class ... T0toN >`
`using void_t = typename voider< T0toN ... >::type;`
- In either case, how does another spelling of `void` help us?



Why This Talk?

- Tutorial

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry
- How to implement and how to use

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry
- How to implement and how to use
- Exploration of the standard set of type traits

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry
- How to implement and how to use
- Exploration of the standard set of type traits
 - Focus on techniques for implementing type traits

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry
- How to implement and how to use
- Exploration of the standard set of type traits
 - Focus on techniques for implementing type traits
- Remove some of the mystique that still surrounds template metaprogramming

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry
- How to implement and how to use
- Exploration of the standard set of type traits
 - Focus on techniques for implementing type traits
- Remove some of the mystique that still surrounds template metaprogramming
- Practical advice from a regular user

Why This Talk?

- Tutorial
- A relatively lengthy, gentle introduction - lower barrier to entry
- How to implement and how to use
- Exploration of the standard set of type traits
 - Focus on techniques for implementing type traits
- Remove some of the mystique that still surrounds template metaprogramming
- Practical advice from a regular user
- So you can more readily use the standard set and implement your own when needed

What is Metaprogramming?

What is Metaprogramming?

- In general, when programs treat programs as data

What is Metaprogramming?

- In general, when programs treat programs as data
- Could be other programs or itself

What is Metaprogramming?

- In general, when programs treat programs as data
- Could be other programs or itself
- Could be at "compile time" or at "run time"

What is Metaprogramming?

- In general, when programs treat programs as data
- Could be other programs or itself
- Could be at "compile time" or at "run time"
- We will discuss compile time metaprogramming in C++

What is Metaprogramming?

- In general, when programs treat programs as data
- Could be other programs or itself
- Could be at "compile time" or at "run time"
- We will discuss compile time metaprogramming in C++
- Wide array of current techniques, but still considered a niche

What is Metaprogramming?

- In general, when programs treat programs as data
- Could be other programs or itself
- Could be at "compile time" or at "run time"
- We will discuss compile time metaprogramming in C++
- Wide array of current techniques, but still considered a niche
- This two-part tutorial helps shed light on a very few essential idioms

Why Care About Metaprogramming (and type traits in particular)?

Why Care About Metaprogramming (and type traits in particular)?



“...one of the most highly regarded and expertly designed C++ library projects in the world.”
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

VERSION 1.13.0

VERSION 1.13.0

February 29th, 2000 12:00 GMT

Adds [Utility Library type_traits](#), [call_traits](#), and [compressed_pair](#) headers from John Maddock, Steve Cleary and Howard Hinnant.



Why Care About Metaprogramming (and type traits in particular)?



“...one of the most highly regarded and expertly designed C++ library projects in the world.”
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

VERSION 1.13.0

VERSION 1.13.0

February 29th, 2000 12:00 GMT

Adds Utility Library [type_traits](#), [call_traits](#), and [compressed_pair](#) headers from John Maddock, Steve Cleary and Howard Hinnant.



Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques

Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques
- Some requirements are impossible without advanced techniques (e.g., `std::optional`)

Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques
- Some requirements are impossible without advanced techniques (e.g., `std::optional`)
- Many third party libraries, not just Boost

Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques
- Some requirements are impossible without advanced techniques (e.g., `std::optional`)
- Many third party libraries, not just Boost
- Tools and idioms have become well developed, no longer black magic, limited to STL and Boost

Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques
- Some requirements are impossible without advanced techniques (e.g., `std::optional`)
- Many third party libraries, not just Boost
- Tools and idioms have become well developed, no longer black magic, limited to STL and Boost
- All C++ programmers should understand the basics

Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques
- Some requirements are impossible without advanced techniques (e.g., `std::optional`)
- Many third party libraries, not just Boost
- Tools and idioms have become well developed, no longer black magic, limited to STL and Boost
- All C++ programmers should understand the basics
- Any library developer should understand a good bit more

Why Care About Metaprogramming (and type traits in particular)?

- Each new standard library employs more metaprogramming techniques
- Some requirements are impossible without advanced techniques (e.g., `std::optional`)
- Many third party libraries, not just Boost
- Tools and idioms have become well developed, no longer black magic, limited to STL and Boost
- All C++ programmers should understand the basics
- Any library developer should understand a good bit more
- C++20 - concepts and independent requires expressions

Tutorial Format

Tutorial Format

- These two talks are intended as more of a tutorial

Tutorial Format

- These two talks are intended as more of a tutorial
- Several levels of discussion in C++ metaprogramming techniques

Tutorial Format

- These two talks are intended as more of a tutorial
- Several levels of discussion in C++ metaprogramming techniques
- Focused mainly on techniques related to standard type trait classes (old school and part of standard library)

Tutorial Format

- These two talks are intended as more of a tutorial
- Several levels of discussion in C++ metaprogramming techniques
- Focused mainly on techniques related to standard type trait classes (old school and part of standard library)
- Gentle entry to accommodate most programmers - shallow深深 of a pool - most of part 1 in the shallow end

Tutorial Format

- These two talks are intended as more of a tutorial
- Several levels of discussion in C++ metaprogramming techniques
- Focused mainly on techniques related to standard type trait classes (old school and part of standard library)
- Gentle entry to accommodate most programmers - shallow深深 of a pool - most of part 1 in the shallow end
- Listed as Beginner/Intermediate - some more advanced techniques in part 2.

Tutorial Format

Tutorial Format

- Will thoroughly explain the techniques used, but assume basic understanding

Tutorial Format

- Will thoroughly explain the techniques used, but assume basic understanding
 - Back to Basics: Templates Part 1
Andreas Fertig, Tuesday

Tutorial Format

- Will thoroughly explain the techniques used, but assume basic understanding
 - Back to Basics: Templates Part 1
Andreas Fertig, Tuesday
 - Lots of small code examples

Tutorial Format

- Will thoroughly explain the techniques used, but assume basic understanding
 - Back to Basics: Templates Part 1
Andreas Fertig, Tuesday
 - Lots of small code examples
 - A number of special considerations about types

Tutorial Format

- Will thoroughly explain the techniques used, but assume basic understanding
 - Back to Basics: Templates Part 1
Andreas Fertig, Tuesday
 - Lots of small code examples
 - A number of special considerations about types
 - To write programs where your data is types you need to consider a lot about the type system

Tutorial Format

- Will thoroughly explain the techniques used, but assume basic understanding
 - Back to Basics: Templates Part 1
Andreas Fertig, Tuesday
- Lots of small code examples
- A number of special considerations about types
 - To write programs where your data is types you need to consider a lot about the type system
- Will try to answer questions periodically - note slide number if relevant

CppCon 2020

Template Metaprogramming: Type Traits

Metafunctions

Metafunctions

Metafunctions

- Traditional functions have zero+ parameters and return a value (or void)

```
void do_something();  
int do_something_else(int, char const*);
```

Metafunctions

- Traditional functions have zero+ parameters and return a value (or void)

```
void do_something();  
int do_something_else(int, char const*);
```

- Mechanism for returning a value from a function is "return"

```
int do_something_else(int, char const*) {  
    return 42;  
}
```

Metafunctions

- Traditional functions have zero+ parameters and return a value (or void)

```
void do_something();  
int do_something_else(int, char const*);
```

- Mechanism for returning a value from a function is "return"

```
int do_something_else(int, char const*) {  
    return 42;  
}
```

- Compiler can enforce return values and syntax

Metafunctions

- Traditional functions have zero+ parameters and return a value (or void)

```
void do_something();  
int do_something_else(int, char const*);
```

- Mechanism for returning a value from a function is "return"

```
int do_something_else(int, char const*) {  
    return 42;  
}
```

- Compiler can enforce return values and syntax
- There is no alternative (ok... out parameters)

Metafunctions

Metafunctions

- A metafunction is not a function but a class/struct

Metafunctions

- A metafunction is not a function but a class/struct
- Metafunctions are not part of the language and have no formal language support

Metafunctions

- A metafunction is not a function but a class/struct
- Metafunctions are not part of the language and have no formal language support
- They exist as an idiomatic use of existing language features

Metafunctions

- A metafunction is not a function but a class/struct
- Metafunctions are not part of the language and have no formal language support
- They exist as an idiomatic use of existing language features
- Their use is not enforced by the language

Metafunctions

- A metafunction is not a function but a class/struct
- Metafunctions are not part of the language and have no formal language support
- They exist as an idiomatic use of existing language features
- Their use is not enforced by the language
- Their use is dictated by convention

Metafunctions

- A metafunction is not a function but a class/struct
- Metafunctions are not part of the language and have no formal language support
- They exist as an idiomatic use of existing language features
- Their use is not enforced by the language
- Their use is dictated by convention
- C++ community has created common "standard" conventions

Metafunctions

Metafunctions

- Technically, a class with zero+ template parameters and zero+ return types and values

Metafunctions

- Technically, a class with zero+ template parameters and zero+ return types and values
- Convention is that a metafunction should return one thing, like a regular function

Metafunctions

- Technically, a class with zero+ template parameters and zero+ return types and values
- Convention is that a metafunction should return one thing, like a regular function
- Convention was developed over time, so plenty of existing examples that do not follow this convention

Metafunctions

- Technically, a class with zero+ template parameters and zero+ return types and values
- Convention is that a metafunction should return one thing, like a regular function
- Convention was developed over time, so plenty of existing examples that do not follow this convention
- More modern metafunctions do follow this convention

Return From a Metafunction

Return From a Metafunction

- Expose a public value "value"

```
template <typename T>
struct TheAnswer {
    static constexpr int value = 42;
};
```

Return From a Metafunction

- Expose a public value "value"

```
template <typename T>
struct TheAnswer {
    static constexpr int value = 42;
};
```

- Expose a public type "type"

```
template <typename T>
struct Echo {
    using type = T;
};
```

Value Metafunctions

Value Metaprograms

- Simple regular function: identity

```
int int_identity(int x) {  
    return x;  
}  
assert(42 == int_identity(42));
```

Value Metaprograms

- Simple regular function: identity

```
int int_identity(int x) {  
    return x;  
}  
assert(42 == int_identity(42));
```

- Simple metaprogram: identity

```
template <int X>  
struct IntIdentity {  
    static constexpr int value = X;  
}  
static_assert(42 == IntIdentity<42>::value);
```

Value Metaprograms

- Simple regular function: identity

```
int int_identity(int x) {  
    return x;  
}  
assert(42 == int_identity(42));
```

- Simple metaprogram: identity

```
template <int X>  
struct IntIdentity {  
    static constexpr int value = X;  
}  
static_assert(42 == IntIdentity<42>::value);
```

Value Metaprograms

- Simple regular function: identity

```
int int_identity(int x) {  
    return x;  
}  
assert(42 == int_identity(42));
```

- Simple metaprogram: identity

```
template <int X>  
struct IntIdentity {  
    static constexpr int my_value = X;  
}  
static_assert(42 == IntIdentity<42>::value);
```

Value Metafunctions

Value Metaprocedures

- Generic Identity Function

Value Metafunctions

- Generic Identity Function

```
template <typename T>
T identity(T x) {
    return x;
}
```

```
// Returned type will be int
assert(42 == identity(42));
```

```
// Returned type will be unsigned long long
assert(42ull == identity(42ull));
```

Value Metafunctions

Value Metafunctions

- Generic Identity Metafunction

Value Metafuctions

- Generic Identity Metafunction

```
template <typename T, T Value>
struct ValueIdentity {
    static constexpr T value = Value;
}

// The type of value will be int
static_assert(42 == ValueIdentity<int, 42>::value);

// The type of value will be unsigned long long
static_assert(ValueIdentity<unsigned long long,
              42ull>::value == 42ull);
```

Value Metafunctions

Value Metafuctions

- Generic Identity Metafunction (C++17)

Value Metaprograms

- Generic Identity Metaprogram (C++17)

```
template <auto X>
struct ValueIdentity {
    static constexpr auto value = X;
}

// The type of value will be int
static_assert(42 == ValueIdentity<42>::value);

// The type of value will be unsigned long long
static_assert(42ull == ValueIdentity<42ull>::value);
```

Value Metaprograms: Sum

Value Metaprograms: Sum

```
int sum(int x, int y) {  
    return x + y;  
}  
  
template <int X, int Y>  
struct IntSum {  
    static constexpr int value = X + Y;  
};  
static_assert(42 == IntSum<30, 12>::value);
```

Value Metaprograms: Sum

Value Metaprograms: Sum

Value Metaprograms: Sum

```
template <typename X, typename Y>
auto sum(X x, Y y) {
    return x + y;
}
```

```
// Return type will be unsigned long long
assert(42ull == sum(30, 12ull));
```

```
template <auto X, auto Y>
struct Sum {
    static constexpr auto value = X + Y;
};
```

```
// Return type will be unsigned long long
static_assert(42ull == Sum<30, 12ull>::value);
```

Value Metaprograms: Sum

```
template <typename X, typename Y>
constexpr auto sum(X x, Y y) {
    return x + y;
}
```

```
// Return type will be unsigned long long
static_assert(42ull == sum(30, 12ull));
```

```
template <auto X, auto Y>
struct Sum {
    static constexpr auto value = X + Y;
};
```

```
// Return type will be unsigned long long
static_assert(42ull == Sum<30, 12ull>::value);
```

Type Metafunctions

Type Metafunctions

- Workhorse (especially with the advent of `constexpr`)

Type Metafunctions

- Workhorse (especially with the advent of `constexpr`)
- "Returns" a type

Type Metafunctions

- Workhorse (especially with the advent of `constexpr`)
- "Returns" a type

```
template <typename T>
struct TypeIdentity {
    using type = T;
};
```

Type Metafunctions

- Workhorse (especially with the advent of `constexpr`)
- "Returns" a type

```
template <typename T>
struct TypeIdentity {
    using type = T;
};
```

- C++20 Introduces `std::type_identity`

Calling Type Metafunctions

Calling Type Metafunctions

- `ValueIdentity<42>::value`

Calling Type Metafunctions

- `ValueIdentity<42>::value`
- `TypeIdentity<int>::type`

Calling Type Metafunctions

- `ValueIdentity<42>::value`
- `TypeIdentity<int>::type`
- Typename Dance

```
typename TypeIdentity<T>::type
```

Convenience Calling Conventions

Convenience Calling Conventions

- Value metafunctions use variable templates ending with "_v"

Convenience Calling Conventions

- Value metafunctions use variable templates ending with "_v"

```
template <auto X>
inline constexpr auto ValueIdentity_v =
    ValueIdentity<X>::value;

static_assert(42 == ValueIdentity<42>::value);

static_assert(42 == ValueIdentity_v<42>);
```

Convenience Calling Conventions

Convenience Calling Conventions

- Type metafunctions use alias templates ending with "_t"

Convenience Calling Conventions

- Type metafunctions use alias templates ending with "_t"

```
template <typename T>
using TypeIdentity_t = typename TypeIdentity<T>::type;

static_assert(std::is_same_v<int,
    TypeIdentity_t<int>>);
```

Convenience Calling Conventions

Convenience Calling Conventions

- Easier to use

Convenience Calling Conventions

- Easier to use
- Each one must be explicitly hand written

Convenience Calling Conventions

- Easier to use
- Each one must be explicitly hand written
- A meta-convention to get around that which I may get to if time for bonus material

std::integral_constant

std::integral_constant

- A very useful metafunction

std::integral_constant

- A very useful metafunction

```
template <class T, T v>
struct integral_constant {
    static constexpr T value = v;

    using value_type = T;
    using type       = integral_constant<T, v>;

    constexpr operator value_type() const noexcept {
        return value;
    }
    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

std::integral_constant

- A very useful metafunction

```
template <class T, T v>
struct integral_constant {
    static constexpr T value = v;

    using value_type = T;
    using type       = integral_constant<T, v>;

    constexpr operator value_type() const noexcept {
        return value;
    }
    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

std::integral_constant

- A very useful metafunction

```
template <class T, T v>
struct integral_constant {
    static constexpr T value = v;

    using value_type = T;
    using type        = integral_constant<T, v>;

    constexpr operator value_type() const noexcept {
        return value;
    }
    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

std::bool_constant

std::bool_constant

- Convenient helpers

std::bool_constant

- Convenient helpers

```
template <bool B>
using bool_constant = integral_constant<bool, B>;  
  
using true_type = bool_constant<true>;  
  
using false_type = bool_constant<false>;
```

std::bool_constant

- Convenient helpers

```
template <bool B>
using bool_constant = integral_constant<bool, B>;  
  
using true_type = bool_constant<true>;  
  
using false_type = bool_constant<false>;
```

- `true_type` and `false_type` are called nullary metafunctions because they have no parameters

```
true_type::value
false_type::value
```

CppCon 2020

Template Metaprogramming:
Type Traits

Standard Type Trait Requirements

Cpp17UnaryTypeTrait

Cpp17UnaryTypeTrait

- Class template

Cpp17UnaryTypeTrait

- Class template
- One template type argument*

Cpp17UnaryTypeTrait

- Class template
- One template type argument*
- Cpp17DefaultConstructible

Cpp17UnaryTypeTrait

- Class template
- One template type argument*
- Cpp17DefaultConstructible
- Cpp17CopyConstructible

Cpp17UnaryTypeTrait

- Class template
- One template type argument*
- Cpp17DefaultConstructible
- Cpp17CopyConstructible
- Publicly and unambiguously derived from a specialization of `std::integral_constant`

Cpp17UnaryTypeTrait

- Class template
- One template type argument*
- Cpp17DefaultConstructible
- Cpp17CopyConstructible
- Publicly and unambiguously derived from a specialization of `std::integral_constant`
- The member names of the base characteristic shall not be hidden and shall be unambiguously available

Cpp17BinaryTypeTrait

Cpp17BinaryTypeTrait

- Class template
- **Two** template type argument*
- Cpp17DefaultConstructible
- Cpp17CopyConstructible
- Publicly and unambiguously derived from a specialization of std::integral_constant
- The member names of the base characteristic shall not be hidden and shall be unambiguously available

Cpp17TransformationTrait

Cpp17TransformationTrait

- Class template

Cpp17TransformationTrait

- Class template
- One template type argument*

Cpp17TransformationTrait

- Class template
- One template type argument*
- Define a publicly accessible nested type named **type**

Cpp17TransformationTrait

- Class template
- One template type argument*
- Define a publicly accessible nested type named **type**
- No default/copy constructible requirement

Cpp17TransformationTrait

- Class template
- One template type argument*
- Define a publicly accessible nested type named **type**
- No default/copy constructible requirement
- No inheritance requirement

Undefined Behavior

Undefined Behavior

- Do not specialize standard type traits

Undefined Behavior

- Do not specialize standard type traits

⁴ Unless otherwise specified, the behavior of a program that adds specializations for any of the templates specified in this subclause [20.15](#) is undefined.

Undefined Behavior

Undefined Behavior

- Be very careful when using incomplete types

Undefined Behavior

- Be very careful when using incomplete types

- 5 Unless otherwise specified, an incomplete type may be used to instantiate a template specified in this subclause. The behavior of a program is undefined if:
- (5.1) — an instantiation of a template specified in subclause 20.15 directly or indirectly depends on an incompletely-defined object type T, and
 - (5.2) — that instantiation could yield a different result were T hypothetically completed.

CppCon 2020

Template Metaprogramming:
Type Traits

Specialization

is_void (Unary Type Trait)

is_void (Unary Type Trait)

- Value metafunction: is the type void?
yields true_type or false_type

is_void (Unary Type Trait)

- Value metafunction: is the type void?
yields true_type or false_type
- Specialization

is_void (Unary Type Trait)

- Value metafunction: is the type void?
yields `true_type` or `false_type`
- Specialization
 - Primary template: general case

```
template <typename T>
struct is_void : std::false_type { };
```

is_void (Unary Type Trait)

- Value metafunction: is the type void?
yields `true_type` or `false_type`
- Specialization
 - Primary template: general case

```
template <typename T>
struct is_void : std::false_type { };
```

- Specialization: special case(s)

```
template <>
struct is_void<void> : std::true_type { };
```

is_void (Unary Type Trait)

- Value metafunction: is the type void?
yields `true_type` or `false_type`
- Specialization
 - Primary template: general case

```
template <typename T>
struct is_void : std::false_type { };
```

- Specialization: special case(s)

```
template <>
struct is_void<void> : std::true_type { };
```

is_void (Unary Type Trait)

- Value metafunction: is the type void?
yields `true_type` or `false_type`
- Specialization
 - Primary template: general case

```
template <typename T>
struct is_void : std::false_type { };
```

- Specialization: special case(s)

```
template <>
struct is_void<void> : std::true_type { };
```

```
static_assert(is_void<void>{});
static_assert(not is_void<int>{});
```

is_void: cv qualifiers

is_void: cv qualifiers

- Is void const void?

is_void: cv qualifiers

- Is void const void?
- Is void volatile void?

is_void: cv qualifiers

- Is void const void?
- Is void volatile void?
- is_void is in Primary Type Categories

is_void: cv qualifiers

- Is void const void?
- Is void volatile void?
- is_void is in Primary Type Categories

"For any given type T, the result of applying one of these templates to T and to cv T shall yield the same result."

is_void: cv qualifiers

- Is void const void?
- Is void volatile void?
- is_void is in Primary Type Categories

"For any given type T, the result of applying one of these templates to T and to cv T shall yield the same result."

- Yes - no matter your opinion ;-)

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type { };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_void: redux

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type { };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

remove_const (Transformation Trait)

`remove_const` (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int  
remove_const<const int> -> int
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
```

```
remove_const<const int> -> int
```

```
remove_const<const volatile int> -> volatile int
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
remove_const<const int> -> int
remove_const<const volatile int> -> volatile int
remove_const<int *> -> int *
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
remove_const<const int> -> int
remove_const<const volatile int> -> volatile int
remove_const<int *> -> int *
remove_const<const int *> -> int *
```

remove_const (Transformation Trait)

- Formal Definition

*"The member `typedef` type names the same type as T except that any **top-level** const-qualifier has been removed."*

```
remove_const<int> -> int
remove_const<const int> -> int
remove_const<const volatile int> -> volatile int
remove_const<int *> -> int *
remove_const<const int *> -> int * 💩
```



remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
remove_const<const int> -> int
remove_const<const volatile int> -> volatile int
remove_const<int *> -> int *
remove_const<const int *> -> const int *
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
remove_const<int const> -> int
remove_const<int const volatile> -> int volatile
remove_const<int *> -> int *
remove_const<int const *> -> int const *
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
remove_const<int> -> int
remove_const<int const> -> int
remove_const<int const volatile> -> int volatile
remove_const<int *> -> int *
remove_const<int const *> -> int const *
```

```
remove_const<int const * const> -> int const *
remove_const<int * const> -> int *
```

`remove_const` (Transformation Trait)

`remove_const` (Transformation Trait)

- **Formal Definition**

"The member `typedef` type names the same type as T except that any top-level `const`-qualifier has been removed."

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };

// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };

// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };
```

```
// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

- Formal

"The members of type T except those marked const are removed."

```
template <typename T>
struct TypeIdentity {
    using type = T;
};
```

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };
```

```
// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };
```

```
// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };
```

```
// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };
```

```
// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `const`-qualifier has been removed."

```
// Primary template, do nothing if no const
template <typename T>
struct remove_const : TypeIdentity<T> { };
```

```
// Partial specialization, when detect const
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_const_t = typename remove_const <T>::type;
```

remove_const (Transformation Trait)

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

- Contains "const" so the partial specialization will match

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

- Contains "const" so the partial specialization will match

```
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

- Contains "const" so the partial specialization will match

```
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

The full match is **remove_const<int volatile const>**

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

- Contains "const" so the partial specialization will match

```
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

The full match is **remove_const<int volatile const>**

The const is explicitly matched so the part remaining to match with the "T" is **int volatile**

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

- Contains "const" so the partial specialization will match

```
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

The full match is **remove_const<int volatile const>**

The const is explicitly matched so the part remaining to match with the "T" is **int volatile**

So, we inherit from **TypeIdentity<int volatile>**

remove_const (Transformation Trait)

- Example

```
remove_const<int volatile const>
```

- Contains "const" so the partial specialization will match

```
template <typename T>
struct remove_const<T const> : TypeIdentity<T> { };
```

The full match is **remove_const<int volatile const>**

The const is explicitly matched so the part remaining to match with the "T" is **int volatile**

So, we inherit from **TypeIdentity<int volatile>**

Returned type will be **int volatile**

conditional

conditional

```
template <bool Condition, typename T, typename F>
struct conditional : TypeIdentity<T> { };

template <typename T, typename F>
struct conditional<false, T, F> : TypeIdentity<F> { };

static_assert(is_same_v<int,
                     conditional_t<is_void<void>::value, int, long>);
static_assert(is_same_v<long,
                     conditional_t<is_void<char>::value, int, long>);
```

conditional

```
template <bool Condition, typename T, typename F>
struct conditional : TypeIdentity<T> { };

template <typename T, typename F>
struct conditional<false, T, F> : TypeIdentity<F> { };

static_assert(is_same_v<int,
                     conditional_t<is_void<void>::value, int, long>);
static_assert(is_same_v<long,
                     conditional_t<is_void<char>::value, int, long>);
```

CppCon 2020

Template Metaprogramming: Type Traits Part 2

Jody Hagins
jhagins@maystreet.com
coachhagins@gmail.com

CppCon 2020

Template Metaprogramming: Type Traits

Primary Type Categories

Primary type categories

Primary type categories

is_void

is_null_pointer

is_integral

is_floating_point

is_array

is_enum

is_union

is_class

is_function

is_pointer

is_lvalue_reference

is_rvalue_reference

is_member_object_pointer

is_member_function_pointer

Primary type categories

is_void

is_null_pointer

is_integral

is_floating_point

is_array

is_enum

is_union

is_class

is_function

is_pointer

is_lvalue_reference

is_rvalue_reference

is_member_object_pointer

is_member_function_pointer

All are to have base characteristic of either true_type or false_type

Primary type categories

is_void

is_null_pointer

is_integral

is_floating_point

is_array

is_enum

is_union

is_class

is_function

is_pointer

is_lvalue_reference

is_rvalue_reference

is_member_object_pointer

is_member_function_pointer

All are to have base characteristic of either true_type or false_type

All should yield the same result in light of cv qualifiers

Primary type categories

is_void

is_null_pointer

is_integral

is_floating_point

is_array

is_enum

is_union

is_class

is_function

is_pointer

is_lvalue_reference

is_rvalue_reference

is_member_object_pointer

is_member_function_pointer

All are to have base characteristic of either true_type or false_type

All should yield the same result in light of cv qualifiers

For any given type T, exactly one of the primary type categories shall yield true_type

is_void

is_void

```
template <typename T> struct is_void : std::false_type { };

template <> struct is_void<void> : std::true_type { };

template <> struct is_void<void const> : std::true_type { };

template <> struct is_void<void volatile> : std::true_type { };

template <> struct is_void<void const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_void_v = is_void<T>::value;
```

is_null_pointer

is_null_pointer

```
template <typename T> struct is_null_pointer : std::false_type { };

template <> struct is_null_pointer<std::nullptr_t>
: std::true_type { };

template <> struct is_null_pointer<std::nullptr_t const>
: std::true_type { };

template <> struct is_null_pointer<std::nullptr_t volatile>
: std::true_type { };

template <> struct is_null_pointer<std::nullptr_t const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_null_pointer_v = is_null_pointer<T>::value;
```

is_null_pointer

```
template <typename T> struct is_null_pointer : std::false_type { };

template <> struct is_null_pointer<std::nullptr_t>
: std::true_type { };

template <> struct is_null_pointer<std::nullptr_t const>
: std::true_type { };

template <> struct is_null_pointer<std::nullptr_t volatile>
: std::true_type { };

template <> struct is_null_pointer<std::nullptr_t const volatile>
: std::true_type {     };

// The standard mandates this as well...
template <typename T>
inline constexpr bool is_null_pointer_v = is_null_pointer<T>::value;
```

is_floating_point

is_floating_point

- float

is_floating_point

- float
- double

is_floating_point

- float
- double
- long double

is_floating_point

- float
- double
- long double
- Requires 12 specializations

is_floating_point

```
template <typename T> struct is_floating_point : std::false_type { };
template <> struct is_floating_point<float> : std::true_type { };
template <> struct is_floating_point<float const> : std::true_type { };
template <> struct is_floating_point<float volatile> : std::true_type { };
template <> struct is_floating_point<float const volatile> : std::true_type { };
template <> struct is_floating_point<double> : std::true_type { };
template <> struct is_floating_point<double const> : std::true_type { };
template <> struct is_floating_point<double volatile> : std::true_type { };
template <> struct is_floating_point<double const volatile> : std::true_type { };
template <> struct is_floating_point<long double> : std::true_type { };
template <> struct is_floating_point<long double const> : std::true_type { };
template <> struct is_floating_point<long double volatile> : std::true_type { };
template <> struct is_floating_point<long double const volatile> : std::true_type { };
template <typename T> inline constexpr bool is_floating_point_v =
    is_floating_point<T>::value;
```

is_integral

is_integral

- Five standard signed integer types: signed char, short int, int, long int, long long int

is_integral

- Five standard signed integer types: signed char, short int, int, long int, long long int
- Implementation defined extended signed integer types

is_integral

- Five standard signed integer types: signed char, short int, int, long int, long long int
- Implementation defined extended signed integer types
- Corresponding, but different, unsigned integer types

is_integral

- Five standard signed integer types: signed char, short int, int, long int, long long int
- Implementation defined extended signed integer types
- Corresponding, but different, unsigned integer types
- `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`

is_integral

- Five standard signed integer types: signed char, short int, int, long int, long long int
- Implementation defined extended signed integer types
- Corresponding, but different, unsigned integer types
- `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`
- `bool`

is_integral

- Five standard signed integer types: signed char, short int, int, long int, long long int
- Implementation defined extended signed integer types
- Corresponding, but different, unsigned integer types
- `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`
- `bool`
- Requires $16^4 = 64$ specializations

Metafunction Abstractions

Metafunction Abstractions

- We would have reached for this long before now with regular/normal programming

Metafunction Abstractions

- We would have reached for this long before now with regular/normal programming
- Treat metafunction programming like regular programming because, well, that's what it is

Metafunction Abstractions

- We would have reached for this long before now with regular/normal programming
- Treat metafunction programming like regular programming because, well, that's what it is
- Step back to the land of regular functions

Metafunction Abstractions

- We would have reached for this long before now with regular/normal programming
- Treat metafunction programming like regular programming because, well, that's what it is
- Step back to the land of regular functions
- Pretend we needed to implement these same ideas with strings instead of types

is_void

```
bool is_void(std::string_view s)
{
    return s == "void"
        || s == "void const"
        || s == "void volatile"
        || s == "void const volatile"
        || s == "void volatile const";
}
```

is_null_pointer

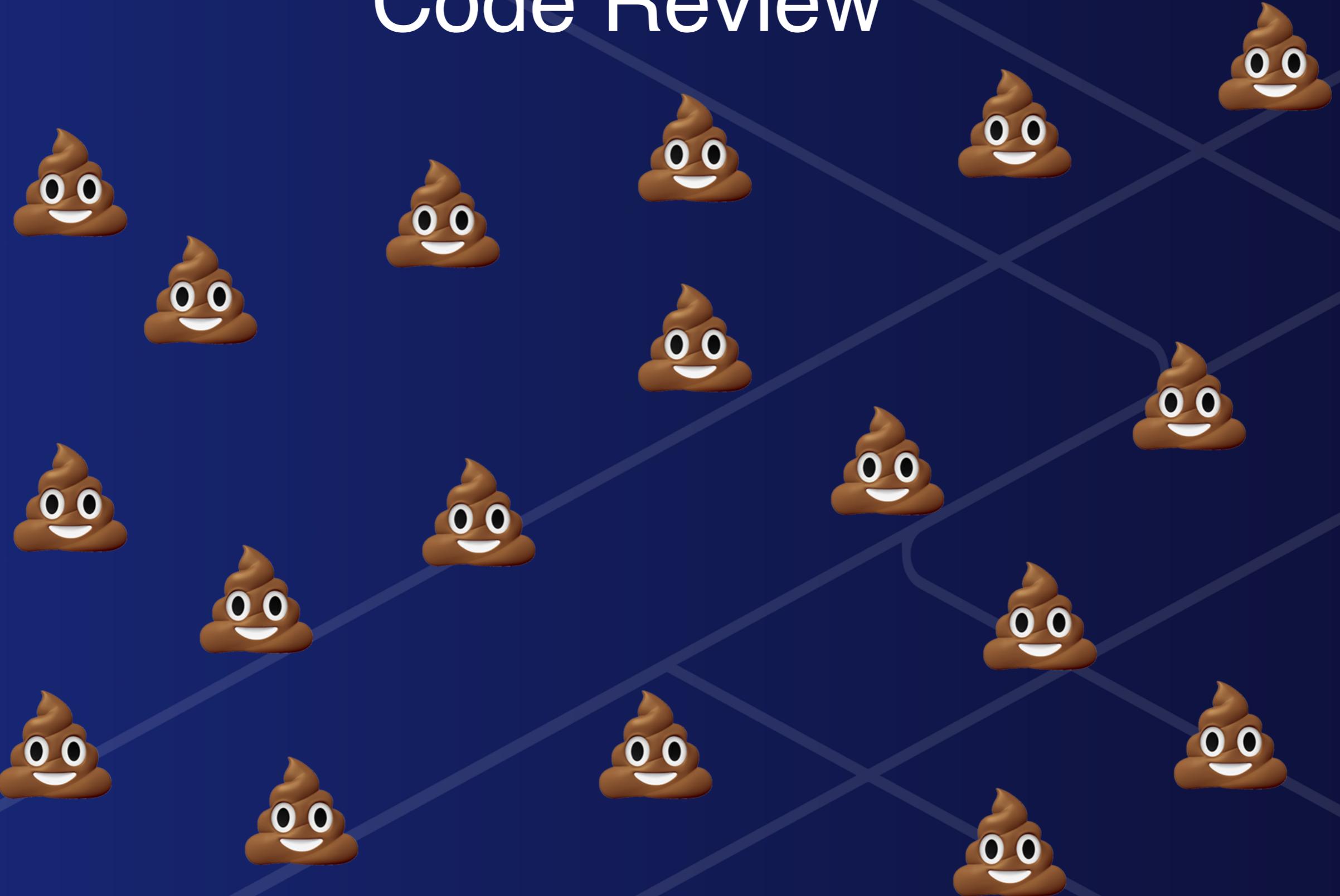
```
bool is_null_pointer(std::string_view s)
{
    return s == "std::nullptr_t"
        || s == "std::nullptr_t const"
        || s == "std::nullptr_t volatile"
        || s == "std::nullptr_t const volatile"
        || s == "std::nullptr_t volatile const";
}
```

is_floating_point

```
bool is_floating_point(std::string_view s)
{
    return s == "float"
        || s == "float const"
        || s == "float volatile"
        || s == "float const volatile"
        || s == "float volatile const"
        || s == "double"
        || s == "double const"
        || s == "double volatile"
        || s == "double const volatile"
        || s == "double volatile const"
        || s == "long double"
        || s == "long double const"
        || s == "long double volatile"
        || s == "long double const volatile"
        || s == "long double volatile const";
}
```

Code Review

Code Review



Code Review

A Step In The Right Direction

A Step In The Right Direction

```
// Remove all cv-qualifiers from the string, leaving the raw type
std::string_view remove_cv(std::string_view);

bool is_void(std::string_view s)
{
    return remove_cv(s) == "void";
}

bool is_null_pointer(std::string_view s)
{
    return remove_cv(s) == "std::nullptr_t";
}

bool is_floating_point(std::string_view input)
{
    auto const s = remove_cv(input);
    return s == "float"
        || s == "double"
        || s == "long double"
}
```

A Step In The Right Direction

```
std::string_view strip_signed(std::string_view);  
  
bool is_integral(std::string_view input)  
{  
    auto const s = strip_signed(remove_cv(input));  
    return s == "bool"  
        || s == "char8_t"  
        || s == "char16_t"  
        || s == "char32_t"  
        || s == "wchar_t"  
        || s == "char"  
        || s == "short"  
        || s == "int"  
        || s == "long"  
        || s == "long long";  
}
```

remove_cv

remove_cv

- Already have remove_const

remove_cv

- Already have remove_const
- Need remove_volatile

remove_cv

- Already have remove_const
- Need remove_volatile
- Compose them to get remove_cv

remove_volatile

remove_volatile

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level `volatile`-qualifier has been removed."

remove_volatile

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `volatile`-qualifier has been removed."

```
// Primary template, do nothing if no volatile
template <typename T>
struct remove_volatile : TypeIdentity<T> { };

// Partial specialization, when detect volatile
template <typename T>
struct remove_volatile<T volatile> : TypeIdentity<T> { };

// Standard mandated convenience alias
template <typename T>
using remove_volatile_t = typename remove_volatile<T>::type;
```

remove_volatile

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `volatile`-qualifier has been removed."

```
// Primary template, do nothing if no volatile
template <typename T>
struct remove_volatile : TypeIdentity<T> { };
```

```
// Partial specialization, when detect volatile
template <typename T>
struct remove_volatile<T volatile> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
template <typename T>
using remove_volatile_t = typename remove_volatile<T>::type;
```

remove_volatile

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level `volatile`-qualifier has been removed."

```
// Primary template, do nothing if no volatile
template <typename T>
struct remove_volatile : TypeIdentity<T> { };
```

```
// Partial specialization, when detect volatile
```

```
template <typename T>
struct remove_volatile<T volatile> : TypeIdentity<T> { };
```

```
// Standard mandated convenience alias
```

```
template <typename T>
using remove_volatile_t = typename remove_volatile<T>::type;
```

remove_cv

remove_cv

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level cv-qualifier has been removed."

remove_cv

- Formal Definition

"The member `typedef` type names the same type as T except that any top-level cv-qualifier has been removed."

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;

template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

remove_cv

- Formal Definition

"The member type `remove_cv` names the same type as `T` if `T` is not `const` or `volatile`, and names the type `remove_cv<T>` otherwise." [\[C++11\]](#)

```
template <typename T>
using remove_cv = remove_const<remove_volatile<T>::type>;
```

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```



```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

remove_cv

- Formal Definition

"The member typedef type names the same type as
T using remove_cv =
remove_const<remove_volatile<T>::type>;

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```



```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

remove_cv

- Formal Definition

"The member `typedef` type names the same type as `T` except that any top-level cv-qualifier has been removed."

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;

template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
remove_const<remove_volatile_t<int const volatile>>
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
remove_const<remove_volatile_t<int const volatile>>
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
remove_const<remove_volatile_t<int const volatile>>
remove_const<typename remove_volatile<int const volatile>::type>
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
remove_const<remove_volatile_t<int const volatile>>
remove_const<typename remove_volatile<int const volatile>::type>
remove_const<int const>
```

remove_cv<int const volatile>

```
// Removing volatile, then const
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;
```

```
remove_cv<int const volatile>
remove_const<remove_volatile_t<int const volatile>>
remove_const<typename remove_volatile<int const volatile>::type>
remove_const<int const>
```

remove_cv_t<int const volatile>

remove_cv_t<int const volatile>

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

remove_cv_t<int const volatile>

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

```
remove_cv_t<int const volatile>
```

remove_cv_t<int const volatile>

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

```
remove_cv_t<int const volatile>
typename remove_cv<int const volatile>::type
```

remove_cv_t<int const volatile>

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

```
remove_cv_t<int const volatile>
typename remove_cv<int const volatile>::type
typename remove_const<int const>::type
```

remove_cv_t<int const volatile>

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

```
remove_cv_t<int const volatile>
typename remove_cv<int const volatile>::type
typename remove_const<int const>::type
int
```

remove_cv_t<int const volatile>

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;
```

```
remove_cv_t<int const volatile>
typename remove_cv<int const volatile>::type
typename remove_const<int const>::type
int
```

Comparing Types

Comparing Types

- Earlier, I said that specialization is like a comparison function for types

Comparing Types

- Earlier, I said that specialization is like a comparison function for types
- However, it can be a bit verbose and can only be used in a context that allows explicit specializations of class templates

Comparing Types

- Earlier, I said that specialization is like a comparison function for types
- However, it can be a bit verbose and can only be used in a context that allows explicit specializations of class templates
- Let's write a true comparison metafunction for types

is_same

is_same

```
// Primary template - two types are never the same
template <typename T1, typename T2>
struct is_same : std::false_type { };
```

is_same

```
// Primary template - two types are never the same
template <typename T1, typename T2>
struct is_same : std::false_type { };
```

is_same

```
// Primary template - two types are never the same
template <typename T1, typename T2>
struct is_same : std::false_type { };
```

is_same

```
// Primary template - two types are never the same
template <typename T1, typename T2>
struct is_same : std::false_type { };

// Partial specialization - when they are both the same
template <typename T>
struct is_same<T, T> : std::true_type { };

template <typename T1, typename T2>
constexpr bool is_same_v = is_same<T1, T2>::value;
```

is_same

```
// Primary template - two types are never the same
template <typename T1, typename T2>
struct is_same : std::false_type { };

// Partial specialization - when they are both the same
template <typename T>
struct is_same<T, T> : std::true_type { };

template <typename T1, typename T2>
constexpr bool is_same_v = is_same<T1, T2>::value;

static_assert(not is_same_v<int, unsigned>);
// T1 = int, T2 = unsigned - primary template matches
// No way to make T to match specialization
```

is_same

```
// Primary template - two types are never the same
template <typename T1, typename T2>
struct is_same : std::false_type { };

// Partial specialization - when they are both the same
template <typename T>
struct is_same<T, T> : std::true_type { };

template <typename T1, typename T2>
constexpr bool is_same_v = is_same<T1, T2>::value;
```

```
static_assert(not is_same_v<int, unsigned>);
// T1 = int, T2 = unsigned - primary template matches
// No way to make T to match specialization
```

```
static_assert(is_same_v<int, int>);
// T1 = int, T2 = int - primary template matches
// T = int - specialization matches
```

is_same_raw

is_same_raw

```
template <typename T1, typename T2>
using is_same_raw = is_same<remove_cv_t<T1>, remove_cv_t<T2>>;
```



```
template <typename T1, typename T2>
constexpr bool is_same_raw_v = is_same_raw<T1, T2>::value;
```

is_floating_point: redux

is_floating_point: redux

```
template <typename T>
using is_floating_point = std::bool_constant<
    is_same_raw_v<float,           T>
  || is_same_raw_v<double,        T>
  || is_same_raw_v<long double,   T>;
```

is_floating_point: redux

```
template <typename T>
using is_floating_point = std::bool_constant<
    is_same_raw_v<float, T> ||
    is_same_raw_v<double, T> ||
    is_same_raw_v<long double, T>;
```

```
template <bool B>
using bool_constant = integral_constant<bool, B>;
```

is_floating_point: redux

```
template <typename T>
using is_floating_point = std::bool_constant<
    is_same_raw_v<float,           T>
  || is_same_raw_v<double,        T>
  || is_same_raw_v<long double,   T>;
```

is_integral: redux

is_integral: redux

```
template <typename T>
using is_integral = std::bool_constant<
    is_same_raw_v<bool, T> ||
    is_same_raw_v<char, T> ||
    is_same_raw_v<char8_t, T> ||
    is_same_raw_v<char16_t, T> ||
    is_same_raw_v<char32_t, T> ||
    is_same_raw_v<wchar_t, T> ||
    is_same_raw_v<signed char, T> ||
    is_same_raw_v<short, T> ||
    is_same_raw_v<int, T> ||
    is_same_raw_v<long, T> ||
    is_same_raw_v<long long, T> ||
    is_same_raw_v<unsigned char, T> ||
    is_same_raw_v<unsigned short, T> ||
    is_same_raw_v<unsigned int, T> ||
    is_same_raw_v<unsigned long, T> ||
    is_same_raw_v<unsigned long long, T>>;
```

is_integral: redux

is_integral: redux

```
template <typename TargetT, typename ... Ts>
using is_type_in_pack = ... ;  
  
template <typename T>
using is_integral = is_type_in_pack<remove_cv_t<T>,
    bool
    , char, char8_t, char16_t, char32_t, wchar_t
    , signed char, unsigned char
    , signed short, unsigned short
    , signed int, unsigned int
    , signed long, unsigned long
    , signed long long, unsigned long long
>;
```

is_array

is_array

```
template <typename T>
struct is_array : std::false_type { };

template <typename T, std::size_t N>
struct is_array<T[N]> : std::true_type { };

template <typename T>
struct is_array<T[]> : std::true_type { };
```

is_array

```
template <typename T>
struct is_array : std::false_type { };

template <typename T, std::size_t N>
struct is_array<T[N]> : std::true_type { };

template <typename T>
struct is_array<T[]> : std::true_type { };
```

is_array

```
template <typename T>
struct is_array : std::false_type { };

template <typename T, std::size_t N>
struct is_array<T[N]> : std::true_type { };

template <typename T>
struct is_array<T[]> : std::true_type { };
```

```
static_assert(is_array<int[5]>);
// T = int[5] - primary template matches
// T = int, N = 5 - first specialization matches
// No way to form T to match second specialization
```

is_array

```
template <typename T>
struct is_array : std::false_type { };

template <typename T, std::size_t N>
struct is_array<T[N]> : std::true_type { };

template <typename T>
struct is_array<T[]> : std::true_type { };
```

```
static_assert(is_array<int[5]>);
// T = int[5] - primary template matches
// T = int, N = 5 - first specialization matches
// No way to form T to match second specialization
```

```
static_assert(is_array<int[]>);
// T = int[] - primary template matches
// No way to form T and N to match first specialization
// T = int - second specialization matches
```

is_pointer

is_pointer

```
// Primary template - most things are not pointers
template <typename T>
struct is_pointer : std::false_type { };

// When we have a pointer
template <typename T>
struct is_pointer<T *> : std::true_type { };
```

is_pointer

```
namespace detail {  
template <typename T>  
struct is_pointer_impl : std::false_type { };  
  
template <typename T>  
struct is_pointer_impl<T *> : std::true_type { };  
}  
  
template <typename T>  
using is_pointer = detail::is_pointer_impl<remove_cv_t<T>>;
```

is_union

is_union

```
// This metafunction is actually impossible to implement without
// support from the compiler. Both clang and gcc provide this
// particular compiler intrinsic to determine if a type is a union.
template <typename T>
using is_union = std::bool_constant<__is_union(T)>;
```

is_class

is_class

- Almost always implemented as compiler intrinsic

is_class

- Almost always implemented as compiler intrinsic
- Without compiler assistance, impossible to distinguish between union and nonunion class type

is_class

- Almost always implemented as compiler intrinsic
- Without compiler assistance, impossible to distinguish between union and nonunion class type
- We have `is_union` (with help from the compiler)

is_class

- Almost always implemented as compiler intrinsic
- Without compiler assistance, impossible to distinguish between union and nonunion class type
- We have `is_union` (with help from the compiler)
- Can we tell if something in the "union or class" category?

is_class

" T is a non-union class type."

- Almost always implemented as compiler intrinsic
- Without compiler assistance, impossible to distinguish between union and nonunion class type
- We have is_union (with help from the compiler)
- Can we tell if something in the "union or class" category?
- Each type must be in exactly one of the 14 categories

is_class_or_union

is_class_or_union

- What do we know about unions and classes that is unique to those two types?

is_class_or_union

- What do we know about unions and classes that is unique to those two types?
- They can have members

is_class_or_union

- What do we know about unions and classes that is unique to those two types?
- They can have members
- Devise a way to detect if a type can have a member

is_class_or_union

- What do we know about unions and classes that is unique to those two types?
- They can have members
- Devise a way to detect if a type can have a member
- How can you tell if a class has a member?

is_class_or_union

- What do we know about unions and classes that is unique to those two types?
- They can have members
- Devise a way to detect if a type can have a member
- How can you tell if a class has a member?
- The syntax for a pointer-to-member is valid for any class, even without any members

is_class_or_union

is_class_or_union

- `int*` is a valid pointer type - does not have to point to anything

is_class_or_union

- `int*` is a valid pointer type - does not have to point to anything
- `int Foo::*` is a member pointer type - does not have to point to anything

is_class_or_union

- `int*` is a valid pointer type - does not have to point to anything
- `int Foo::*` is a member pointer type - does not have to point to anything

```
// An empty struct, with no members of any kind
struct Bar { };

// BarIntObjectMemPtr is an alias for a type that is a pointer to
// a member of class Bar, where the member is an int.
using BarIntObjectMemPtr = int Bar::*;

// This, however, generates a hard compiler error
using LongIntObjectMemPtr = int long::*;


```

Function Overload Resolution

Function Overload Resolution

```
namespace detail {
std::true_type is_nullptr(std::nullptr_t);
std::false_type is_nullptr(...);
}

template <typename T>
using is_null_pointer = decltype(detail::is_nullptr(std::declval<T>()));
```

Function Overload Resolution

```
namespace detail {
std::true_type is_nullptr(std::nullptr_t);
std::false_type is_nullptr(...);
}

template <typename T>
using is_null_pointer = decltype(detail::is_nullptr(std::declval<T>()));
```

Function Overload Resolution

```
namespace detail {
std::true_type is_nullptr(std::nullptr_t);
std::false_type is_nullptr(...);
}

template <typename T>
using is_null_pointer = decltype(detail::is_nullptr(std::declval<T>()));
```

Function Overload Resolution

```
namespace detail {
    std::true_type is_nullptr(std::nullptr_t);
    std::false_type is_nullptr(...);
}

template <typename T>
using is_null_pointer = decltype(detail::is_nullptr(std::declval<T>()));
```



```
static_assert(not is_null_pointer<int>::value);
static_assert(is_null_pointer<std::nullptr_t>::value);
```

Function Overload Resolution

```
namespace detail {  
    std::true_type is_nullptr(std::nullptr_t);  
    std::false_type is_nullptr(...);  
}
```

```
template <typename T>  
using is_null_pointer = decltype(detail::is_nullptr(std::declval<T>()));
```

```
static_assert(not is_null_pointer<int>::value);  
static_assert(is_null_pointer<std::nullptr_t>::value);
```

Function Overload Resolution

```
namespace detail {
    std::true_type is_nullptr(std::nullptr_t);
    std::false_type is_nullptr(...);
}

template <typename T>
using is_null_pointer = decltype(detail::is_nullptr(std::declval<T>()));
```



```
static_assert(not is_null_pointer<int>::value);
static_assert(is_null_pointer<std::nullptr_t>::value);
```

Function Overload Resolution

Function Overload Resolution

```
namespace detail {
template <typename T>
std::true_type isconst(TypeIdentity<T const>);
template <typename T>
std::false_type isconst(TypeIdentity<T>);
}
template <typename T>
using is_const =
decltype(detail::isconst(std::declval<TypeIdentity<T>>()));
```

Function Overload Resolution

```
namespace detail {
template <typename T>
std::true_type isconst(TypeIdentity<T const>);
template <typename T>
std::false_type isconst(TypeIdentity<T>);
}
template <typename T>
using is_const =
decltype(detail::isconst(std::declval<TypeIdentity<T>>()));
```

Function Overload Resolution

```
namespace detail {
template <typename T>
std::true_type isconst(TypeIdentity<T const>);
template <typename T>
std::false_type isconst(TypeIdentity<T>);
}
template <typename T>
using is_const =
decltype(detail::isconst(std::declval<TypeIdentity<T>>()));
```

```
static_assert(not is_const<std::nullptr_t>::value);
static_assert(is_const<int const>::value);
```

Function Overload Resolution

```
namespace detail {
template <typename T>
std::true_type isconst(TypeIdentity<T const>);

template <typename T>
std::false_type isconst(TypeIdentity<T>);

using is_const =
 decltype(detail::isconst(std::declval<TypeIdentity<T>>()));
```

```
static_assert(not is_const<std::nullptr_t>::value);
static_assert(is_const<int const>::value);
```

Function Overload Resolution

```
namespace detail {
template <typename T>
std::true_type isconst(TypeIdentity<T const>);
template <typename T>
std::false_type isconst(TypeIdentity<T>);
}
template <typename T>
using is_const =
decltype(detail::isconst(std::declval<TypeIdentity<T>>()));
```

```
static_assert(not is_const<std::nullptr_t>::value);
static_assert(is_const<int const>::value);
```

SFINAE

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

```
struct Foo { };
static_assert(std::declval<can_have_pointer_to_member<Foo>>().operator()());
```

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

```
struct Foo { };
static_assert(std::decltype(can_have_pointer_to_member<Foo>(nullptr)));
```

```
std::true_type can_have_pointer_to_member<Foo>(int Foo::* );
std::false_type can_have_pointer_to_member<Foo>(...);
```

SFINAE

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

```
static_assert(std::is_same_v<decltype(can_have_pointer_to_member<int>(nullptr)), std::true_type>);
```

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::*);  
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

```
static_assert(std::is_same_v<decltype(can_have_pointer_to_member<int>(nullptr)), std::true_type>);
```

```
std::true_type can_have_pointer_to_member<int>(int int::*);  
std::false_type can_have_pointer_to_member<int>(...);
```

SFINAE

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);

template <typename T>
using can_have_member_ptr =
    decltype(can_have_pointer_to_member<T>(nullptr));
```

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

```
template <typename T>
using can_have_member_ptr =
    decltype(can_have_pointer_to_member<T>(nullptr));
```

```
static_assert(decltype(can_have_member_ptr<int>(nullptr)));
```

SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

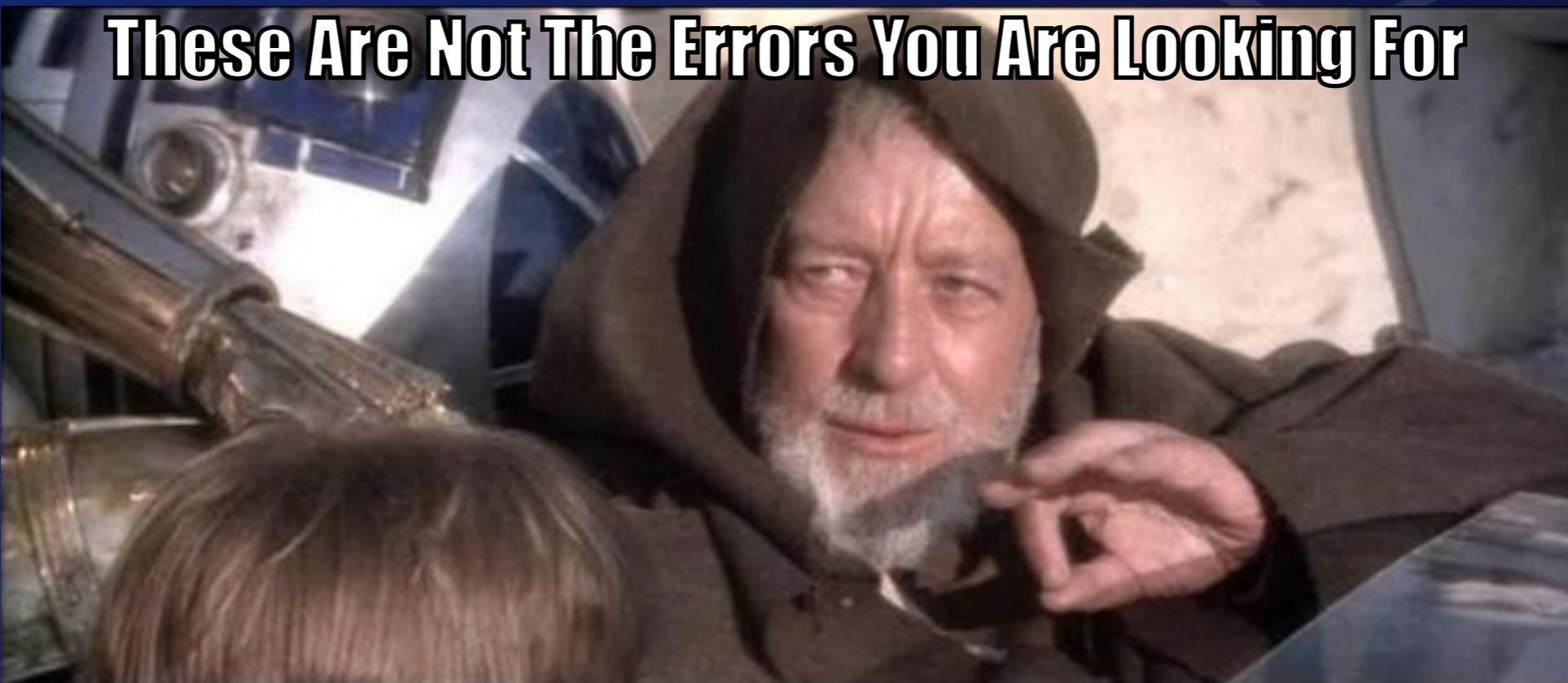
```
template <typename T>
using can_have_member_ptr =
    decltype(can_have_pointer_to_member<T>(nullptr));
```

```
static_assert(decltype(can_have_member_ptr<int>(nullptr)));
```

```
std::true_type can_have_pointer_to_member<int>(int int::* );
std::false_type can_have_pointer_to_member<int>(...);
```

SFINAE

These Are Not The Errors You Are Looking For



SFINAE

```
template <typename T>
std::true_type can_have_pointer_to_member(int T::* );
template <typename T>
std::false_type can_have_pointer_to_member(...);
```

```
template <typename T>
using can_have_member_ptr =
    decltype(can_have_pointer_to_member<T>(nullptr));
```

```
static_assert(decltype(can_have_member_ptr<int>(nullptr)));
```

```
std::true_type can_have_pointer_to_member<int>(int int::* );
std::false_type can_have_pointer_to_member<int>(...);
```

is_class

is_class

```
namespace detail {  
template <typename T>  
std::bool_constant<not std::is_union_v<T>>  
is_class_or_union(int T::*);  
  
template <typename T>  
std::false_type is_class_or_union(...);  
}  
  
template <typename T>  
using is_class = decltype(detail::is_class_or_union<T>(nullptr));
```

is_class

is_class

```
namespace detail {
template <typename T> constexpr bool is_class_or_union(int T::* ) {
    return not std::is_union<T>::value;
}
template <typename T> constexpr bool is_class_or_union(...) {
    return false;
}
}

template <typename T>
using is_class =
    std::bool_constant<detail::is_class_or_union<T>(nullptr)>;
```

CppCon 2020

Template Metaprogramming: Type Traits

Bonus: `is_in_pack`

Traditional Recursion

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };

static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };
```

```
static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Traditional Recursion

```
// Template declaration, with no definition
template <typename TargetT, typename ... Ts> struct IsInPack;

// Base case - no more elements
template<typename TargetT>
struct IsInPack<TargetT> : std::false_type { };

// If the first one matches the target, we are done
template<typename TargetT, typename ... Ts>
struct IsInPack<TargetT, TargetT, Ts...> : std::true_type { };

// Otherwise, check the remaining ones
template<typename TargetT, typename T, typename ... Ts>
struct IsInPack<TargetT, T, Ts...> : IsInPack<TargetT, Ts...> { };

static_assert(IsInPack<int, double, char, int, float>::value);
static_assert(not IsInPack<long, double, char, int, float>::value);
```

Using `is_base_of`

```
namespace detail {  
template <typename ... Ts>  
struct IsInPackImpl : TypeIdentity<Ts>... { };  
}
```

```
template <typename TargetT, typename ... Ts>  
using IsInPack = std::is_base_of<  
    TypeIdentity<TargetT>,  
    detail::IsInPackImpl<Ts...>>;
```

```
static_assert(IsInPack<int, double, char, int, float>::value);  
static_assert(not IsInPack<long, double, char, int, float>::value);
```

CppCon 2020

Template Metaprogramming: Type Traits

Bonus: `void_t`

is_class (void_t)

is_class (void_t)

```
template <typename...> using void_t = void;
```

is_class (void_t)

```
template <typename...> using void_t = void;
```

```
namespace detail {
    template <typename T, typename = void>
    struct is_class_or_union : std::false_type { };

    template <typename T>
    struct is_class_or_union<T, std::void_t<int T::*>>
        : std::bool_constant<not std::is_union_v<T>> { };
}

template <typename T>
using is_class = typename detail::is_class_or_union<T>::type;
```

void_t; continued

void_t; continued

<https://stackoverflow.com/questions/44845945/why-void-t-doesnt-work-in-sfinae-but-enable-if-does>

void_t; continued

```
template <typename...> using void_t = void;
```

```
struct One { using x = int; };
struct Two { using y = int; };

template <typename T, std::void_t<typename T::x>* = nullptr>
void func() { }

template <typename T, std::void_t<typename T::y>* = nullptr>
void func() { }

int main() {
    func<One>();
    func<Two>();
}
```

void_t; continued

```
template <typename...> using void_t = void;
```

```
struct One { using x = int; };
struct Two { using y = int; };

template <typename T, std::void_t<typename T::x>* = nullptr>
void func() { }

template <typename T, std::void_t<typename T::y>* = nullptr>
void func() { }

int main() {
    func<One>();
    func<Two>();
}
```

void_t; continued

```
template <typename ... Ts> using Blarg = void;

struct One { using x = int; };
struct Two { using y = int; };

template <typename T, Blarg<typename T::x>* = nullptr>
void func() { }

template <typename T, Blarg<typename T::y>* = nullptr>
void func() { }

int main() {
    func<One>();
    func<Two>();
}
```

void_t; continued

```
struct Blart { };  
template <typename ... Ts> using Blarg = Blart;  
  
struct One { using x = int; };  
struct Two { using y = int; };  
  
template <typename T, Blarg<typename T::x>* = nullptr>  
void func() { }  
  
template <typename T, Blarg<typename T::y>* = nullptr>  
void func() { }  
  
int main() {  
    func<One>();  
    func<Two>();  
}
```

void_t; continued

```
struct Blargt { };  
template <typename ... Ts> struct Blarg : Blargt { };  
  
struct One { using x = int; };  
struct Two { using y = int; };  
  
template <typename T, Blarg<typename T::x>* = nullptr>  
void func() { }  
  
template <typename T, Blarg<typename T::y>* = nullptr>  
void func() { }  
  
int main() {  
    func<One>();  
    func<Two>();  
}
```

void_t; continued

```
template <typename T> struct TypeIdentity { using type = T; };
struct Blart { };
template <typename ... Ts>
using Blarg = typename TypeIdentity<Blart>::type;

struct One { using x = int; };
struct Two { using y = int; };

template <typename T, Blarg<typename T::x>* = nullptr>
void func() { }

template <typename T, Blarg<typename T::y>* = nullptr>
void func() { }

int main() {
    func<One>();
    func<Two>();
}
```

void_t; continued

```
template <typename ...> struct void_t_impl { using type = void; };
template <typename ... Ts>
using VoidT = typename void_t_impl<Ts...>::type;
```

```
struct One { using x = int; };
struct Two { using y = int; };
```

```
template <typename T, VoidT<typename T::x>*> = nullptr>
void func() { }
```

```
template <typename T, VoidT<typename T::y>*> = nullptr>
void func() { }
```

```
int main() {
    func<One>();
    func<Two>();
}
```

void_t; continued

```
template <typename T, typename U> struct FooBlarg;

template <typename T, typename U, typename = void> struct Blip;

template <typename T, typename U>
struct Blip<T, U, std::void_t<decltype(FooBlarg<T, U>::value)>>
{
};

template <typename T, typename U>
struct Blip<T, U, std::void_t<typename T::zz, typename T::yy>>
{
};
```

void_t; continued

```
template <typename T, typename U> struct FooBlarg;

template <typename T, typename U, typename = void> struct Blip;

template <typename T, typename U>
struct Blip<T, U, std::void_t<decltype(FooBlarg<T, U>::value)>>
{
};

template <typename T, typename U>
struct Blip<T, U, std::void_t<typename T::zz, typename T::yy>>
{
};
```

void_t; continued

```
<source>:14:8: error: redefinition of 'Blip<T, U,  
std::void_t<typename T::zz, typename T::yy> '>  
struct Blip<T, U, std::void_t<typename T::zz, typename T::yy>>  
^~~~~~  
  
<source>:10:8: note: previous definition is here  
struct Blip<T, U, std::void_t<decltype(FooBlarg<T, U>::value)>>
```

void_t; continued

```
template <typename T, typename U> struct FooBlarg;

template <typename T, typename U, typename = void> struct Blip;

template <typename T, typename U>
struct Blip<T, U, std::void_t<decltype(FooBlarg<T, U>::value)>>
{
};

template <typename T, typename U>
struct Blip<T, U, std::void_t<typename T::zz, typename T::yy>>
{
};
```

void_t; continued

```
template <typename T, typename U> struct FooBlarg;

template <typename T, typename U, typename = void> struct Blip;

template <typename T, typename U>
struct Blip<T, U, VoidT<decltype(FooBlarg<T, U>::value)>>
{
};

template <typename T, typename U>
struct Blip<T, U, VoidT<typename T::zz, typename T::yy>>
{
};
```

void_t; continued

```
namespace detail {
template <typename T,
          std::void_t<decltype(++std::declval<T &>())> * = nullptr>
T & blarg_impl(T &t)
{
    return ++t;
}
}

inline constexpr auto blarg = [](auto &x)
-> decltype(detail::blarg_impl(x))
{
    return detail::blarg_impl(x);
};

// We can call pre-increment on an int, so this should be true
static_assert(std::is_invocable_v<decltype(blarg), int &>);

struct Bar { };
// We cannot call pre-increment on a Bar, so this should be false
static_assert(not std::is_invocable_v<decltype(blarg), Bar &>);
```

void_t; continued

```
template <typename T,  
         std::void_t<decltype(++std::declval<T &>())> * = nullptr>  
T & blarg_impl(T &t)  
noexcept(noexcept(++std::declval<T &>()))  
{  
    return ++t;  
}
```

CppCon 2020

Template Metaprogramming: Type Traits

Bonus: Misc

Meta Calling Convention

Meta Calling Convention

```
template <template <typename...> class MF, typename... Ts>
using CallMF = typename MF<Ts...>::type;
```

```
static_assert(std::is_same_v<
    CallMF<remove_cv, int const volatile>,
    std::remove_cv_t<int const volatile>>);
```

```
template <template <typename...> class MF, typename... Ts>
constexpr bool CallVMF = MF<Ts...>::value;
```

```
static_assert(CallVMF<std::is_same,
    CallMF<remove_cv, int const volatile>,
    std::remove_cv_t<int const volatile>>);
```

is_(lval/rval)_reference

is_(lval/rval)_reference

```
// Primary template, most things are not lvalue references
template <typename T>
struct is_lvalue_reference : std::false_type { };

// Specialization for when something is a lvalue reference
template <typename T>
struct is_lvalue_reference<T &> : std::true_type { };

// Primary template, most things are not rvalue references
template <typename T>
struct is_rvalue_reference : std::false_type { };

// Specialization for when something is an rvalue reference
template <typename T>
struct is_rvalue_reference<T &&> : std::true_type { };
```

is_function

is_function

```
// primary template - for things that are not functions
template <typename T>
struct is_function : std::false_type { };

// specialization for functions
template <typename Ret, typename ... Args>
struct is_function<Ret (Args...)> : std::true_type {};
```

is_function

```
// primary template - for things that are not functions
template <typename T>
struct is_function : std::false_type { };

// specialization for functions
template <typename Ret, typename ... Args>
struct is_function<Ret (Args...)> : std::true_type {};
```

is_function

```
// primary template - for things that are not functions
template <typename T>
struct is_function : std::false_type { };

// specialization for functions
template <typename Ret, typename ... Args>
struct is_function<Ret (Args...)> : std::true_type {};
```

is_function

```
// primary template - for things that are not functions
template <typename T>
struct is_function : std::false_type { };

// specialization for functions
template <typename Ret, typename ... Args>
struct is_function<Ret (Args...)> : std::true_type {};
```

void () // Ret = void, Args = []

is_function

```
// primary template - for things that are not functions
template <typename T>
struct is_function : std::false_type { };

// specialization for functions
template <typename Ret, typename ... Args>
struct is_function<Ret (Args...)> : std::true_type {};
```

```
void () // Ret = void, Args = []
```

```
int (double, char) // Ret = int, Args = [double, char]
```

is_function (C Variadics)

is_function (C Variadics)

```
// specialization for functions that have a C-variadic part
template <typename Ret, typename ... Args>
struct is_function<Ret (Args..., ...) > : std::true_type {};
```

is_function (C Variadics)

```
// specialization for functions that have a C-variadic part
template <typename Ret, typename ... Args>
struct is_function<Ret (Args..., ...) > : std::true_type {};
```

```
// This is the same...
template <typename Ret, typename ... Args>
struct is_function<Ret (Args... ...) > : std::true_type {};
```

is_function (C Variadics)

```
// specialization for functions that have a C-variadic part
template <typename Ret, typename ... Args>
struct is_function<Ret (Args..., ...) > : std::true_type {};
```

```
// This is the same...
template <typename Ret, typename ... Args>
struct is_function<Ret (Args... ...) > : std::true_type {};
```

```
// And so is this...
template <typename Ret, typename ... Args>
struct is_function<Ret (Args.....) > : std::true_type {};
```

is_function (cv-qualifiers)

is_function (cv-qualifiers)



is_function (cv-qualifiers)

```
using F = int (double) const volatile;
```

is_function (cv-qualifiers)

```
using F = int (double) const volatile;
```

- ⁷ The effect of a *cv-qualifier-seq* in a function declarator is not the same as adding cv-qualification on top of the function type. In the latter case, the cv-qualifiers are ignored. [Note: A function type that has a *cv-qualifier-seq* is not a cv-qualified type; there are no cv-qualified function types. — *end note*] [Example:

```
typedef void F();
struct S {
    const F f;           // OK: equivalent to: void f();
};
```

is_function (cv-qualifiers)

```
using F = int (double) const volatile;
```

- Six more specializations (8)
- const, volatile, const volatile
 - Normal function
 - C-variadic function

is_function (ref-qualifiers)

is_function (ref-qualifiers)

```
using F1 = int (double) &;  
using F2 = int (double) &&;
```

is_function (ref-qualifiers)

```
using F1 = int (double) &;  
using F2 = int (double) &&;
```

- Sixteen more specializations (24)
- &, &&
 - Each of existing 8

is_function (noexcept)

is_function (noexcept)

```
using F = int (double) noexcept;
```

is_function (noexcept)

```
using F = int (double) noexcept;
```

- Twenty four more specializations (48)
- noexcept
 - Each of existing 24

is_function (noexcept)

```
using F = int (double) noexcept;
```

- Twenty four more specializations (48)
- noexcept
 - Each of existing 24

<https://twitter.com/ericniebler/status/852192542653329408>