Proposing Standard Library Support for the C++ Detection Idiom, v2

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 **std::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]

```
1  template< class... >
2  using
3    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**:

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

6  // specialization recognizes types that do have a nested ::type member:
7  template< class T >
8  struct
9    has_type_member<T, void_t<typename T::type>>
10 : 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 any significant 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 <u>highlighted</u> 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 <u>without</u> 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 forms the basis of a new CWG issue, 2054. 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 technique that, in this context, we refer to as a *metafunction callback*.

Our initial version was formulated as follows:

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

6  // specialization recognizes/validates only types supporting the archetype:
7  template< class T, template<class> class Op >
8  struct
9    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:

```
1  // archetypal expression for assignment operation:
2  template< class T >
3  using
4    assign_t = decltype( declval<T&>() = declval<T const &>() )

6  // trait corresponding to that archetype:
7  template< class T >
8  using
9    is_assignable = detect<void, assign_t, T>;
```

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**.

```
1  // primary template handles all types not supporting the archetypal Op:
2  template< class Default
3           , class // always void; supplied externally
4           , template<class...> class Op
5           , class... Args
6           >
7  struct
8    detector
9  {
10   using value_t = false_type;
11   using type = Default;
12 };

14 // the specialization recognizes and handles only types supporting Op:
15 template< class Default
16          , template<class...> class Op
17          , class... Args
18          >
19 struct
20   detector<Default, void_t<Op<Args...>>, Op, Args...>
21 {
22   using value_t = true_type;
23   using type = Op<Args...>;
24 };
```

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]

```
1  template< template<class...> class Op, class... Args >
2  using
3    is_detected = typename detector<void, void, Op, Args...>::value_t;

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

9  template< template<class...> class Op, class... Args >
10 using
11   detected_t = typename detector<void, void, Op, Args...>::type;
```

6.3 The `detected_or` interface

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

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

5  template< class Default, template<class...> class Op, class... Args >
6  using
7    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**:

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

5  template< class Expected, template<class...> class Op, class... Args >
6  constexpr bool
7    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**:

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

5  template< class To, template<class...> class Op, class... Args >
6  constexpr bool
7    is_detected_convertible_v
8      = 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++**.

```
1  struct
2    nonesuch
3  {
4    nonesuch( )                        = delete;
5    ~nonesuch( )                       = delete;
6    nonesuch( nonesuch const& )        = delete;
7    void
8      operator = ( nonesuch const& ) = delete;
9  };
```

Given this type, we have found it expedient to make one small adjustment in our earlier definition of **is_detected** (and, for consistency, in **detected_t** as well): We prefer to specify **nonesuch** as the (default) result when the provided archetype is not detected. These changes, shown below, avoid 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).

```
1  template< template<class...> class Op, class... Args >
2  using
3    is_detected = typename detector<nonesuch, void, Op, Args...>::value_t;

5  template< template<class...> class Op, class... Args >
6  using
7    detected_t = typename detector<nonesuch, void, Op, Args...>::type;
```

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**.

8 Examples

Here are two versions of a trait to detect copy-assignability. The first version is a direct analog of the standard library trait, while the second additionally insists on the canonical reference result type:

```
1  // archetypal alias for a copy assignment operation:
2  template< class T >
3  using
4    copy_assign_t = decltype( declval<T&>() = declval<T const &>() );

6  template< class T >
7  using
8    is_copy_assignable = is_detected< T, copy_assign_t >;

10 template< class T >
```

```
11  using
12    is_canonical_copy_assignable = is_detected_exact< T&, T, copy_assign_t >;
```

The following example illustrates the idiom's applicability to discriminate a member function from a free function:

```
1  // archetypal alias for the result type of a valid member operator &:
2  template< class T >
3  using
4    addrof_mbr_t = decltype( declval<T const&>() . operator &() );

6  template< class T >
7  using
8    has_addressof = is_detected< T, addrof_mbr_t >;
```

The last example demonstrates a possible implementation of a part of **pointer_traits**, namely that the **difference_type** member be an alias for **ptrdiff_t** unless the given type has its own **difference_type** member:

```
1  // archetypal alias for a valid type member:
2  template< class T >
3  using
4    diff_t = typename T::difference_type;

6  template< class Ptr >
7  using
8    difference_type = detected_or_t< ptrdiff_t, diff_t, Ptr >;
```

9   Proposed wording[7]

9.1   Synopsis

Append the following specifications to [meta.type.synop] (20.10.2), above paragraph 1. (Note that, although WG21 adopted **void_t** for the C++17 working paper, a TS may not by ISO rules refer to a feature in a document that it has not yet officially published. As it is an integral part of the detection idiom being proposed here, we therefore specify **void_t** here, in the **experimental** namespace, as well.)

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
  ...
  // 20.10.8, detection idiom:
  template <class...> using void_t = void;

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

---

[7]All proposed additions (there are no proposed deletions) are relative to the pre-Lenexa Working Draft [N4431]. Editorial notes are displayed against a   gray   background.

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

  template <template<class...> class Op, class... Args>
  using is_detected = see below;

  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 = see below;

  template <class Default, template<class...> class Op, class... Args>
  using detected_or = see below;

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

  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;

  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;

} } }  // namespaces
```

9.2 New subclause

Insert the following as a new subclause after, and at the level of, [meta.trans].

20.10.8 Detection idiom   [meta.detect]

```
template <class Default, class AlwaysVoid,
          template<class...> class Op, class... Args>
struct DETECTOR {  // exposition only
  using value_t = false_type;
  using type = Default;
};

template <class Default, template<class...> class Op, class... Args>
struct DETECTOR<Default, void_t<Op<Args...>>, Op, Args...> { // exposition only
  using value_t = true_type;
  using type = Op<Args...>;
};
```

```
template <template<class...> class Op, class... Args>
using is_detected = typename DETECTOR<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename DETECTOR<nonesuch, void, Op, Args...>::type;

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

1 [*Example:*

```
// archetypal helper alias for a copy assignment operation:
template <class T>
using copy_assign_t = decltype(declval<T&>() = declval<T const &>());

// plausible implementation for the is_assignable type trait:
template <class T>
using is_copy_assignable = is_detected<T, copy_assign_t>;

// plausible implementation for an augmented is_assignable type trait
// that also checks the return type:
template <class T>
using is_canonical_copy_assignable = is_detected_exact<T&, T, copy_assign_t>;
```

— *end example*]

2 [*Example:*

```
// archetypal helper alias for a particular type member:
template <class T>
using diff_t = typename T::difference_type;

// alias the type member, if it exists, otherwise alias ptrdiff_t:
template <class Ptr>
using difference_type = detected_or_t<ptrdiff_t, diff_t, Ptr>;
```

— *end example*]

9.3 Feature-testing

For the purposes of SG10, we recommend a feature-testing macro named `__cpp_lib_experimental_detect`.

10 Acknowledgments

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

11 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/Lenexa mailing), 2015-02-09. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf.

[N4431] Richard Smith: "Working Draft, Standard for Programming Language C++" ISO/IEC JTC1/SC22/WG21 document N4431 (pre-Lenexa mailing), 2015-04-10. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4431.pdf.

12   Document history

Version	Date	Changes
1	2015-04-09	• Published as N4436.
2	2015-05-03	• Fixed a few small typos and thinkos. • Referred to new CWG issue by number. • Improved toolkit's presentation. • Added §8 (examples) and §9 (proposed wording) sections. • Published as N4502.