A Soupçon of SFINAE

soupçon *n. A very small quantity of something.*

Arthur O'Dwyer
2017-09-27

Outline

- **Part I: Write your own type traits**
    - Partial specialization and “best match” [3–9]
    - Introducing SFINAE [10–21]
    - `enable_if` maps bools into SFINAE-space [22–25]
- Part II: SFINAE case studies
    - Constructor with a problematic body [27–37]
    - Mutually exclusive overloads [38–46]
    - Problematic parameter type [47–58]
    - Bonus slides: `priority_tag` [60–65]

Write Your Own Type Traits

Value space, type space, and SFINAE space

WYOTT: true_type and false_type

```
template<class Ty, Ty V>
struct integral_constant {
   static constexpr Ty value = V;
   // and some other members that don't matter
};

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

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

WYOTT: partial specialization

```
template<class T> struct is_reference      : false_type {};
template<class T> struct is_reference<T&>  : true_type {};
template<class T> struct is_reference<T&&> : true_type {};

template<class T>
inline constexpr bool is_reference_v = is_reference<T>::value;

static_assert( is_reference_v< int& >);
static_assert( ! is_reference_v< int >);
```

WYOTT: partial specialization

```
template<class T> struct remove_reference       { using type = T; };
template<class T> struct remove_reference<T&>   { using type = T; };
template<class T> struct remove_reference<T&&>  { using type = T; };

template<class T>
using remove_reference_t = typename remove_reference<T>::type;

static_assert( is_same< remove_reference_t< int& >, int >);
static_assert( is_same< remove_reference_t< int >, int >);
```

WYOTT: trouble looms

```
template<class T> struct add_lvalue_reference  { using type = T&; };

template<class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;

static_assert( is_same< add_lvalue_reference_t< int&& >, int& >);
static_assert( is_same< add_lvalue_reference_t< int& >, int& >);
static_assert( is_same< add_lvalue_reference_t< int >, int& >);
```

What’s the trouble with this definition of `add_lvalue_reference`?

WYOTT: trouble looms

```
template<class T> struct add_lvalue_reference  { using type = T&; };

template<class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;

static_assert( is_same< add_lvalue_reference_t< int&& >, int& >);
static_assert( is_same< add_lvalue_reference_t< int& >, int& >);
static_assert( is_same< add_lvalue_reference_t< int >, int& >);
static_assert( is_same< add_lvalue_reference_t< void >, void >);
```

Here we have trouble. When we try to form the type `void&`, the compiler gives us an error and dies. *We must prevent the type `void&` from being formed.*

The wrong fix

```
template<class T> struct add_lvalue_reference  { using type = T&; };
template<> struct add_lvalue_reference<void>   { using type = void; };
template<> struct add_lvalue_reference<const void>    { .......... };
template<> struct add_lvalue_reference<volatile void>     { ....... };
template<> struct add_lvalue_reference<const volatile void>   { ... };
```

This **will** actually work, but it’s tedious to write, and perhaps not future-proof. Wouldn’t it be nice if we could get the compiler to avoid forming the type `T&` **if and only if** that would have caused an error?

Introducing SFINAE

```
template<class T, class Enable>
struct ALR_impl                                 { using type = T; };

template<class T>
struct ALR_impl< T,
    remove_reference_t<T&>                     > { using type = T&; };

template<class T>
struct add_lvalue_reference : ALR_impl<T, remove_reference_t<T>> {};
```

This version uses SFINAE.

```
add_lvalue_ref<int>   ==  impl<int, int>    ==  impl<int, remove_ref_t<int&>>
add_lvalue_ref<int&>  ==  impl<int&, int>   ==  impl<int&, remove_ref_t<int&>>
add_lvalue_ref<void>  ==  impl<void, void>  ==  impl<void, Enable>
```

Introducing SFINAE

```
template<class T, class Enable>
struct ALR_impl                               { using type = T; };

template<class T>
struct ALR_impl< T,
    remove_reference_t<T&>                    > { using type = T&; };

template<class T>
struct add_lvalue_reference : ALR_impl<T, remove_reference_t<T>> {};
```

base template

partial specialization

point of use ("call site")

When we want to use the specialization,
the **`bolded expressions`** in the p.s. and the p.o.u. evaluate to the same type;
and when we don't, the bolded expression in the p.s. is ill-formed, so we use the b.t.

Can we devise a simpler pair of type expressions?

The type-expression in the *p.s.* must be ill-formed exactly when "T&" is ill-formed; and otherwise, it must match the concrete type used at the *p.o.u.*
So we want a type-expression that always produces a simple well-known concrete type, but preserves the "SFINAE-space" well-or-ill-formedness of its parameter.

WYOTT: void_t

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

template<class T, class Enable>
struct ALR_impl                        { using type = T; };

template<class T>
struct ALR_impl<T, void_t<T&>>   { using type = T&; };

template<class T>
struct add_lvalue_reference : ALR_impl<T, void> {};
```

base template

partial specialization

point of use

```
add_lvalue_ref<int>    ==  impl<int, void>    ==  impl<int, void_t<int&>>
add_lvalue_ref<int&>   ==  impl<int&, void>   ==  impl<int&, void_t<int&>>
add_lvalue_ref<void>   ==  impl<void, void>   ==  impl<void, Enable>
```

void_t is a tool for mass production

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

template<class T, class> struct ALR_impl            { using type = T;  };
template<class T> struct ALR_impl<T, void_t<T&>>  { using type = T&; };
template<class T, class> struct ARR_impl            { using type = T;  };
template<class T> struct ARR_impl<T, void_t<T&&>> { using type = T&&;
};
template<class T, class> struct AP_impl             { using type = T;  };
template<class T> struct AP_impl<T, void_t<T*>>    { using type = T*; };

template<class T> struct add_lvalue_reference : ALR_impl<T, void> {};
template<class T> struct add_rvalue_reference : ARR_impl<T, void> {};
template<class T> struct add_pointer : AP_impl<T, void> {};
```

But what if our maybe-ill-formed thing is a value-space expression, not a type-expression? Like not “T&” but something like “a = b”?

WYOTT: `declval`

```
template <class T>
auto declval() noexcept -> add_rvalue_reference_t<T>;
```

declval turns a type into a value, for the purposes of unevaluated expressions. It’s a function with no definition, just a return type. And we use it like this:

```
template <class T, class U>
using assignment_result_t = decltype( declval<T>() = declval<U>() );

static_assert( is_same< assignment_result_t<int&, double>, int& >);
static_assert( is_same< assignment_result_t<int&, int*>, ill-formed >);
```

“Expression SFINAE”

```
template<class T, class U, class Enable>
struct is_assignable_impl                                      : false_type {};

template<class T, class U>
struct is_assignable_impl< T, U,
    decltype(void( declval<T>() = declval<U>() ))    > : true_type {};

template<class T, class U>
struct is_assignable : is_assignable_impl<T, U, void> {};
```

I’ve heard that MSVC has trouble with “expression SFINAE”; but I’ve tested this particular code and MSVC is fine with it.

Microsoft® CERTIFIED

“Expression SFINAE”

I used this formulation to trigger SFINAE:

```
decltype(void( declval<T>() = declval<U>() ))
```

But either of these are fine too:

```
decltype( declval<T>() = declval<U>(), void() )
void_t<decltype( declval<T>() = declval<U>() )>
```

Any of these type-expressions will always evaluate to exactly void, or else SFINAE away.

```
decltype(void(expression))
decltype(expression, void())
void_t<decltype(expression)>
```

Mass production again (1/4)

```
template<class T, class U, class> struct ISC_impl : false_type {};
template<class T, class U> struct ISC_impl<T, U, decltype(void(
    static_cast<U>(declval<T>())
))> : true_type {};

template<class T, class U>
struct is_static_castable : ISC_impl<T, U, void> {};
```

Mass production again (2/4)

```
template<class T, class> struct IP_impl : false_type {};
template<class T> struct IP_impl<T, decltype(
    dynamic_cast<void*>(declval<remove_cv_t<T>*>())
)> : true_type {};

template<class T>
struct is_polymorphic : IP_impl<T, void*> {};
```

Mass production again (3/4)

```
template<class T, class, class...> struct IC_impl : false_type {};
template<class T, class... Us> struct IC_impl<T, decltype(void(
    ::new (declval<void*>()) T(declval<Us>()...)
)), Us...> : true_type {};

template<class T, class... Us>
struct is_constructible : IC_impl<T, void, Us...> {};
```

Mass production again (4/4)

```
template<class T, class, class...> struct INTC_impl : false_type {};
template<class T, class... Us> struct INTC_impl<T, decltype(void(
    ::new (declval<void*>()) T(declval<Us>()...)
)), Us...> : bool_constant<noexcept(
    ::new (declval<void*>()) T(declval<Us>()...)
)> {};

template<class T, class... Us>
struct is_nothrow_constructible : INTC_impl<T, void, Us...> {};
```

WYOTT: `conditional_t`

```
template<bool B, class T, class F>
struct conditional { using type = T; };

template<class T, class F>
struct conditional<false, T, F> { using type = F; };

template<bool B, class T, class F>
using conditional_t = typename conditional<B, T, F>::type;
```

conditional_t is like the ternary operator for type-expressions.
It takes two well-formed types T and F, and produces one or the other of them.
Of course if either T or F is ill-formed, then the whole expression `conditional_t<B,T,F>` will be ill-formed.

WYOTT: `enable_if_t`

```
template<bool B, class T, class F>
struct  enable_if  { using type = T; };

template<class T, class F>
struct  enable_if <false, T, F> { using type = F; };

template<bool B, class T, class F>
using  enable_if_t  = typename  enable_if <B, T, F>::type;
```

conditional_t is like the ternary operator for type-expressions.
It takes two well-formed types T and F, and produces one or the other of them.
Of course if either T or F is ill-formed, then the whole expression `conditional_t<B,T,F>` will be ill-formed.

WYOTT: `enable_if_t`

```
template<bool B, class T = void> struct enable_if { using type = T; };
template<class T> struct enable_if<false, T> {};

template<bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;

template<bool B> using bool_if_t = enable_if_t<B, bool>;
```

`enable_if_t` takes a well-formed type `T` (which defaults to `void`), and produces either `T` or something ill-formed, depending on its boolean argument. So it's a way of converting a value-space boolean `true` or `false` into a SFINAE-space "well-formed" or "ill-formed."

`bool_if_t` is not standard. We'll see a use-case for it in Part 2.

SFINAE, there and back again

Have a boolean `B` that is well-formed, and want to map it into SFINAE-space? Then you do this:

```
enable_if_t<B>
```

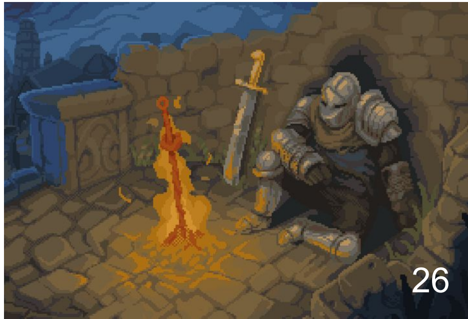
That type is well-formed if `B` is `true`, and ill-formed if `B` is `false`.

Have a value-expression *expr* that might be ill-formed, and want to project its well-formedness into boolean value-space? Then you do this:

```
template<class, Args> struct impl : false_type {};
template<Args> struct impl<decltype(void( expr )), Args> : true_type
{};
template<Args> struct expr_is_well_formed : impl<void, Args> {};
```

Outline

- Part I: Write your own type traits
  - Partial specialization and “best match” [3–9]
  - Introducing SFINAE [10–21]
  - `enable_if` maps bools into SFINAE-space [22–25]
- **Part II: SFINAE case studies**
  - Constructor with a problematic body [27–37]
  - Mutually exclusive overloads [38–46]
  - Problematic parameter type [47–58]
  - Bonus slides: `priority_tag` [60–65]



Case Study 1.

SFINAE away a non-template function

Motivation: Polymorphic memory resources

```
namespace std::pmr {

    class memory_resource;

    memory_resource *get_default_resource();

    template<class T> class polymorphic_allocator;

    template<class T>
    class polymorphic_allocator {
        memory_resource *mr_;
    public:
        polymorphic_allocator() : mr_(get_default_resource()) {}
    };
}
```

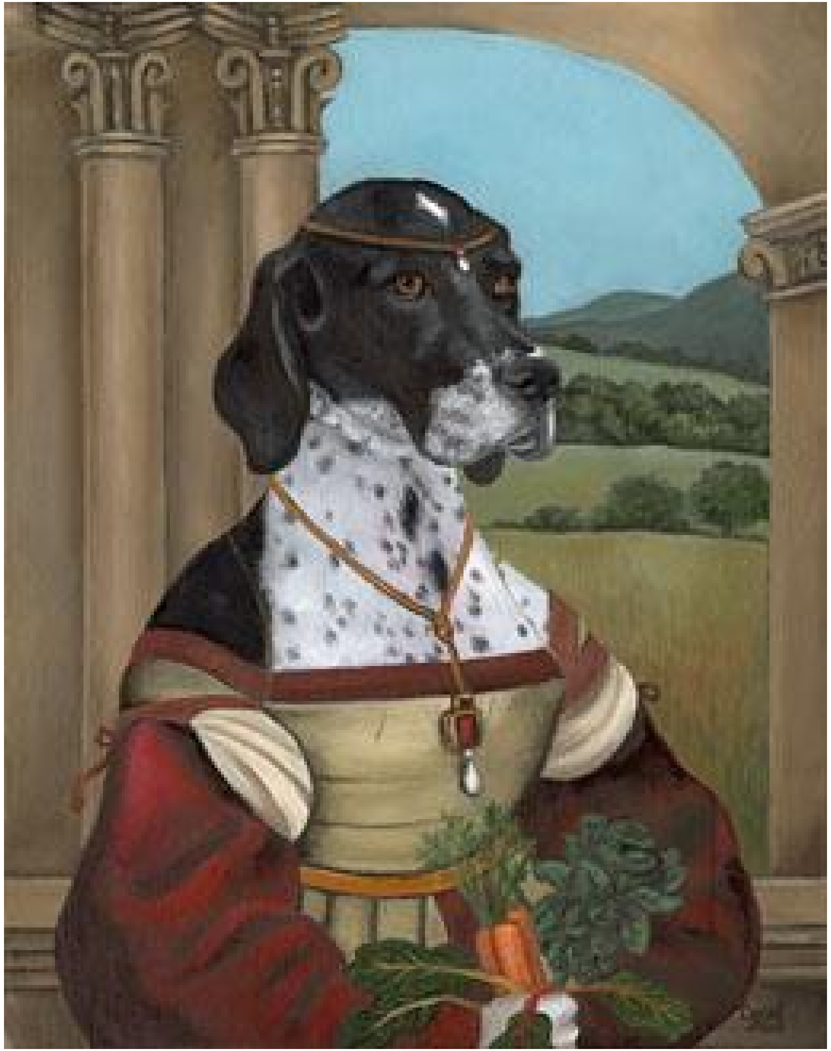
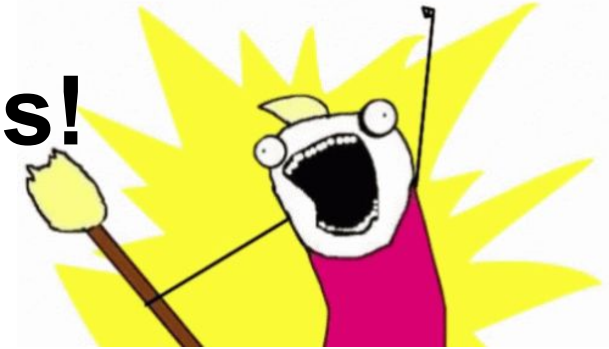


Image credit: carollew.com

Fancy-pointer all the things!



```
template<class VoidPtr>
class fancy_memory_resource;

template<class T, class VoidPtr>
class fancy_poly_allocator;

using memory_resource = fancy_memory_resource<void*>;
template<class T>
using polymorphic_allocator = fancy_poly_allocator<T, void*>;

template<class T> using shmem_ptr = boost::interprocess::offset_ptr<T>;
using shmem_resource = fancy_memory_resource<shmem_ptr<void>>;
template<class T>
using shmem_allocator = fancy_poly_allocator<T, shmem_ptr<void>>;
```

But now we have trouble

```
using memory_resource = fancy_memory_resource<void*>;
memory_resource *get_default_resource();

template<class T, class VoidPtr>
class fancy_poly_allocator {
    fancy_memory_resource<VoidPtr> *mr_;
public:
    fancy_poly_allocator() : mr_(get_default_resource()) {}
};
```

This works only for void*!

```
error: cannot initialize a member subobject of type 'fancy_memory_resource<VoidPtr> *'
with an rvalue of type 'memory_resource *' (aka 'fancy_memory_resource<void *> *')
    fancy_poly_allocator() : mr_(get_default_resource()) {}
                             ^   ~~~~~~~~~~~~~~~~~~~~~~
```

But now we have trouble

This works only for void*!

```
template<class T, class VoidPtr>
class fancy_poly_allocator {
    fancy_memory_resource<VoidPtr> *mr_;
public:
    fancy_poly_allocator() : mr_(get_default_resource()) {}
};
```

We decide to SFINAE away the problem whenever `VoidPtr` is not exactly `void*`.
This is a constructor, so we know we can't SFINAE it away using the return type.
We can use a template parameter, though. Let's try `enable_if`:

```
    template<class = enable_if_t<is_same_v<VoidPtr, void*>>>
    fancy_poly_allocator() : mr_(get_default_resource()) {}
```

Oops!

```
template<class T, class VoidPtr>
class fancy_poly_allocator {
    fancy_memory_resource<VoidPtr> *mr_;
public:
    template<class = enable_if_t<is_same_v<VoidPtr, void*>>>
    fancy_poly_allocator() : mr_(get_default_resource()) {}
};
```

```
type_traits: error: no type named 'type' in 'enable_if<false, void>';
'enable_if' cannot be used to disable this declaration
template <bool B, class T = void> using enable_if_t = typename enable_if<B, T>::type;
                                                                         ^
note: in instantiation of template type alias 'enable_if_t' requested here
        template<class = enable_if_t<is_same_v<VoidPtr, void*>>>
                         ^
```

Explanation and solution

It doesn't work because there is nothing in that `enable_if_t` that depends on the *p.o.u*. The compiler will evaluate template default arguments eagerly whenever possible. So if we want to delay the evaluation of the template argument, we have to put something in it that depends on the *p.o.u*.

The compiler isn't smart enough to know that there is no way for the user to explicitly specify constructor template arguments. So we can do this mechanical transformation:

```
    template<class = enable_if_t<is_same_v<VoidPtr, void*>>>
    fancy_poly_allocator() : mr_(get_default_resource()) {}

    template<bool B = is_same_v<VoidPtr, void*>, class =
enable_if_t<B>>
    fancy_poly_allocator() : mr_(get_default_resource()) {}
```

It works!

Now we can instantiate `fancy_poly_allocator<T, shmem_ptr<void>>`, no problem.

```
polymorphic_allocator<int> vanilla;          // ok
shmem_resource *res = ...;
shmem_allocator<int> footwork(res);          // ok
shmem_allocator<int> poodle;                 // error, as desired
```

```
error: no matching constructor for initialization of 'shmem_allocator<int>'
(aka 'fancy_polymorphic_allocator<int, boost::interprocess::offset_ptr<void> >')
    shmem_allocator<int> poodle;
                         ^
note: candidate template ignored: requirement 'false' was not satisfied [with B =
false]
        fancy_polymorphic_allocator() : mr_(get_default_resource()) {}
        ^
```

Improve the error message

```
    template<class = enable_if_t<is_same_v<VoidPtr, void*>>>
```

```
(too eager, doesn't work at all)
```

```
    template<bool B = is_same_v<VoidPtr, void*>, class = enable_if_t<B>>
```

```
requirement 'false' was not satisfied [with B = false]
```

```
    template<bool B = true,
             class = enable_if_t<B && is_same_v<VoidPtr, void*>>>
```

```
requirement 'is_same_v' was not satisfied [with B = true]
```

Recommended
Only substitute one template parameter, not all of them.

```
    template<class VoidPtr_ = VoidPtr,
             class = enable_if_t<is_same_v<VoidPtr_, void*>>>
```

```
requirement 'is_same_v<boost::interprocess::offset_ptr<void, long, unsigned long, 0>, void *>'
was not satisfied [with VoidPtr_ = boost::interprocess::offset_ptr<void, long, unsigned long,
0>]
```

https://wandbox.org/permlink/XsBp8Tfty2gzgSIH

Lesson 1

Add a level of indirection

Case Study 2.

Conditional `explicit`

Motivation: *EXPLICIT* constructors

```
template<class T>
class offset_ptr {
    uint m_ptr;
public:
    explicit offset_ptr(T *p) { m_ptr = uint(p) - uint(this); }

    T *ptr() const noexcept { return (T *)(uint(this) + m_ptr); }

    template<class U>  // whenever U* is convertible to T*
    EXPLICIT offset_ptr(const offset_ptr<U>& rhs);
};
```

Motivation: *EXPLICIT* constructors

```
template<class T>
class offset_ptr {
    uintptr_t m_ptr;
public:
    explicit offset_ptr(T *p) { m_ptr = uint(p) - uint(this); }

    T *ptr() const noexcept { return (T *)(uint(this) + m_ptr); }

    template<class U>  // if U* is implicitly convertible to T*
    offset_ptr(const offset_ptr<U>& rhs) : offset_ptr(rhs.ptr()) {}

    template<class U>  // if U* is convertible to T* with a static_cast
    explicit offset_ptr(const offset_ptr<U>& rhs) :
        offset_ptr(static_cast<T *>(rhs.ptr())) {}
};
```

We need more than *comments*.

```
template<class U>  // if U* is implicitly convertible to T*
offset_ptr(const offset_ptr<U>& rhs) : offset_ptr(rhs.ptr()) {}


template<class U>  // if U* is convertible to T* with a static_cast
explicit offset_ptr(const offset_ptr<U>& rhs) :
    offset_ptr(static_cast<T *>(rhs.ptr())) {}
```

```
error: constructor cannot be redeclared
        explicit offset_ptr(const offset_ptr<U>& rhs) :
                 ^
note: previous definition is here
        offset_ptr(const offset_ptr<U>& rhs) :
        ^
```

Comments ➡ code... same error!

```
template<class U, class = enable_if_t<is_convertible_v<U*, T*>>>
offset_ptr(const offset_ptr<U>& rhs) : offset_ptr(rhs.ptr()) {}

template<class U, class = enable_if_t<is_static_castable_v<U*, T*>
                                      && !is_convertible_v<U*, T*>>>
explicit offset_ptr(const offset_ptr<U>& rhs) :
    offset_ptr(static_cast<T *>(rhs.ptr())) {}
```

```
error: constructor cannot be redeclared
        explicit offset_ptr(const offset_ptr<U>& rhs) :
                 ^
note: previous definition is here
        offset_ptr(const offset_ptr<U>& rhs) :
        ^
```

Explanation and solution

It doesn't work because the *signature* of the constructor is still the same in both cases.

```
template<class U, class = enable_if_t<is_convertible_v<U*, T*>>>
offset_ptr(const offset_ptr<U>& rhs) : ...

template<class U, class = enable_if_t<is_static_castable_v<U*, T*>
                                      && !is_convertible_v<U*, T*>>>
explicit offset_ptr(const offset_ptr<U>& rhs) : ...
```

We give the second `class` parameter a new default value in the second declaration, but that's merely a violation of the One Definition Rule, not a change in the signature that would indicate that this is a separate template.

Since we want to have two independently SFINAEable templates here, we need to make sure they have different parameter lists, somehow.

The wrong fix

```
template<class U, class = enable_if_t<is_convertible_v<U*, T*>>>
offset_ptr(const offset_ptr<U>& rhs, int=0) : offset_ptr(rhs.ptr()) {}

template<class U, class = enable_if_t<is_static_castable_v<U*, T*>
                                      && !is_convertible_v<U*, T*>>>
explicit offset_ptr(const offset_ptr<U>& rhs, double=0) :
    offset_ptr(static_cast<T *>(rhs.ptr())) {}
```

This does actually work, but it’s not a great idea. We have to invent a new ad-hoc “tag type” (`int`, `double`, ...) for each mutually exclusive overload, and default function arguments are awful anyway.

A better but subtler solution is to change the *template* parameter lists.

Explanation and solution

```
template<class U, bool_if_t<is_convertible_v<U*, T*>> = true>
offset_ptr(const offset_ptr<U>& rhs) : offset_ptr(rhs.ptr()) {}

template<class U, bool_if_t<is_static_castable_v<U*, T*>
                              && !is_convertible_v<U*, T*>> = true>
explicit offset_ptr(const offset_ptr<U>& rhs) :
    offset_ptr(static_cast<T *>(rhs.ptr())) {}
```

The subtlety here is that the type-expression `bool_if_t<`*expr*`>` can't be evaluated until we know the value of *expr*, which in this case depends on the *p.o.u.* because it depends on `U`. At the *p.o.u.* the compiler will compute the overload set, which means evaluating `bool_if_t<`*expr*`>`; and if *expr* is false then SFINAE will kick in and that template will never be included in the overload set.

Lesson 2

To kick an overload out of your overload set, put the ill-formed thing *somewhere that affects the mangling / signature*.

Case Study 3.

Problematic parameter types

Motivation: pointer_traits<P>

```
template<class P> struct pointer_traits;

template<class T>
struct pointer_traits<T*>
{
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;

    template<class U> using rebind = U*;

    static auto pointer_to(T& r) {
        return &r;
    }
};
```

There is a **base template**, but its definition isn't relevant to this case study.

This is the **partial specialization** for native pointer types.

Motivation: pointer_traits<P>

```
template<class P> struct pointer_traits;

template<class T>
struct pointer_traits<T*>
{
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;

    template<class U> using rebind = U*;

    static auto pointer_to(T& r) {
        return &r;
    }
};
```

There is a **base template**, but its definition isn't relevant to this case study.

This is the **partial specialization** for native pointer types.

This works for every type except void!

Oops!

```
template<class T>
struct pointer_traits<T*>
{

    static auto pointer_to(T& r) {
        return &r;
    }

};
```

```
error: cannot form a reference to 'void'
    static auto pointer_to(T& r) {
                             ^
```

Oops!

We follow our recipe for SFINAE'ing away a non-template function...

```
template<class T>
struct pointer_traits<T*>
{
    template<bool B = is_void_v<T>, class = enable_if_t<!B>>
    static auto pointer_to(T& r) {
        return &r;
    }
};
```

...but the compiler error message doesn't change!

```
error: cannot form a reference to 'void'
    static auto pointer_to(T& r) {
                            ^
```

False starts (1/2)

It doesn't work because the function signature involves `T&`, which is eagerly evaluated. We need to find a way to prevent `T&` from evaluating until we see the *p.o.u*. The first thing that jumps to mind is this:

```
template<bool B = is_void_v<T>, class TR = enable_if_t<!B, T&>>
static auto pointer_to(TR r) {
    return &r;
}
```

Same error.

```
error: cannot form a reference to 'void'
    template<bool B = is_void_v<T>, class TR = enable_if_t<!B, T&>>
                                                               ^
```

False starts (2/2)

Okay, let's make sure the `T` has to remain separated from the & until p.o.u. time.
How about...

```
template<bool B = is_void_v<T>, class TR = enable_if_t<!B, T>&>
static auto pointer_to(TR r) {
    return &r;
}
```

Finally, it compiles! But...

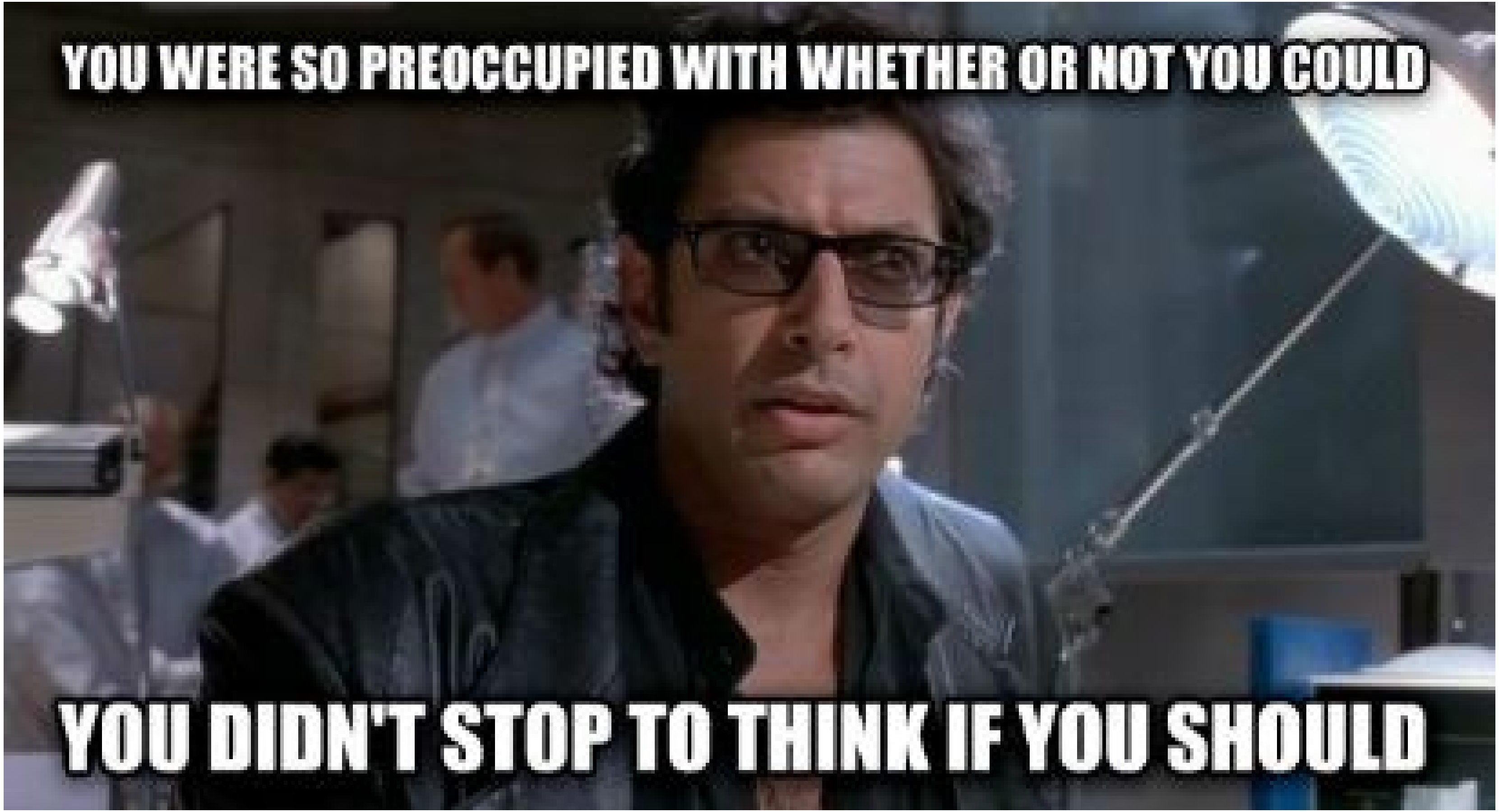
False starts (2/2)

Okay, let's make sure the `T` has to remain separated from the & until p.o.u. time.
How about...

```
template<bool B = is_void_v<T>, class TR = enable_if_t<!B, T>&>
static auto pointer_to(TR r) {
    return &r;
}
```

Finally, it compiles! But...

When we run our code, it segfaults!


YOU WERE SO PREOCCUPIED WITH WHETHER OR NOT YOU COULD
YOU DIDN'T STOP TO THINK IF YOU SHOULD

False starts (2/2)

Why does `pointer_traits<int*>::pointer_to(x)` now segfault?

```
template<bool B = is_void_v<T>, class TR = enable_if_t<!B, T>&>
static auto pointer_to(TR r) {
    return &r;
}
```

The problem is that we have forgotten to pay attention to the meaning of the syntax we're using. We provided a *default* for the template type parameter `TR`; but defaults are used only when the type cannot be deduced, and in this case it *can* be deduced — **as a non-reference type!**

What we need is not a new *template* parameter, but instead a constraint on the existing *function* parameter to match either `T&` or SFINAE'd-away-entirely, never anything else.

Explanation and solution

```
template<bool B = is_void_v<T>>
static auto pointer_to(enable_if_t<!B, T>& r) {
    return &r;
}
```

This works — *and* has the right behavior!

When T is not `void`, the call-site of `pointer_to(x)` sees that `B` was not provided, so it uses `B`'s default value of `false`, and produces a function parameter type of exactly `T&`.

When T is `void`, there are no call-sites. The parameter type `enable_if_t<!B, void>&` doesn't cause any problems, because it cannot be eagerly evaluated any further than that.

Lesson 3

Don’t lose sight of the meaning of your code.

Bonus Case Study 4.

Many non-mutually-exclusive dispatch cases

Hierarchical tag dispatch

```
template<class It>
auto impl(It first, It last, random_access_iterator_tag) {
    return last - first;                        // More specific
}

template<class It>
auto impl(It first, It last, input_iterator_tag) {
    iterator_difference_t<It> n = 0;           // Less specific
    while (first != last) ++first, ++n;
    return n;
}

template<class It> auto distance(It first, It last) {
    return distance_impl(first, last, typename It::iterator_category{});
}
```

Works because a forward iterator tag **IS-A** input iterator tag. A random access iterator tag also **IS-A** input iterator tag, but the first overload matches better because the required conversion-to-base-class is “fewer levels deep.”

Make up the hierarchy from outside

```
template<size_t I> struct priority_tag : priority_tag<I-1> {};
template<> struct priority_tag<0> {};

template<class It, class = enable_if_t<is_random_access_iterator_v<It>>>
auto impl(It first, It last, priority_tag<1>) {
    return last - first;                          // More specific
}

template<class It>                                // Less specific
auto impl(It first, It last, priority_tag<0>) { ... }

template<class It> auto distance(It first, It last) {
    return distance_impl(first, last, priority_tag<1>{});
}
```

Works because a priority tag<1> **IS-A** priority tag<0>. The first overload matches better because the required conversion-to-base-class is “fewer levels deep.”

Sometimes we *must* make up the hierarchy

```
template<class A> auto allocator_pointer(priority_tag<2>)   // More specific
    -> typename A::pointer;
template<class A> auto allocator_pointer(priority_tag<1>)   // Less specific
    -> typename A::value_type*;
template<class A> auto allocator_pointer(priority_tag<0>)   // Least specific
    -> decltype(declval<A&>().allocate(0));

template<class A> using allocator_pointer_t =
    decltype(allocator_pointer<A>(priority_tag<2>{}));

template<class A>
struct allocator_traits {
   using allocator_type = A;
   using value_type = detail::allocator_value_type_t<A>;
   using pointer = detail::allocator_pointer_t<A>;
  // ...
};
```

For another example, see the code from my talk "dynamic_cast from scratch."

Lesson 4

Use `priority_tag` to control prioritized tag dispatch.

’17: Use `if constexpr`... but WYOTT!

```
template<class It> using iterator_category_t = typename iterator_traits<It>::iterator_category;

template<class, int> struct IRA_impl : false_type {};
template<class It> struct IRA_impl<It, (random_access_iterator_tag(iterator_category_t<It>{}), 0)> : true_type {};
template<class It> struct is_random_access_iterator : IRA_impl<It, 0> {};
template<class It> inline constexpr bool is_random_access_iterator_v = is_random_access_iterator<It>::value;
```

```
template<class It>
auto distance(It first, It last)
{
    if constexpr (is_random_access_iterator_v<It>) {
        return last - first;
    } else {
        iterator_difference_t<It> n = 0;
        while (first != last) ++first, ++n;
        return n;
    }
}
```

Q&A

Thanks for coming!