TransformationTrait Alias void_t

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`:

```cpp
template< class, class = void >
struct has_type_member : false_type {};  // 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;

template< class T >
struct has_type_member<T, void_t<typename T::type>> : true_type {};  // 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.
```

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:

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:

```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 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 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:

```cpp
template< class... > struct voider { using type = void; };  // 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;

template< class... T0toN > using void_t = typename voider<T0toN...>::type;  // 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.
```

2While 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.

3There is even a proposed Example that embeds our proposed `void_t` specification!
3 Proposed wording

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

```cpp
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, i.e., void_t is not void<T,U,V>::type.
  “Should it be named closer to its [sic] usage than its implementation? Of course, if it is named based on usage (i.e., 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 sfnae_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_exists<>.” [Jeffrey Yasskin, c++std-lib-ext-692].

---

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


7 Document history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2014-02-23</td>
<td>• Published as N3911.</td>
</tr>
</tbody>
</table>