

The identity metafunction

Timur Doumler (papers@timur.audio)

Document #: P0887R1
Date: 2018-03-18
Project: Programming Language C++
Audience: Library Working Group

Abstract

This paper proposes the library utility `std::type_identity`, which implements the identity metafunction. The functionality it provides is both fundamental and surprisingly useful. It can be used to selectively disable template argument deduction (for which there is currently no standard mechanism), and as a basic building block for other metafunctions. The main issue with standardising this utility has long been the lack of consensus on a name. This paper summarises the history, discusses different names, and provides a rationale for the chosen name.

1 Definition

This paper proposes to add the most fundamental metafunction to the C++ standard library: the identity metafunction. It takes one template argument, type `T`, and provides the member typedef `type` which is the same type `T`.

The implementation is trivial:

```
template <typename T>
struct type_identity { using type = T; };

template <typename T>
using type_identity_t = typename type_identity<T>::type;
```

Despite its simplicity, this metafunction is surprisingly useful in various situations.

2 Motivation

2.1 Disabling template argument deduction

Commonly, the need arises to disable template argument deduction for one or more arguments of a function, and force client code to explicitly specify the template parameter. Consider:

```
template <typename T>
void foo(T t) { /* ... */ }
```

If we wish to disable deduction on `t`, it can be accomplished as follows:

```
template <typename T>
void foo(type_identity_t<T> t) { /* ... */ }
```

In fact, this technique has become a common idiom (let’s call it the “type identity idiom”). Currently, the idiom requires one to type out their own identity metafunction, defined as above. We should instead provide it in the standard library. This would standardise existing practice and make the idiom more convenient to use and easier to teach (somewhat similar to how `enable_if` is a standard metafunction for selectively removing functions from overload resolution).

2.2 Disabling deduction guides

C++17 introduced class template argument deduction, creating a new use case for the identity metafunction. Often, the need arises to explicitly disable deduction guides to prevent unsafe or unexpectedly behaving code. For example, consider a library smart pointer class that takes a raw pointer to an object as its constructor argument, and then somehow manages the object (for a real-world example, see [JuceScopedPointer](#)):

```
template <class T>
class smart_pointer
{
public:
    smart_pointer(T* object);
    // Other public methods...
}
```

If the users of this library switch to C++17, this becomes well-formed:

```
Widget* widget{/* ... */};
smart_pointer ptr{widget}; // using implicit deduction guide
```

This is inherently dangerous as the constructor of `smart_pointer` cannot differentiate between a pointer to an object and an array. In cases like this, the automatic deduction guide for the offending constructor is usually disabled. This forces the client code to explicitly specify the template parameter. The “type identity idiom” is the easiest way to achieve this:

```
smart_pointer(type_identity_t<T>* object);
```

2.3 Fundamental metafunction building block

There are two ubiquitous idioms for type traits:

- define a public data member `value` with a given value
- define a public member typedef `type` that names a given type

It is surprising that there is a standard utility providing the former (`std::integral_constant`), but no standard utility providing the latter.

`type_identity` is this utility. It is a fundamental building block that other metafunctions can simply inherit from. For example, `remove_const` could be implemented as follows:

```
template <typename T>
struct remove_const : type_identity<T> {};

template <typename T>
struct remove_const<T const> : type_identity<T> {};
```

Several other examples are given by Walter E. Brown in his talk “Modern Template Metaprogramming: A Compendium” [\[Brown2014\]](#).

3 Alternative approaches

For disabling deduction guides, an alternative to `type_identity` is to add a new core language feature. This would require new syntax, such as `using = delete` on deduction guides. This was proposed in [P0091] in Kona 2017 but not adopted. Such a facility would be more limited than `type_identity`: it could not be used to selectively disable single arguments, and would not work for template argument deduction on functions. A core language change is also more difficult to justify if the same effect¹ can be accomplished with a simple library utility that already follows an established idiom.

Another alternative is repurposing other standard metafunctions to do the job of the identity metafunction. However the resulting code ends up:

- redundant, for example `enable_if_t<true, T>`,
- confusing, for example `remove_reference_t<T>` can often be used, but then ends up in contexts where `T` could not even be a reference in the first place,
- not providing the same functionality, for example `common_type_t<T>` is not equivalent to `type_identity_t<T>` because it decays `T`.

The standard should give us a way to write what we actually mean. See [N3766] for further discussion and real-world examples.

4 Historical context

The fact that the identity metafunction is not yet standardised has historical reasons.

Pre-C++98 implementations of the standard library had an entity named `std::identity`. This was different from the utility discussed here: it was a function object for value identity and defined an `operator()`. This original version of `std::identity` still survives today as a legacy SGI extension to the C++ standard library and in an extension namespace of `stdlibc++`.

Later, a metafunction `std::identity` with an implementation identical to the one discussed here was proposed in [N1856] in 2005 and merged into the C++0x working draft [N2284] in 2007. The motivation at the time was to simplify usage of `std::forward`, via the same mechanism as discussed above: it forces client code to explicitly specify the template parameter.

This caused a defect report [700] because of the name clash with the older SGI extension. The defect was resolved by adding `operator()` to align both definitions of `std::identity`. However this addition caused further issues and led to two more defect reports [823][939]. Ultimately, `std::forward` evolved and no longer needed `std::identity` to work correctly. The defects were resolved by completely removing `std::identity` from the working draft.

The wish to have this fundamental metafunction in the standard library persisted, and so `identity` was proposed again in [N3766] in 2013. The two (mutually exclusive) options proposed in that paper were: either fixing the still unresolved issues with `operator()`, or removing `operator()` and changing the name to `identity_of`. Neither of the options received consensus in LEWG; there was no further work on these issues.

The prevalent opinion has always been that the identity metafunction is useful and should be standardised, and that the only hurdle is the current lack of consensus on a name for it. Overcoming this hurdle is exactly the goal of this proposal.

¹It is somewhat unclear whether such a language feature should remove deduction guides from deduction, as `type_identity` does, or instead make the program ill-formed if that deduction guide is selected by deduction (similar to existing `using = delete` for function bodies), although for most real-world use cases this distinction would probably not matter.

5 Choosing a name

5.1 Names suggested so far

Since [N3766], various alternative names for the identity metafunction were suggested. Below is a summary of all serious naming suggestions we are currently aware of, along with their drawbacks.

- `identity` — The original name. It is currently unclear whether the name clash with the old SGI version of `identity` would still be a problem for a relevant number of people. However, the bigger problem with the word “identity” is its ambiguity: it could stand either for the type identity metafunction proposed here, or for the value identity function object (notably, the Ranges TS [N4685] currently uses `std::ranges::identity` for the latter), or even for a combination of both (which was proposed in [N3766] but failed to get LEWG consensus).
- `identity_of` — This was an alternative proposal in [N3766] to circumvent the name clash with the old SGI version of `identity`. However the name does nothing to resolve the inherent ambiguity of `identity` and failed to get LEWG consensus.
- `type` — A simple and clear alternative. Unfortunately it has two problems. First, it is a very popular and generic identifier and could create unforeseen name clashes if added to namespace `std`. Second, it is not possible in C++ to define a member typedef that has the same name as the class itself. Therefore, `type<T>::type` can only be implemented with a workaround via a helper struct (Boost.Hana does this; see 5.2). Alternatively, one could omit the member `type` form altogether and only use the `_t` form. Both solutions would no longer satisfy the concept of *UnaryTypeTrait* and thus create a new inconsistency in header `type_traits`.
- `type_identity` — Arguably the most logical and precise name from the English language perspective, resolving the ambiguity of “identity”. [Boost.TypeTraits] uses this name. The downsides of this name are that some consider it too long for such a basic utility, and that there is some unwanted similarity with the `typeid` operator.
- `identity_type` — A variation of `type_identity`, equally long and perhaps less optimal from the English language perspective.
- `meta_identity` — Yet another attempt to resolve the ambiguity of “identity”. However the prefix `meta_` feels out of place in this context, because the other type traits do not have such a prefix despite also being metafunctions.
- `id` — Abbreviating the word “identity” does not fix the problem of its ambiguity. Also, this causes a name clash with the Objective-C keyword `id`, creating problems for codebases that mix both languages.
- `type_is` — Walter E. Brown’s choice [Brown2014]. However some people consider code like `type_is<T>` to be too “cute” for a standard library utility.
- `type_of` — This is clearly wrong. The “type of a type” is not the same as “the type that is this type”. A type of a type would be a higher-order thing (which has no obvious and natural corresponding C++ language construct).
- `this_type` — This could be misinterpreted as “type of `this`” (the `this` pointer).
- `same_type` — This name sounds too much like a query and would be better fitting for a variadic metafunction that takes several types and checks whether they are the same, similarly to existing `std::common_type`.
- `same` — This is too similar to existing `std::is_same`.

- `omit_from_deduction`, `no_deduce`, etc. — Such names are great when used in the context of disabling argument deduction, but preclude other possible use cases of the identity metafunction.
- `wrapper_type`, `nested_type`, `type_alias`, `type_wrapper`, etc. — Such names attempt to describe the implementation of the identity metafunction rather than its meaning, and are not clear enough.
- `type_constant` — This name emphasises the relation to `integral_constant`. Unfortunately, it seems to suggest that the type could somehow be mutable.

The discussion and plethora of suggested names makes it clear that there is no ideal name that fixes all problems. The task at hand is to decide on the name that is the least bad one.

5.2 Names used in existing libraries

It is illuminating to look at existing practice: what names do popular third-party metaprogramming libraries use for the identity metafunction? Below is an incomplete list.

- [Boost.MPL] has `boost::mpl::identity`. The definition matches the one in section 1.
- [Boost.TypeTraits] has `boost::type_identity`. The definition matches the one in section 1.
- [Boost.Hana] has `boost::hana::type`. To avoid the name shadowing issue with its member typedef also called `type`, it employs the following workaround:

```
template <typename T>
struct __type { using type = T; };

template <typename T>
using type = __type<T>;

template <typename T>
using type_t = typename type<T>::type;
```

- [Boost] further has `boost::type` in header `boost/type.hpp`. It avoids the name shadowing issue by not having a member typedef `type`.
- [Brigand] has `brigand::identity`. The definition matches the one in section 1.
- [Meta] does not have the identity metafunction as such, however it does have a utility typedef for the `T::type` idiom, named `_t`, which is used all over the code, demonstrating its usefulness:

```
template <typename T>
using _t = typename T::type;
```

- [Erasure] has both a typedef `_t` like Meta and a struct `type_` defined as in section 1. The trailing underscore avoids the name shadowing issue.
- [Mp11] has `mp_identity` (`mp_` is the common prefix in this library). The definition matches the one in section 1.

To summarise, as far as we are aware, all popular implementations of the identity metafunction use as a name either `identity`, `type`, or `type_identity`.

5.3 LEWG consensus in Jacksonville

At the 2018 Jacksonville meeting, LEWG reviewed R0 of the present paper and discussed all of the above considerations for naming the identity metafunction. As a result, the name `type_identity` was approved by unanimous consent.

This name is the most logical and unambiguous one. It is unfortunately somewhat long for such a basic utility, but apart from that, `type_identity` does not create any technical, denotative, or connotative issues (unlike all of the other suggestions).

We recognise that the naming of existing metafunctions in the standard is not very consistent in general, and proposals for a more consistent naming policy would be welcome. Going even further, it would be very interesting to consider adopting a new, more modern metaprogramming library for the standard. [P0949] is a recent proposal in this direction. Nevertheless, `type_identity` is useful today, and these directions should not delay its addition to the currently existing type traits.

6 Proposed wording

The proposed changes are relative to the C++ working paper [Smith2018].

In 23.15.2 Header `<type_traits>` synopsis [meta.type.synop], add:

```
// 23.15.7.6, other transformations
template<class T> struct type_identity;

template<class T>
    using type_identity_t = typename type_identity<T>::type;
```

In 23.15.7.6 Other transformations [meta.trans.other], add to Table 50 — Other transformations:

<pre>template<class T> struct type_identity;</pre>	<u>The member typedef <code>type</code> names the type <code>T</code>.</u>
--	--

Document history

- **R0**, 2018-02-12: initial version, proposing wording for `identity` and `type` as mutually exclusive alternatives.
- **R1**, 2018-03-18: Added wording for `type_identity`; removed wording for `identity` and `type`; updated text to reflect the LEWG result in Jacksonville; fixed incorrect description of Boost.Hana's `type`.

Acknowledgements

Many thanks to Michael Spertus, Richard Smith, Zhihao Yuan, John Bytheway, Andrey Davydov, Graham Haynes, Gašper Ažman, Simon Brand, Jonathan Wakely, Jon Chesterfield, Thomas Köppe, and Louis Dionne for their very helpful comments and suggestions on earlier versions of this paper. Many thanks to Titus Winters, Walter E. Brown, and all of LEWG for the very productive discussion in Jacksonville.

References

- [700] P.J. Plauger. Defect Report 700. N1856 defines struct `identity`. <http://cplusplus.github.io/LWG/lwg-defects.html#700>, 2007 (accessed 2018-03-18).

- [823] Walter E. Brown. Defect Report 823. `identity<void>` seems broken. <http://cplusplus.github.io/LWG/lwg-defects.html#823>, 2008 (accessed 2018-03-18).
- [939] Alisdair Meredith. Defect Report 939. Problem with `std::identity` and `reference-to-temporaries`. <http://cplusplus.github.io/LWG/lwg-defects.html#939>, 2008 (accessed 2018-03-18).
- [Boost] Boost. Type Documentation. http://www.boost.org/doc/libs/1_66_0/boost/type.hpp, 2001 (accessed 2018-03-18).
- [Boost.Hana] Boost. Hana Documentation. http://www.boost.org/doc/libs/1_63_0/libs/hana/doc/html/structboost_1_1hana_1_1type.html, 2007 (accessed 2018-03-18).
- [Boost.MPL] Boost. MPL Documentation. http://www.boost.org/doc/libs/1_48_0/libs/mpl/doc/refmanual/identity.html, 2009 (accessed 2018-03-18).
- [Boost.TypeTraits] Boost. Type Traits Documentation. http://www.boost.org/doc/libs/1_66_0/libs/type_traits/doc/html/index.html, 2011 (accessed 2018-03-18).
- [Brigand] Edouard Alligand. Brigand Meta-programming library. <https://github.com/edouarda/brigand>, 2017 (accessed 2018-03-18).
- [Brown2014] Walter E. Brown. Modern Template Metaprogramming: A Compendium. <https://www.youtube.com/watch?v=Am2is2QCvxY>, 2014 (accessed 2018-03-18).
- [Erasure] Gašper Ažman. `liberasure`: A no-dependencies C++ extensible type erasure library. <https://github.com/atomgalaxy/liberasure/>, 2017 (accessed 2018-03-18).
- [JuceScopedPointer] JUCE API Reference. Class `ScopedPointer`. <https://juce.com/doc/classScopedPointer>, 2017 (accessed 2018-03-18).
- [Meta] Eric Niebler. `Meta`: A tiny metaprogramming library. <https://github.com/ericniebler/meta>, 2017 (accessed 2018-03-18).
- [Mp11] Peter Dimov. `Mp11`, a C++11 metaprogramming library. <https://github.com/boostorg/mp11>, 2018 (accessed 2018-03-18).
- [N1856] Howard E. Hinnant. Rvalue Reference Recommendations for Chapter 20. <https://wg21.link/n1856>, 2005 (accessed 2018-03-18).
- [N2284] Pete Becker. Working Draft, Standard for Programming Language C++. <https://wg21.link/n2284>, 2007 (accessed 2018-03-18).
- [N3766] Jeffrey Yasskin. The `identity` type transformation. <https://wg21.link/n3766>, 2013 (accessed 2018-03-18).
- [N4685] Eric Niebler. Working Draft, C++ Extensions for Ranges. <https://wg21.link/n4685>, 2017 (accessed 2018-03-18).
- [P0091] Mike Spertus, Faisal Vali, and Richard Smith. Template argument deduction for class templates (Rev. 7): Deleted deduction guides. <https://wg21.link/p0091>, 2017 (accessed 2018-03-18).
- [P0949] Peter Dimov. Adding support for type-based metaprogramming to the standard library. <https://wg21.link/p0949>, 2018 (accessed 2018-03-18).
- [Smith2018] Richard Smith. Working Draft, Standard for Programming Language C++. <https://github.com/cplusplus/draft>, 2018 (accessed 2018-03-18).