The identity metafunction

Timur Doumler (papers@timur.audio)

Document #: P0887R0
Date: 2018-02-12
Project: Programming Language C++
Audience: Library Evolution Working Group, Library Working Group

Abstract

This paper proposes a library utility 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 is the current lack of consensus on a name. This paper summarises the history and naming suggestions so far, and proposes two mutually exclusive alternatives: `std::identity` and `std::type`.

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 identity { using type = T; };

template <typename T>
using identity_t = typename 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(identity_t<T> t) {/* ... */}
```

In fact, this technique has become a common idiom (let's call it the "identity type 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 "identity type idiom" is the easiest way to achieve this:

```
smart_pointer(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.

`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 : identity<T> {};

template <typename T>
struct remove_const<T const> : 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 `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 `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 effect[1] 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 `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 unary function object and defined an `operator()`. This original version of `std::identity` still survives today as an SGI extension to the C++ standard library (see [SGIDoc]) 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.

Since N3766, there was no updated proposal. The prevalent opinion seems to be that the identity metafunction is useful and should be standardised — without the `operator()` — 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.

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

5 Bikeshedding the name

5.1 Names suggested so far

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

— `identity` — The original name. It is currently unclear whether this name is viable or whether the name clash with the old SGI version of `identity` would still be a problem for a relevant number of people. Also, the word "identity" is somewhat ambiguous. For example, in Boost, `boost::mpl::identity` implements the exact same metafunction proposed here, but in the Ranges TS [N4685], `ranges::identity` names a unary function object similar to the old SGI version, and [N3766] attempted to combine both.

— `identity_of` – This was an alternative proposal in [N3766] to circumvent the name clash with the old SGI version of `identity`. However the name is still too similar to `identity` and was rejected by LEWG.

— `type` — A simple and clear alternative. Unfortunately, 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 (see 6.2).

— `type_identity` or `identity_type` — Attempts to remove the ambiguity of just "identity", but one could argue that those names are too long. [Boost.TypeTraits] uses `type_identity`.

— `id` — Abbreviating the word "identity" does not fix the problem of its ambiguity. Additionally, the abbreviation `id` causes a name clash with the Objective-C keyword.

— `type_is` — Walter E. Brown's choice [Brown2014]. The problem with this name is that `type_is<T>` could be considered 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`, 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`, defined exactly as in section 1.

— [Boost.TypeTraits] has `boost::type_identity`, defined exactly as in section 1.

— [Boost.Hana] has `boost::hana::type`, but it is does not have a member typedef `type` (it is used somewhat differently) and therefore avoids the name shadowing issue.

— [Boost] further has a struct `boost::type` in header `boost/type.hpp`, but it also does not have a member typedef `type`.

— [Brigand] has a struct `brigand::identity`, defined exactly as 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 exactly as in section 1. The trailing underscore avoids the name shadowing issue.

— [Mp11] has `mp_identity`, defined exactly as in section 1 (`mp_` is the common prefix in this library).

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

6 This proposal

Based on the discussion above, we believe that the only viable names for a standardised identity metafunction are `identity` or `type`. This paper proposes those two names as two mutually exclusive alternatives.

6.1 Alternative 1: `identity`

Despite the potential ambiguity of the name `identity`, it is the canonical name for the identity metafunction. We therefore believe it is the best possible choice. Before going forward with this name, two questions need to be answered:

— Is the name clash with the old SGI version of `identity` still a problem for a relevant number of people?

— Are there any plans to standardise `identity` as either the unary function, or as a combination of the unary function and the identity metafunction as proposed in [N3766] after fixing the issues with its `operator()`?

The author is hoping for guidance from LEWG on these two questions. If the answers to both questions are “no”, we see no reason to not adopt `identity` as the identity metafunction, as defined in section 1.

6.2 Alternative 2: `type`

In case it turns out that `identity` is not an option, we propose `type` as a viable alternative. It is a simple and clear name that does not create any clashes or ambiguities. It results in clean client code. The only drawback of `type` is a technicality: it is impossible to define a struct `type` with a member typedef `type` like in section 1 because of name shadowing.

There is a straightforward 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;
```

where `__type` is a struct defined for exposition only.

One could argue that the necessity of such a workaround makes the name `type` a bad option, and that we should simply pick another name to avoid it — seemingly going back to the beginning of the discussion. However we have looked at all other possible names (see discussion above), and believe that the issues created by those are even worse. In case we cannot have `identity`, we believe that the name `type` works best, despite the necessary workaround.

7 Proposed wording

The proposed changes are relative to the C++ working paper [Smith2017].

7.1 Alternative 1: `identity`

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

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

template<class T>
  using identity_t = typename identity<T>::type;
```

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

`template<class T>` `struct identity;`	The member typedef `type` names the type `T`.

7.2 Alternative 2: `type`

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

```
// 23.15.7.6, other transformations
template <class T>
  using type = unspecified;  // see below

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

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

`template<class T>` `using type =` *`unspecified`*`;`	For any type `T`, `type<T>` names a class with a publicly accessible member typedef `type` equal to `T`.

Acknowledgements

Many thanks to Michael Spertus, Richard Smith, Zhihao Yuan, John Bytheway, Andrey Davydov, Graham Haynes, Gašper Ažman, Simon Brand, Jonathan Wakely, Jon Chesterfield, and Thomas Köppe for their very helpful comments and suggestions.

References

[700] P.J. Plauger. Defect Report 700. N1856 defines struct identity. http://cplusplus.github.io/LWG/lwg-defects.html#700, 2007 (accessed 2017-12-07).

[823] Walter E. Brown. Defect Report 823. identity<void> seems broken. http://cplusplus.github.io/LWG/lwg-defects.html#823, 2008 (accessed 2017-12-07).

[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 2017-12-07).

[Boost] Boost. Type Documentation. http://www.boost.org/doc/libs/1_66_0/boost/type.hpp, 2001 (accessed 2018-01-02).

[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 2018-01-02).

[Boost.MPL] Boost. MPL Documentation. http://www.boost.org/doc/libs/1_48_0/libs/mpl/doc/refmanual/identity.html, 2009 (accessed 2018-01-02).

[Boost.TypeTraits] Boost. Type Traits Documentation. http://www.boost.org/doc/libs/1_66_0/libs/type_traits/doc/html/index.html, 2011 (accessed 2018-01-02).

[Brigand] Edouard Alligand. Brigand Meta-programming library. https://github.com/edouarda/brigand, 2017 (accessed 2018-01-02).

[Brown2014] Walter E. Brown. Modern Template Metaprogramming: A Compendium. https://www.youtube.com/watch?v=Am2is2QCvxY, 2014 (accessed 2017-12-07).

[Erasure] Gašper Ažman. liberasure: A no-dependencies C++ extensible type erasure library. https://github.com/atomgalaxy/liberasure/, 2017 (accessed 2018-01-02).

[JuceScopedPointer] JUCE API Reference. Class ScopedPointer. https://juce.com/doc/classScopedPointer, 2017 (accessed 2017-12-07).

[Meta] Eric Niebler. Meta: A tiny metaprogramming library. https://github.com/ericniebler/meta, 2017 (accessed 2018-01-02).

[Mp11] Peter Dimov. Mp11, a C++11 metaprogramming library. https://github.com/boostorg/mp11, 2017 (accessed 2018-01-07).

[N1856] Howard E. Hinnant. Rvalue Reference Recommendations for Chapter 20. https://wg21.link/n1856, 2005 (accessed 2017-12-07).

[N2284] Pete Becker. Working Draft, Standard for Programming Language C++. https://wg21.link/n2284, 2007 (accessed 2017-12-07).

[N3766] Jeffrey Yasskin. The identity type transformation. https://wg21.link/n3766, 2013 (accessed 2017-12-07).

[N4685] Eric Niebler. Working Draft, C++ Extensions for Ranges. https://wg21.link/n4685, 2017 (accessed 2017-12-07).

[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 2017-12-07).

[SGIDoc] SGI STL Documentation. identity<T>. http://www.sgi.com/tech/stl/identity.html, 2017 (accessed 2017-12-07).

[Smith2017] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2017 (accessed 2017-12-07).