

# Modern Template Metaprogramming: A Compendium

Revised and expanded edition

WALTER E. BROWN, PH.D.

`<webrown.cpp @ gmail.com>`

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

## Abstract

---

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


## A little about me

---

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for almost 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Lectured internationally as a software consultant and commercial trainer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still available for consulting work. (Email me!)



## Emeritus participant in C++ standardization

- Written 85+ papers for WG21, introducing such now-standard C++ library features as `cbegin/cend` and `common_type`, as well as the entirety of headers `<random>` and `<ratio>`. 
  - Heavily influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*.
  - Conceived and served as Project Editor for ISO/IEC 29124 (Int'l Standard on Mathematical Special Functions in C++); now serving as an Associate Project Editor for C++17.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! 😊

## Preview of examples (not in order)

- From the `std::` library:
  - `integral_constant`, `true_type`, `false_type`
  - `is_same`, `is_void`, `is_integral`, `is_floating_point`, `is_signed`
  - `is_copy_assignable`, `is_move_assignable`
  - `remove_const`, `remove_volatile`, `remove_cv`
  - `conditional`, `enable_if`
  - `distance`
- Not (yet?) from the `std::` library:
  - `abs`, `gcd`
  - `type_is`, `bool_constant`
  - `is_one_of`
  - `void_t (!)`, `has_type_member`, `is_valid`, `is_complete`

# What is template metaprogramming?

- “Metaprogramming is the writing of computer programs:
  - ① “That write or manipulate other programs (or themselves) as their data, or
  - ② “That do ... work at compile time that would otherwise be done at runtime” [Wikipedia].
- C++ template metaprogramming uses template instantiation to drive compile-time evaluation:
  - When we use the name of a template where a {function, type, variable} is expected, the compiler will instantiate (create) the expected entity from that template.
    - Example: a call `f(x)` where `f` names a function template.
  - Template metaprogrammers exploit this machinery to improve
    - ① source code flexibility and ② runtime performance.

## Representative timings (2001)

	std::pow( )	pow<> v1	pow<> v2
real	11.858 s	8.081 s	3.035 s
user	11.837 s	8.081 s	3.024 s
sys	0.020 s	0.020 s	0.030 s

- Measured  $x^{50}$  repeated 10,000,000 times.
- Used gcc 2.95.2, -O0, on 700 MHz PIII, Win2K.

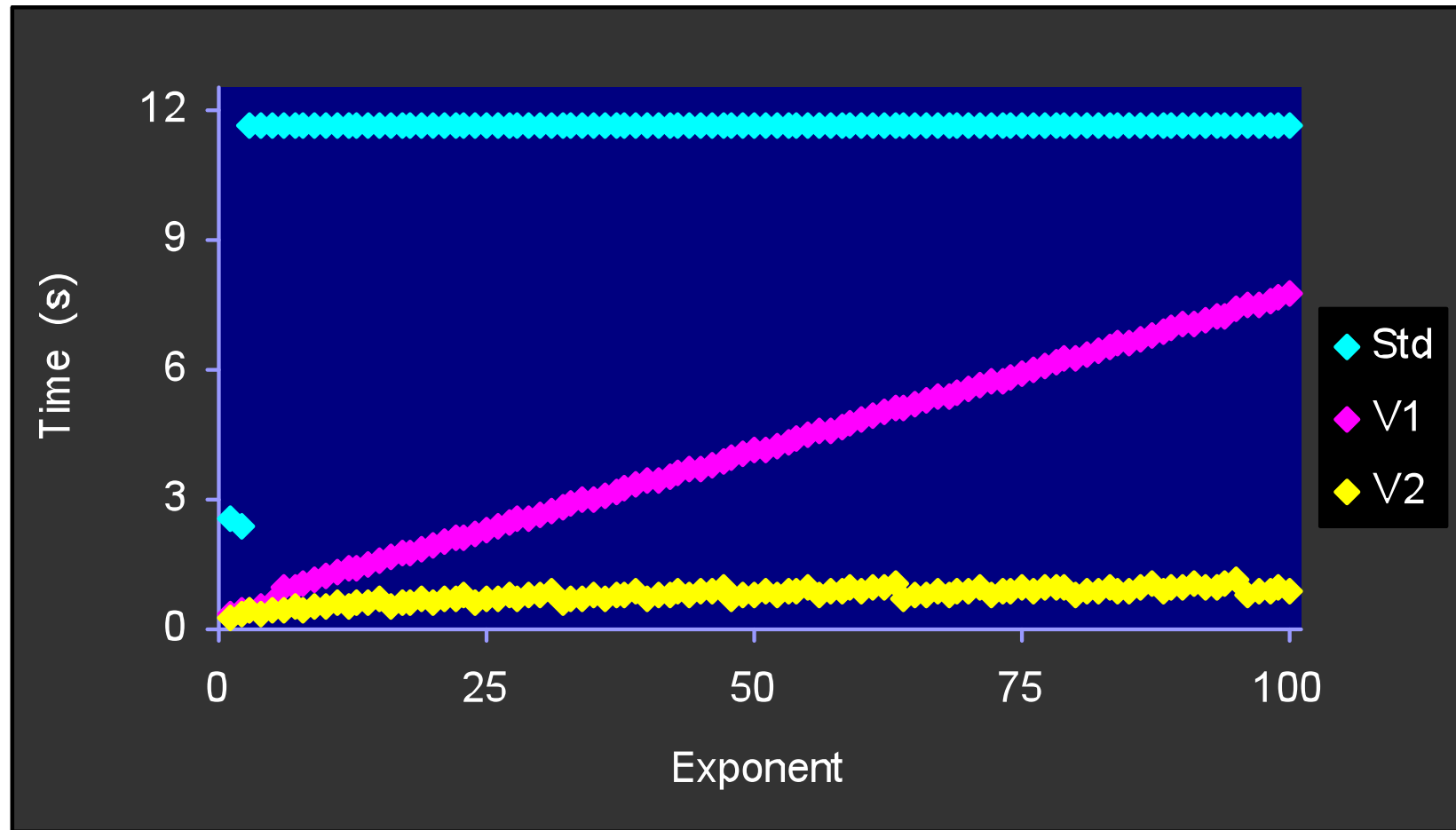
## Timings with optimization (2001)

	std::pow( )	pow<> v1	pow<> v2
real	11.857 s	4.286 s	0.300 s
user	11.847 s	4.236 s	0.190 s
sys	0.020 s	0.010 s	0.010 s

- Used g++ -O2 to compile; otherwise identical.
- Improvements: ~ 47% for v1; ~ 90% for v2!



## Performance for increasing powers (2001)



## When metaprogramming ...

- Keep in mind that run-time == compile-time, so can't rely on: ☒ mutability, ☒ virtual functions, ☒ other RTTI, *etc.*
- To put it simply,



## How to shift work to compile-time

- Example: a compile-time absolute value metafunction:

- `template< int N > // template param used as the metafcn param`  
`struct abs {`  
    `static_assert( N != INT_MIN ); // C++17-style guard`  
    `static constexpr auto value = (N < 0) ? -N : N; // "return"`  
`};`

- Usage (a metafunction call):

- Metafunction arg(s) are supplied as the template's arg(s).
  - "Call" syntax is a request for the template's published value.
  - `int const n = ... ; // could instead declare as constexpr`  
`... abs<n>::value ... // instantiation yields a compile-time constant`

## A C++11 constexpr function can be useful, but ...

- Example: a compile-time absolute value function:
  - `constexpr auto abs( int N ) { return ( N < 0 ) ? - N : N; }`
- Usage is via familiar function call syntax:
  - `int const n = ... ;    // could instead declare as constexpr`  
`... abs( n ) ...        // yields a compile-time constant`
- But, as structs, metafunctions offer us more tools, *e.g.*:
  - Public member type declarations (*e.g.*, typedef or using).
  - Public member data declarations (static const/constexpr, each initialized via a constant expression).
  - Public member function declarations and constexpr member function definitions.
  - Public member templates, static\_asserts, and more!

## Compile-time recursion with specialization as base

- Example: primary template for  $gcd(m, n)$  metafunction (compile-time *greatest common divisor* calculation):

- ```
template< unsigned M, unsigned N >
    struct gcd {    // per Euclid
        static constexpr auto value = gcd<N, M%N>::value;
    };

```

- Much like pattern matching, this partial specialization recognizes the (base) case  $gcd(m, 0)$ :

- ```
template< unsigned M >
    struct gcd<M, 0> {
        static_assert( M != 0 );    // gcd(0, 0) is undefined, so disallow
        static constexpr auto value = M;
    };

```

## A metafunction can take a type as a parameter/argument

- sizeof is a built-in type function, but we can write our own.
- Example: obtain the (compile-time) rank of an array type:
  - *// primary template handles scalar (non-array) types as base case:*  
template< class T >  
struct rank { static constexpr size\_t value = 0u; };
  - *// partial specialization recognizes any array type:*  
template< class U, size\_t N >  
struct rank< U[N] > {  
 static constexpr size\_t value = 1u + rank<U>::value;  
};
- Usage:
  - using array\_t = int [10] [20] [30];  
... rank<array\_t>::value ... *// yields 3u (at compile-time)*

## A metafunction can also produce a type as its result

- Example: “remove” a type’s const-qualification:
  - No real removal; “give me the equivalent type without const.”
  - *// primary template handles types that are not const-qualified:*  
template< class T >  
struct remove\_const { using type = T; }; *// identity*
  - *// partial specialization recognizes const-qualified types:*  
template< class U >  
struct remove\_const< U const > { using type = U; };
- Usages (call syntax):
  - remove\_const<T>::type t; *// ensure t has a mutable type*
  - remove\_const\_t<T> t; *// C++14 equivalent; more later*

## C++11 library metafunction convention #1

- A metafunction with a type result aliases that result to type:
  - Except for a few `std::` metafunctions predating this convention.
  - *E.g.*, `iterator_traits` has 5 type results; none is named type.
- Example: an identity metafunction:
  - `template< class T >`  
    `struct type_is { using type = T; };`
- Convenient to apply the convention via public inheritance:
  - *// primary template handles types that are not volatile-qualified:*  
    `template< class T >`  
    `struct remove_volatile : type_is< T > { };` *// identity*
  - *// partial specialization recognizes volatile-qualified types:*  
    `template< class U >`  
    `struct remove_volatile< U volatile > : type_is< U > { };`



## Compile-time decision-making

- Imagine a metafunction, IF/IF\_t, to select one of two types:
  - `template< bool p, class T, class F >`  
`struct IF : type_is< ... > { }; // p ? T : F`
- Such a facility would let us write self-configuring code:
  - Assume: `int const q = ... ; // user's configuration parameter`
  - `IF_t< (q<0), int, unsigned > k;`  
*// declare k to have 1 of these 2 integer types*
  - `IF_t< (q<0), F, G >{ }( ... )`  
*// instantiate and call 1 of these 2 function objects*
  - `class D : public IF_t< (q<0), B1, B2 > { ... };`  
*// inherit from 1 of these 2 base classes*

## Behind the scenes of IF

- Straightforward to implement:
  - *// primary template assumes the bool value is true:*  
template< bool, class T, class > *// needn't name unused param's*  
struct IF : type\_is< T > { };
  - *// partial specialization recognizes a false value:*  
template< class T, class F >  
struct IF<false, T, F> : type\_is< F > { };
- This IF is in C++11, named conditional:
  - Augmented (C++14) by a convenience call alias, conditional\_t.
  - (All the C++14 standard type-returning traits have an analogous ...\_t convenience metafunction call alias.)

## A single-type variation on conditional

- “If true, use the given type; if false, use no type at all”:
  - *// primary template assumes the bool value is true:*  
template< bool, class T = void >      *// default is useful, not essential*  
struct enable\_if : type\_is< T > { };  
  
▪ *// partial specialization recognizes a false value, computing nothing:*  
template< class T >  
struct enable\_if<false, T> { };      *// no member named type!*
- Now consider a meta-call `enable_if< false, ... >::type :`
  - Always an error, right?
  - No, only sometimes an error: SFINAE!
  - SFINAE: Substitution Failure Is Not An Error.  
(Also sometimes termed explicit overload set management.)

## SFINAE applies during implicit template instantiation

- During template instantiation, the compiler will:
  - ① Obtain (or figure out) the template arguments:
    - Taken verbatim if explicitly supplied at template's point of use.
    - Else deduced from function arguments, if any, at point of call.
    - Else taken from the declaration's default template arguments.
  - ② Replace each template parameter, throughout the template, by its corresponding template argument.
- If these steps produce well-formed code, the instantiation succeeds, but ...
- If the resulting code is ill-formed, it is considered not viable (due to substitution failure) and is silently discarded.

## SFINAE in use

---

- Example: want one algorithm `f` taking integral types `T`, and overload it with a second `f` taking floating-point types `T`.
- For a given type `T`, want at most one of the two algorithms to be instantiated, so explicitly manage the overload set:
  - `template< class T >`  
  `enable_if_t< is_integral<T>::value, maxint_t >`  
  `f ( T val ) { ... };`
  - `template< class T >`  
  `enable_if_t< is_floating_point<T>::value, long double >`  
  `f ( T val ) { ... };`
- What if neither overload were viable?
  - Calling `f` with, say, a string argument produces an ill-formed program since both candidates will be SFINAE'd away.

## A taste of the future

- *Concepts Lite* seems likely to be published (as a TS) in 2015, and thence perhaps to be integrated with C++17:
  - Its “constraints” metaprogramming feature seems likely to reduce or obviate many current uses for SFINAE, *etc.*
  - Based on decades of concepts work by A. Stepanov, inspired by the founder of abstract algebra, Emmy Noether (1882-1935).
- Revisiting part of our SFINAE example:
  - ```
template< class T >
  enable_if_t< is_integral<T>::value, maxint_t >      // SFINAE
  f ( T val ) { ... };
```
  - ```
template< Integral T >      // constrained template (short form)
  maxint_t
  f ( T val ) { ... };
```

## C++11 library metafunction convention #2

- A metafunction with a value result has:
  - A static constexpr member, value, giving its result, and ...
  - A few convenience member types and constexpr functions.
- Canonical C++11 value-returning metafunction:
  - ```
template< class T, T v >
struct integral_constant {
    static constexpr T value = v;
    constexpr operator T ( ) const noexcept { return value; }
    constexpr T operator ( ) ( ) const noexcept { return value; }
    ... // remaining members are only occasionally useful
};
```
  - Inheriting from integral\_constant provides more options for meta-call syntax (details in just a moment).

## Revised rank metafunction

- Example: obtain the (compile-time) rank of an array type:
  - *// primary template handles scalar (non-array) types as base case:*  
template< class T >  
struct rank : integral\_constant< size\_t, 0u > { };
  - *// partial specialization recognizes bounded array types:*  
template< class U, size\_t N >  
struct rank< U[N] >  
: integral\_constant< size\_t, 1u + rank<U>::value > { };
  - *// partial specialization recognizes **un**bounded array types:*  
template< class U >  
struct rank< U[ ] >  
: integral\_constant< size\_t, 1u + rank<U>::value > { };



## Some integral\_constant conveniences

- A useful convenience alias:
  - `template< bool b >`  
`using bool_constant = integral_constant<bool, b>;`
- Some useful C++11 convenience aliases:
  - `using true_type = bool_constant<true>;`
  - `using false_type = bool_constant<false>;`
- Value-returning metafunction calls have evolved:
  - `is_void<T>::value` *// since Technical Report 1*
  - `bool( is_void<T>{ } )` *// instantiate/cast; since C++11*
  - `is_void<T>{ }( )` *// instantiate/call; since C++14*
  - `is_void_v<T>` *// a C++14 variable template;*  
*// planned for C++17 standard library*

## Using inheritance + specialization together

- Example 1: given a type, is it a void type?
  - *// primary template handles non-void types:*  
template< class T > struct is\_void : false\_type { };
  - *// four specializations, one to recognize each of the four void types:*  
template< > struct is\_void<void> : true\_type { };  
template< > struct is\_void<void const> : true\_type { };  
⋮
- Example 2: given two types, are they one and the same?
  - *// primary template handles distinct types:*  
template< class T, class U > struct is\_same : false\_type { };
  - *// partial specialization recognizes identical types:*  
template< class T > struct is\_same<T, T> : true\_type { };

## Aliasing == delegation + binding

- Example: given a type, is it a void type?
  - ```
template< class T >
    using is_void = is_same< remove_cv_t<T>
                          , void
                          >;
```
- Where `remove_cv` and `remove_cv_t` are simply:
  - ```
template< class T >
    using remove_cv = remove_volatile< remove_const_t<T> >;
```
  - ```
template< class T >
    using remove_cv_t = typename remove_cv<T>::type;
```

## Dispatching to best-performing algorithm

- Example: The performance of `std::distance` depends on its iterator's capabilities:
  - `template< class Iter >`  
`auto`  
`distance( Iter b, Iter e, true_type ) { return e - b; } // O(1)`
  - `template< class Iter >`  
`auto`  
`distance( Iter b, Iter e, false_type ) { /* loop */ } // O(N)`
  - `template< class Iter >`  
`inline auto`  
`distance( Iter b, Iter e )`  
`{ return distance(b, e, is_random_access_iter_t<Iter>{ } ); }`
- The standard library is more finely grained than this.

## Dispatching via iterator tags

- Via `iterator_traits`, each iterator is associated with a type (tag) denoting its capabilities (category):
  - `template< class Iter >`  
`auto`  
    `distance( Iter b, Iter e, random_access_iterator_tag );`
  - `template< class Iter >`  
`auto`  
    `distance( Iter b, Iter e, input_iterator_tag );`
  - `template< class Iter >`  
`inline auto`  
    `distance( Iter b, Iter e )`  
`{ return distance( b, e`  
                        `, iterator_traits<Iter>::iterator_category{ }`  
                        `); }`

## Using a parameter pack in a metafunction

- Example: generalize `is_same` into `is_one_of`:
  - *// primary template: is T the same as one of the types P0toN... ?*  
`template< class T, class... P0toN >`  
`struct is_one_of;                               // declare the interface only`
  - *// base #1: specialization recognizes empty list of types:*  
`template< class T >`  
`struct is_one_of<T> : false_type { };`
  - *// base #2: specialization recognizes match at head of list of types:*  
`template< class T, class... P1toN >`  
`struct is_one_of<T, T, P1toN...> : true_type { };`
  - *// specialization recognizes mismatch at head of list of types:*  
`template< class T, class P0, class... P1toN >`  
`struct is_one_of<T, P0, P1toN...>`  
`: is_one_of<T, P1toN...> { };`                               *// go inspect list's tail*

## Re-visiting is\_void

---

- Example: given a type, is it a void type?
  - ```
template< class T >
    using is_void = is_one_of< T
        , void
        , void const
        , void volatile
        , void const volatile
    >;
```

## Unevaluated operands

- Recall that operands of `sizeof`, `alignof`, `typeid`, `decltype`, and `noexcept` are never evaluated, not even at compile time:
  - Implies that no code is generated (in these contexts) for such operand expressions, and ...
  - Implies that we need a declaration only, not a definition, to use a (function's or object's) name in these contexts.
- An unevaluated function call (*e.g.*, `to foo`) can usefully map one type to another:
  - `decltype( foo( declval<T>( ) ) )`      *// declval is in <utility>*  
      *// gives foo's return type, were it called with a T rvalue*
  - The unevaluated call `std::declval<T>( )` is declared to give an rvalue result of type `T`. (Use `std::declval<T&>( )` for lvalue.)



## Example: testing for copy-assignability

- *// helper alias for the result type of a valid copy assignment:*

```
template< class T >  
using copy_assign_t  
    = decltype( declval<T&>( ) = declval< T const& >( ) );
```

- ```
template< class T >  
struct is_copy_assignable {  
private:
```

```
    template< class U, class = copy_assign_t<U> >  
    static true_type  
        try_assign( U&& );           // SFINAE may apply!
```

```
    static false_type  
        try_assign( ... );           // catch-all overload
```

```
public:  
    using type = decltype( try_assign( declval<T>( ) ) );  
};
```

## Older technique

---

- Before C++11 introduced `decltype`, `sizeof` was (ab?)used to obtain an unevaluated metaprogramming context:
  - Useful to keep this in mind when reading earlier (C++98/C++03) template metaprogramming code.
- In detail:
  - ① In place of `true_type/false_type`, overloads' return types were crafted to have distinct sizes, *e.g.*,  
`typedef char (&yes) [1];` and `typedef char (&no) [2];` .
  - ② As before, call the overloaded function in an unevaluated context: `sizeof( try_assign( ... ) )`.
  - ③ To determine which overload was chosen, embed that call:  
`typedef bool_constant< sizeof( ... ) == sizeof(yes) > type;` .

## 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?

## Utility of void\_t

- Acts as a metafunction call that maps any well-formed type(s) into the (predictable!) type void:
  - Initially devised as an implementation detail while proposing SFINAE-friendly versions of `common_type` and `iterator_traits`.
  - But “a method is a device which you used twice.” — G. Polya
- Example: detect the presence/absence of a valid type member named `T::type` (per the metafunction convention):
  - ① *// primary template:*  
`template< class, class = void >`  
`struct has_type_member : false_type { };`
  - ② *// partial specialization:*  
`template< class T >`  
`struct has_type_member< T, void_t< typename T::type > >`  
`: true_type { };`

## In detail

---

- Called via `has_type_member<T>::value` or equivalent.
- When type `T` does have a valid type member named `type`:
  - ② `template< class T >`  
    `struct has_type_member< T, void_t< typename T::type > >`  
    `: true_type { };`
    - This specialization is well-formed and thus viable (despite a funny spelling of the second argument, `void`); as the more-specialized candidate, it will be selected by the compiler.
- When type `T` does not have a type member named `type`:
  - `T::type` is ill-formed, so the specialization ② is nonviable (SFINAE!); as the only viable candidate, primary template ① will be selected.
- ① `template< class, class = void >    // default argument is essential`  
    `struct has_type_member : false_type { };`

## Revisiting is\_copy\_assignable

- *// recall this helper alias for the result type of a valid copy assignment:*  
template< class T >  
using copy\_assign\_t  
= decltype( declval<T&>( ) = declval<T const&>( ) );
- *// primary template handles all non-copy-assignable types:*  
template< class T, class = void > // default argument is essential  
struct is\_copy\_assignable : false\_type { };
- *// specialization recognizes and validates only copy-assignable types:*  
template< class T >  
struct is\_copy\_assignable< T, void\_t< copy\_assign\_t<T> > >  
: true\_type { };
- Want is\_move\_assignable? Change T const& → T&&.

## Factoring the idiom

---

- A template template parameter is a useful placeholder:
  - *// primary template handles all types not supporting the operation:*  
`template< class T, template< class > class Op, class = void >  
struct is_valid : false_type { };`
  - *// specialization recognizes/validates only types supporting the op:*  
`template< class T, template< class > class Op >  
struct is_valid<T, Op, void_t<Op<T>>> : true_type { };`
- Now can supply, for example, the earlier `copy_assign_t` or `move_assign_t` alias as the corresponding argument:
  - `template< class T >  
using is_copy_assignable = is_valid<T, copy_assign_t>;`
  - `template< class T >  
using is_move_assignable = is_valid<T, move_assign_t>;`

## The idiom need not depend on void\_t

- E.g., can use enable\_if\_t to obtain void conditionally.
- Example: detect whether a given type is a signed type:
  - // primary template handles non-arithmetic types and unsigned types:  
template< class T, class = void >  
struct is\_signed : false\_type { };
  - // specialization recognizes and validates only signed types:  
template< class T >  
struct is\_signed< T, enable\_if\_t< is\_arithmetic<T>::value  
and T(-1) < T(0)  
>  
> : true\_type { };
  - static\_assert( is\_signed<long>::value );  
static\_assert( not is\_signed<unsigned>::value );



## The idiom need not depend on void either

- However, the idiom requires that some designated type appear in 2 places:
  - As a default argument in the primary template, and also ...
  - As the result of a type function in the specialization's argument.
- Example: detect whether a given type is a complete type:
  - *// primary template handles all incomplete types:*  
template< class T, class = size\_t >  
struct is\_complete : false\_type { };
  - *// specialization recognizes and validates only complete types:*  
template< class T >  
struct is\_complete<T, decltype( sizeof(T) )> : true\_type { };
  - static\_assert( is\_complete<long>::value );  
static\_assert( not is\_complete<void>::value );

## Status of void\_t standardization

- Proposed (document N3911) to C++ Standards Committee.
  - Aimed at <type\_traits> in C++17, but may be published in a Technical Specification before then.
  - On agenda for next Committee meeting.
- Early feedback includes “highly advanced (and elegant)” and “awesome”!
- Trait’s name seems slightly contentious:
  - make\_void\_t, as\_void\_t, to\_void\_t
  - voidify\_t, always\_void
  - enable\_if\_types\_exist\_t, enable\_if\_valid, enable\_if\_exist
  - void\_if\_valid, true\_if\_valid, type\_check

## Summary of techniques and tools

- Metafunction member types and static constexpr data members for intermediate and final metafunction results.
- Metafunction calls (possibly recursive), inheritance, and aliasing to factor commonalities.
- Template specializations (complete and partial) for metafunction argument pattern-matching.
- SFINAE to guide overload resolution and specialization.
- Unevaluated operands, such as function calls to map types.
- Parameter packs as lists (usually of types).
- std:: metafunctions in <type\_traits>, plus such classical metafunctions as iterator\_traits<> and numeric\_limits<>.
- void\_t, the is\_valid idiom, and more!

# Acknowledgements

- Thank you for technical comments:
  - Stephen T. Lavavej
  - Howard Hinnant
  - Eric Niebler
  - Samuel Benzaquen
- Special thanks for making my tie, and for so much more:
  - Carol A. Brown

## A final thought re template metaprogramming

“Although we're professionals now, we all started out as humble students .... Back then, everything was new, and we had no real way of knowing whether what we were looking at was wizardry or WTF.”

— *<http://thedailywtf.com>, 2014-09-09*

# Modern Template Metaprogramming: A Compendium

Revised and expanded edition

# FIN

WALTER E. BROWN, PH.D.

`<webrown.cpp @ gmail.com>`

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