TransformationTrait Alias `void_t`

Document #: WG21 N3911
Date: 2014-02-23
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes a new TransformationTrait alias, `void_t`, for the C++ Standard Library. The trait has previously been described as an implementation detail toward enhanced versions of two other C++11 standard library components. Its value thus proven, `void_t`'s standardization has been requested by several noted C++ library experts, among others.

1 Introduction

We introduced an alias template named `void_t` in each of two recent papers ([N3843] and [N3909]) that were otherwise independent. While very similar in design and intent, the technical details of the two versions of `void_t` differed somewhat from each other in that the latter version had a more general form than did the former. However, each of those papers treated `void_t` as merely an implementation detail en route to a different goal.

After seeing those papers, C++ library experts Stephan T. Lavavej, Howard Hinnant, and Eric Niebler, among several others, independently commented[1] that, even though the alias is extremely simple to implement, they would nonetheless find it useful to have `void_t` as a standard component of the C++ library. This paper therefore proposes to make it so.

We begin with an edited recap of our previous writings on the design, utility, and implementation of `void_t`. We then propose wording for its future incorporation into `<type_traits>`. Finally, the Addendum recapitulates questions raised on the lib-ext reflector regarding the new trait's name.

2 Discussion

2.1 Overview and use case

The purpose of the `void_t` alias template is simply to map any given sequence of types to a single type, namely `void`. Although it seems a trivial transformation, it is nonetheless an exceedingly

[1] For example, STL wrote in private email on 2013-11-19, "In fact, this . . . is so clever that I'd like to see `void_t` proposed for standardization."

useful one, for it makes an arbitrary number of well-formed types into one completely predicable type.

Consider the following example of **`void_t`**'s utility, a trait-like metafunction to determine whether a type **`T`** has a type member named **`type`**:

```
template< class, class = void >
  struct has_type_member : false_type  { };
template< class T >
  struct has_type_member<T, void_t<typename T::type>> : true_type  { };
```

Compared to traditional code that computes such a result, this version seems considerably simpler, and has no special cases (e.g., to avoid forming any pointer-to-reference type). The code features exactly two cases, each straightforward:

a) When there is a type member named **`type`**, the specialization is well-formed (with **`void`** as its second argument) and will be selected, producing a **`true_type`** result;
b) When there is no such type member, SFINAE will apply, the specialization will be nonviable, and the primary template will be selected instead, yielding **`false_type`**.

Each case thus obtains the appropriate result.

As described in our cited papers, we have also applied **`void_t`** in the process of implementing enhanced versions of the C++11 standard library components **`common_type`** and **`iterator_traits`**.

2.2 Implementation/specification

Our preferred implementation (and specification) of **`void_t`** is given by the following near-trivial definition:

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

Given a template argument list consisting of any number[2] of well-formed types, the alias will thus always name **`void`**. However, if even a single template argument is ill-formed, the entire alias will itself be ill-formed. As demonstrated above and in our earlier papers, this becomes usefully detectable, and hence exploitable, in any SFINAE context.

2.3 Implementation workaround

Alas, we have encountered implementation divergence (Clang vs. GCC) while working with the above very simple definition. We (continue to) conjecture that this is because of CWG issue 1558: "The treatment of unused arguments in an alias template specialization is not specified by the current wording of 14.5.7 [temp.alias]."

The notes from the CWG issues list indicate that CWG has all along intended "to treat this case as substitution failure," a direction entirely consistent with our intended uses. Moreover, proposed wording[3] generated and approved during the recent Issaquah meeting follows the indicated direction to resolve the issue, so it seems increasingly likely that we will in the not-too-distant future be able to make portable use of our preferred simpler form.

Until such time, we employ the following workaround to ensure that our template's argument is always used:

```
template< class... >  struct voider  { using type = void; };
template< class... T0toN >  using void_t = typename voider<T0toN...>::type;
```

[2] While we have not yet found a use for the degenerate case of a zero-length template argument list, we also see no reason to forbid it.

[3] There is even a proposed Example that embeds our proposed **`void_t`** specification!

3 Proposed wording[4]

Append to [meta.type.synop] (20.10.2), above paragraph 1, as shown:

```
namespace std {
  ...
  template <class...>
    using void_t = void;
}
```

For the purposes of SG10, we recommend a feature-testing macro named either **`__cpp_lib_void_t`** or **`__cpp_lib_has_void_t`**.

4 Addendum

After a preprint of this paper was made available on the Issaquah wiki, the above-proposed trait's name was questioned. This section will summarize the issues and proposals as recorded on the lib-ext reflector so as to permit a full and fair ~~bikeshed~~ discussion at an appropriate future time.

- "Should **`void_t`** be named something else?
  "It doesn't follow the 'old' use of **`_t`** like **`size_t`** or **`nullptr_t`**. It doesn't quite follow the new use, like **`decay_t`** being **`decay<T>::type`**. ie **`void_t`** is not **`void<T,U,V>::type`**.
  "Should it be named closer to it [*sic*] usage than its implementation? Of course, if it is named based on usage (ie for SFINAE), and is later reused for something else, the name (or new usage) may be seen as 'incorrect'." [Tony Van Eerd, c++std-lib-ext-681].
- "... I have no problem with **`void_t`**. It's not too hard to understand that this is a type transformation from any type to **`void`**." [Ville Voutilainen, c++std-lib-ext-682].
- "... I think **`make_void_t`**, **`as_void_t`**, or **`to_void_t`** would be more descriptive. ..." [Pablo Halperin, c++std-lib-ext-684].
- "By its very nature, the whole thing is confusing.... At the same time, it is very awesome. That's why I wonder about **`check_for_type<>`** or **`sfinae_check<>`** or... [*sic*] something more about its usage. Because without seeing it in context, it is boggling. [Tony Van Eerd, c++std-lib-ext-685].
- "The naive assumption would be **`typedef void void_t;`** but why would you want a typedef for **`void`**?
  "I think it might be a good idea not to lead people into this misconception and the obvious questions that would arise from that." [Bjarne Stroustrup, c++std-lib-ext-686].
- "I [suggest] **`void_type`** as a trait with a nested type, **`void_type_t`** as an alias for that nested type." [Ville Voutilainen, c++std-lib-ext-687].
- "... What about **`enable_if_types_exist_t`**[?] [Pablo Halperin, c++std-lib-ext-688].
- "**`voidify_t`**! :-)" [Pablo Halperin, c++std-lib-ext-690].
- "... **`enable_if_valid`**" [Howard Hinnant, c++std-lib-ext-691].
- "... **`enable_if_exist<>`**." [Jeffrey Yasskin, c++std-lib-ext-692].

[4] All proposed additions and ~~deletions~~ are relative to the post-Chicago Working Draft [N3797]. Editorial notes are displayed against a gray background.

- " These are good ideas . . . , but I'd like to point out that Walter's overall technique is highly advanced (and elegant), and surprising even to experienced template metaprogrammers. I don't think that we need to worry about making the name extremely self-explanatory. Something like **`always_void`** would describe what it does (immediately, not overall), without introducing **`enable_if`**'s connotations (**`enable_if`** takes a bool and an optional type, so what does **`enable_if_valid`** take?).

  "Hmm. How about **`void_if_valid`**? That both says what it returns, and says what it's trying to do." [Stephan T. Lavavej, c++std-lib-ext-693].

- "**`void_if_valid`** would satisfy me, particularly given the lack of the optional type." [Jeffrey Yasskin, c++std-lib-ext-694].

- "Actually, what it returns isn't very important. In fact, I don't want to lose the elegance of it, but it should maybe return **`true_type`**, not **`void`**. More self-documenting. (There is a subtle difference there — **`void`** can't be instantiated, but I don't think that makes a difference any where?)

  "So **`true_if_valid`**?

  "Or just **`type_check<>`**." [Tony Van Eerd, c++std-lib-ext-703].

- "Maybe: **`template<typename T, typename U = void> using enable_if_valid_t = U;`**" [Richard Smith, c++std-lib-ext-708].

  "It needs to be var-arg. **`T...`** [*sic*]" [Tony Van Eerd, c++std-lib-ext-709].

- "I've been using the **`first`** template for a while (the **`::type`** version would be **`first_t`** then). The idea is the same as **`void_t`**, except that the type you get is not **`void`** but the first of the template parameters. Just thought I'd mention this version. On the other hand, we will probably want a **`kth_t`** (**`nth_param_t`**?) to extract the k-th parameter from a pack, which makes **`first_t`** unnecessary but may be a bit overkill for **`void_t`**."

  "Just to expand a bit on the uses of **`first_t`**:

  "1) **`first_t<T...>`** extracts the first type.

  "2) **`first_t<void,...>`** same as **`void_t`**. With partial specializations, and until we get concepts, it is occasionally helpful to use it with something other than void (although in practice I add an extra dummy parameter to classes I intend to partially specialize in complicated ways, so I don't often use **`first_t`** for that).

  "3) **`first_t<T>`** same as **`std::identity<T>::type`**, makes it non-deducible.

  "It is multi-purpose ;-) On the other hand, that makes it less convenient as a vocabulary helper because its name can't reflect all the uses (**`type_checker`**, **`nondeducible_t`**, etc). [Marc Glisse, c++std-lib-ext-697, c++std-lib-ext-717].

Despite the above opinions, it remains our belief that the **`void_t`** name was selected ". . . following a common convention of long standing, namely that **`_t`** often denotes a typedef name, as is the case in **`size_t`** and **`ptrdiff_t`**, for example. By that reasoning, **`void_t`** seems consistent with precedent." [W. Brown, c++std-lib-ext-681].

5 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments.

6 Bibliography

[N3797] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3797 (post-Chicago mailing), 2013-10-13. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

[N3843] Walter E. Brown: "A SFINAE-Friendly **`std::common_type`**." ISO/IEC JTC1/SC22/WG21 document N3843 (pre-Issaquah mailing), 2014-01-01. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3843.pdf.

[N3844] Walter E. Brown: "A SFINAE-Friendly **`std::iterator_traits`**." ISO/IEC JTC1/SC22/WG21 document N3844 (pre-Issaquah mailing), 2014-01-01. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3844.pdf.

[N3909] Walter E. Brown: "A SFINAE-Friendly **`std::iterator_traits`**, v2." ISO/IEC JTC1/SC22/WG21 document N3909 (post-Issaquah mailing), 2014-02-10. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3909.pdf. A revision of [N3844].

7 Document history

Version	Date	Changes
1	2014-02-23	• Published as N3911.

Proposing Standard Library Support for the C++ Detection Idiom

Document #: WG21 N4436
Date: 2015-04-09
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper describes a coding pattern, termed the C++ *detection idiom*, powered by the **`void_t`** metafunction recently accepted into the C++17 standard library. A fully-implemented toolkit of interfaces to this idiom is then presented and proposed for future standardization.

Detection is, or ought to be, an exact science. . . .
— SIR ARTHUR IGNATIUS CONAN DOYLE

1 Introduction

At the 2014 Urbana meeting, WG21 adopted [N3911], thereby adding to the C++17 standard library an alias template named **`void_t`**. Originating as an implementation detail in each of two otherwise-independent earlier papers ([N3843] and [N3909]), it rapidly became clear that this near-trivial **`void_t`** trait made possible a straightforward application of SFINAE in a pattern that we term the C++ *detection idiom*. Further, recent evidence[1] suggests that the use of this idiom provides small but measurable improvements in compilation performance, when compared to traditional approaches.

In experimenting with **`void_t`** in the context of this detection idiom, we have discovered a means of encapsulating and parameterizing the idiom so as to simplify its application. Moreover, we have found that such encapsulation forms the basis of a small toolkit that dramatically simplifies implementation of a large class of metafunctions such as those in the standard library.

[1] "I've run a few tests and using [**`void_t`**] in our **`_GLIBCXX_HAS_NESTED_TYPE`** macro reduces the front-end's memory footprint and compile-time compared to the old implementation. . . " [Jonathan Wakely, **`libstdc++@gcc.gnu.org`**, 2014-11-11].

We begin with a summary of the design, utility, and implementation of **void_t**, then describe the detection idiom. Subsequent sections will describe a fully-implemented toolkit of interfaces to the idiom, and will propose this toolkit for incorporation into a future TS or IS.

2 The void_t alias

The **void_t** alias template behaves as a metafunction that maps any given sequence of types to a single type, namely to **void**. Although a trivial transformation, it has nonetheless proven exceedingly useful, for it makes an arbitrary number of well-formed types into one completely predicable type.

Our preferred implementation (and specification) of **void_t** is the following near-trivial definition:[2]

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

Given a template argument list consisting of any number[3] of well-formed types, the alias will thus always name **void**. However, if even a single template argument is ill-formed, the entire alias will itself be ill-formed.

As demonstrated in our earlier papers, this behavior becomes usefully detectable, and hence exploitable, in any SFINAE context.

3 The detection idiom

As an idiomatic application of **void_t**, we previously presented the following trait-like metafunction that determines whether a type **T** has a type member named **T::type**:

```
// primary template handles types that have no nested ::type member
template< class, class = void_t<> >
struct
  has_type_member : false_type  { };

// specialization recognizes types that do have a nested ::type member
template< class T >
struct
  has_type_member<T, void_t<typename T::type>>
: true_type  { };
```

The code features exactly two cases, each straightforward:

a) When there is a type member named **type**: the specialization is well-formed (albeit with a funny spelling of **void** as its second argument) and will be selected,[4] producing a **true_type** result;

b) When there is no such type member: the specialization will be nonviable (due to SFINAE) and the primary template will be selected instead, yielding **false_type** as the result.

[2] This definition relies on the resolution of CWG issue 1558 ("The treatment of unused arguments in an alias template specialization is not specified by the current wording of 14.5.7 [temp.alias]") adopted at the Urbana meeting. An alternate formulation of **void_t** is available for compilers whose semantics are inconsistent with this resolution; see [N3911].

[3] We have to date stilll not found a use for the degenerate case of a zero-length template argument list. However, we also see no harm in it, especially as forbidding this case would have slightly complicated **void_t**'s design.

[4] See §4 for a discussion of this point.

Each case thus obtains the appropriate result. As we noted in our **void_t** paper, "Compared to traditional code that computes such a result, this version seems considerably simpler, and has no special cases (e.g., to avoid forming any pointer-to-reference type)."

We term this code pattern the C++ *detection idiom* because it is capable of recognizing the validity of essentially any C++ expression. For example, the following transformation of the above code (differing only in name and in the highlighted code) detects whether a type supports a pre-increment operator:

```
1  // primary template handles types that do not support pre-increment
2  template< class, class = void_t<> >
3  struct
4    has_pre_increment_member : false_type  { };

6  // specialization recognizes types that do support pre-increment
7  template< class T >
8  struct
9    has_pre_increment_member<T, void_t<decltype( ++declval<T&>() )>>
10 : true_type  { };
```

Note particulary the role of **std::declval** in forming an archetypal expression to be detected, and the use of **decltype** to inspect this expression in an unevaluated context.

With careful attention to the form of the archetypal expression, it is possible to detect whether an operator is supported via a member function or via a non-member function. Consider the following three expressions:

(a) **&declval<T&>()**,

(b) **declval<T&>().operator&()**, and

(c) **operator&(declval<T&>())**.

When used as the operand to **decltype**, we can detect, respectively,

(a) whether a type supports the address-of operator,

(b) whether a type supports that operator via a member function, and

(c) whether a type supports that operator via a free function.

Such granularity has proven useful. For example, we have been able to ensure that an instance of a type can have its address taken. (Such a requirement is part of the **Semiregular** concept described in [N3351].) Moreover, we can further guarantee via the detection idiom that the type supports the operation without providing any **operator&** overload, thus ensuring that only the built-in operator is available.

4 Validity of the idiom

In a core reflector thread (subject: "Class SFINAE?"), John Spicer commented on the coding technique underlying the detection idiom. He wrote, "This is not overloading of class declarations, it is just partial specialization. The question is whether SFINAE applies . . . in [the] deduction process used in partial specialization. I believe it does in all implementations, and is important functionality" [c++std-core-26537, 2014-12-08].

However, later in that same thread, Richard Smith observed "that we're missing the core wording for template argument deduction for partial specializations. 14.5.5.1/2 says 'go look in 14.8.2', and then 14.8.2 doesn't say what to do; the particular issue here is that the 14.8.2 words that support SFINAE only talk about a function type and its template parameters, but we more generally seem to be missing a subclause of 14.8.2 that describes this form of deduction for

matching partial specializations against a template argument list" [c++std-core-26539, 2014-12-08].

It is our understanding that Smith's observation re missing wording will form the basis of a new CWG issue. Once formally resolved, there should be no doubt as to the idiom's validity. Until then, we rely on Spicer's comment as sufficient validation for our usage.

5 A detection toolkit

Since we first devised **void_t** and recognized the detection idiom, we have been quite extensively experimenting with it. For the most part, we have been reimplementing a large cross-section of the standard library (including, for example, all of headers **<type_traits>**, **<cmath>**, **<iterator>**, **<string>**, **<algorithm>**, and **<random>**). We have observed that the use of the detection idiom has wide-ranging applicability leading to significant reduction in code complexity and attendant increase in code comprehensibility.

Initially, we performed manual transformations of the archetypal expressions in the idiom. This led to significant code duplication, as the rest of the idiom's code (other than the resulting trait's name) is boilerplate. We subsequently discovered a means of encapsulating the detection idiom as a self-contained metafunction that is parameterized on the archetypal expression via a techinique that, in this context, we refer to as a *metafunction callback*.

Our initial version was formulated as follows:

```
// primary template handles all types not supporting the operation:
template< class, template<class> class, class = void_t< > >
struct
  detect : false_type  { };

// specialization recognizes/validates only types supporting the archetype:
template< class T, template<class> class Op >
struct
  detect< T, Op, void_t<Op<T>> > : true_type  { };
```

To use this **detect** metafunction, we supply it with another metafunction (i.e., a meta-callback) that fills the role of the archetypal expression. For example, here is an implementation of the **is_assignable** type trait:

```
// archetypal expression for conversion operation
template< class L, class R >
using
  assign_t = decltype( STD::declval<L>() = STD::declval<R>() )

// trait corresponding to that archetype
template< class L, class R >
using
  is_assignable = detect<void, assign_t, L, R>;
```

Such application of the **detect** metafunction dramatically decreased the amount of boilerplate code to be written in adapting the detection idiom to new circumstances. Although the resulting code was significantly more comprehensible than the original, we disliked the above **detect** interface because the **void** argument in the metafunction call is an implementation detail that shouldn't leak out to client code. Accordingly, we designed a different interface, shown below in §6.2 under the name **is_detected**. In addition, we found use cases for three variations on the basic theme:

1. The first variation is to seek a specified nested type, and yield an alias to that type if it is detected, and to produce an alias to a specified default type if the desired nested type is not detected. This variation is useful in implementing such specifications as "**`Alloc::pointer`** if such a type exists; otherwise, **`value_type*`**" [allocator.traits.types]/1. We name this variant **`detected_or`**.
2. The second variation is to detect an archetype iff it also produced a designated result type. This is useful to ensure that only canonical operations are recognized. For example, the current specification of the **`is_assignable`** trait is silent with respect to the resulting type, although a canonical assignment operator must result in a reference type. We name this variant **`is_detected_exact`**.
3. The third variation is to detect an archetype iff it also produced a result type convertible to a specified type. This is useful in recognizing, for example, relational operations (whose result types must be convertible to **`bool`**). We name this last variant **`is_detected_convertible`**.

It is our experience that these four interfaces to the detection idiom satisfy the overwhelming majority of our applications of the idiom. We will therefore consider these as the components of our detection idiom toolkit. The next section will first present a common infrastructure that supports the entire toolkit, and will then provide a complete implementation of all proposed variations.

6 Implementing the toolkit

6.1 The `detector` infrastructure

We have devised the following **`detector`** template as a common infrastructure to support the four desired components of our detection idiom toolkit: (a) **`is_detected`**, (b) **`detected_or`**, (c) **`is_detected_exact`**, and (d) **`is_detected_convertible`**.

```
// primary template handles all types not supporting the archetypal Op
template< class Default
        , class // always void; supplied externally
        , template<class...> class Op
        , class... Args
        >
struct
  detector
{
  constexpr static auto value = false;
  using type = Default;
};

// specialization recognizes and handles only types supporting Op
template< class Default
        , template<class...> class Op
        , class... Args
        >
struct
  detector<Default, void_t<Op<Args...>>, Op, Args...>
{
  constexpr static auto value = true;
  using type = Op<Args...>;
};
```

Now we can implement each of our four desired interfaces as aliases to this infrastructure.

6.2 The `is_detected` interface

First we have **`is_detected`** and its associates **`is_detected_v`** and **`detected_t`**:[5]

```
template< template<class...> class Op, class... Args >
using
  is_detected = detector<void, void, Op, Args...>;

template< template<class...> class Op, class... Args >
constexpr bool
  is_detected_v = is_detected<Op, Args...>::value;

template< template<class...> class Op, class... Args >
using
  detected_t = typename is_detected<Op, Args...>::type;
```

6.3 The `detected_or` interface

Next we show **`detected_or`** and the associated **`detected_or_t`**:

```
template< class Default, template<class...> class Op, class... Args >
using
  detected_or = detector<Default, void, Op, Args...>;

template< class Default, template<class...> class Op, class... Args >
using
  detected_or_t = typename detected_or<Default, Op, Args...>::type;
```

6.4 The `is_detected_exact` interface

Next are **`is_detected_exact`** and associate **`is_detected_exact_v`**:

```
template< class Expected, template<class...> class Op, class... Args >
using
  is_detected_exact = is_same< Expected, detected_t<Op, Args...> >;

template< class Expected, template<class...> class Op, class... Args >
constexpr bool
  is_detected_exact_v = is_detected_exact< Expected, Op, Args...>::value;
```

6.5 The `is_detected_convertible` interface

Finally, we have **`is_detected_convertible`** and the associated **`is_detected_convertible_v`**:

```
template< class To, template<class...> class Op, class... Args >
using
  is_detected_convertible = is_convertible< detected_t<Op, Args...>, To >;

template< class To, template<class...> class Op, class... Args >
constexpr bool
  is_detected_convertible_v
    = is_detected_convertible<To, Op, Args...>::value
```

6.6 The `nonesuch` utility type

We also recommend the following nearly-useless type, **`nonesuch`**[6]:

[5] But see also §6.6 for a recommended tweak to this definition.

[6] This type was inspired by, and patterned after, the internal type **`__nat`** (which we believe is an acronym for "not a type") found in **`libc++`**.

```
struct
  nonesuch
{
  nonesuch( )                        = delete;
  ~nonesuch( )                       = delete;
  nonesuch( nonesuch const& )        = delete;
  void
    operator = ( nonesuch const& ) = delete;
};
```

Given this type, we have found it expedient to make one small adjustment in our earlier definition of **`detected_or_t`**: We prefer to specify **`nonesuch`** as the (default) result when the provided archetype is not detected. This change avoids the possibility of a spurious result in **`is_detected_exact`** (in the case where the expected result type is **`void`** but the archetypal operation is not detected: we ought not yield **`void`** in such a case).

7 Proposal

Using the above-described interfaces to the detection idiom, we have produced a library of concept-like functions consistent with those described in Section 3 of [N3351]. We therefore believe that the detection toolkit detailed in §6 above is a library solution that is fully compatibile with the semantics of *function concept*s and *variable concept*s as set forth in Clause [dcl.spec.concept] of [N4377].

We respectfully recommend that the Concepts Study Group, the Library Evolution Working Group, and the Evolution Working Group jointly study the relationship of the proposed toolkit to the Concepts Lite proposal [N4377] and come to a unified recommendation as to these proposals' future direction. Until then, **we propose this detection toolkit for WG21 standardization**. Upon LEWG approval of this proposal, we will provide LWG with proposed wording that specifies the described interface and behavior.

8 Acknowledgments

Many thanks, for their thoughtful comments, to the readers of early drafts of this paper.

9 Bibliography

[N3351] B. Stroustrup and A. Sutton (eds.): "A Concept Design for the STL." ISO/IEC JTC1/SC22/WG21 document N3351 (post-Issaquah mailing), 2012-01-13. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf.

[N3843] Walter E. Brown: "A SFINAE-Friendly **`std::common_type`**." ISO/IEC JTC1/SC22/WG21 document N3843 (pre-Issaquah mailing), 2014-01-01. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3843.pdf.

[N3844] Walter E. Brown: "A SFINAE-Friendly **`std::iterator_traits`**." ISO/IEC JTC1/SC22/WG21 document N3844 (pre-Issaquah mailing), 2014-01-01. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3844.pdf.

[N3909] Walter E. Brown: "A SFINAE-Friendly **`std::iterator_traits`**, v2." ISO/IEC JTC1/SC22/WG21 document N3909 (post-Issaquah mailing), 2014-02-10. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3909.pdf. A revision of [N3844].

[N3911] Walter E. Brown: "TransformationTrait Alias **`void_t`**." ISO/IEC JTC1/SC22/WG21 document N3911 (post-Issaquah mailing), 2014-02-23. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3911.pdf.

[N4377] Andrew Sutton: "Programming Languages—C++ Extensions for Concepts." ISO/IEC JTC1/SC22/WG21 document N4377 (mid-Urbana/Lexena mailing), 2015-02-09. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf.

10 Document history

Version	Date	Changes
1	2015-04-09	• Published as N4436.