Proposing Standard Library Support for the C++ Detection Idiom

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\footnote{\`{I’ve run a few tests and using [void_t] in our \texttt{GLIBCXX\_HAS\_NESTED\_TYPE} macro reduces the front-end’s memory footprint and compile-time compared to the old implementation…” [Jonathan Wakely, \texttt{libstdc++@gcc.gnu.org}, 2014-11-11].} 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.
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:

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

Given a template argument list consisting of any number 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:

```cpp
// 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, 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 still 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:

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

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

Note particularly 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 technique that, in this context, we refer to as a metafunction callback.

Our initial version was formulated as follows:

```cpp
// 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:

```cpp
// 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.

# Implementing the toolkit

## 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`.

```cpp
// 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:\(^5\)

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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 compatible with the semantics of `function concepts` and `variable concepts` 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


10 Document history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2015-04-09</td>
<td>• Published as N4436.</td>
</tr>
</tbody>
</table>