# TransformationTraits Redux, v2


*Document #:* WG21 N3655
*Date:* 2013-04-18
*Revises:* N3546
*Project:* JTC1.22.32 Programming Language C++


*Reply to:* Walter E. Brown <webrown.cpp@gmail.com>


## Contents




**Abstract**

This paper proposes to augment C++11's *TransformationTraits* with a number of template aliases whose use dramatically simplifies the traits' most common applications.


## 1 Background

We find the definition of a *TransformationTrait* in [meta.rqmts]/3 of [DuT12]:

> A *TransformationTrait* modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a nested type[1] named **type**, which shall be a synonym for the modified type.

This definition follows a long-standing design and protocol that [AG05, §2.2] terms a *metafunction*; the nested type **type** is an example of *metadata*.

A number of *TransformationTraits* (also known as *modifications*) are specified in subclauses of [meta.trans]:

- six are subclassified as const-volatile modifications (e.g., **add_const**),
- three as reference modifications (e.g., **remove_reference**),
- two as sign modifications (**make_signed** and **make_unsigned**),
- two as array modifications (**remove_extent** and **remove_all_extents**),
- two as pointer modifications (**add_pointer** and **remove_pointer**), and
- eight as other transformations (e.g., **enable_if)**.

It seems obvious that these traits can be composed by passing the metadata of one as the argument to another. Somewhat less obvious, perhaps, is the equally useful capability of passing

[1] Note that the Working Paper's definition lacks the requirement that the nested type be publicly accessible. The Proposed Wording below will remedy this oversight as a drive-by fix.

a metafunction itself as an argument to another metafunction. It is a strength of the design that both forms of composition are available to programmers.

## 2 Proposal

Unfortunately, the above-described flexibility comes with a cost for the most common use cases. In a template context, C++ requires that each “metacall” to a metafunction bear syntactic overhead in the form of an introductory **`typename`** keyword, as well as the suffixed `::`**`type`**:

```
typename metafunction-name<metafunction-argument(s)>::type
```

Even relatively straightforward compositions can rather quickly become somewhat messy; deeper nesting is downright unwieldy:

```
template< class T >  using reference_t
  = typename conditional<is_reference<T>::value, T,
                         typename add_lvalue_reference<T>::type>::type;
```

Worse, accidentally omitting the keyword can lead to diagnostics that are arcane to programmers who are inexpert in metaprogramming details.

In our experience, passing metafunctions (rather than metadata) constitutes a relatively small fraction of metafunction compositions. We find ourselves passing metafunction results far more frequently. We therefore **propose to add a set of template aliases for the library’s *TransformationTraits*** in order to reduce the programmer burden of expressing this far more common case. Note, in the following rewrite of the above example, the absence of any **`typename`** keyword, as well as the absence of any `::`**`type`** suffix, thus condensing the statement from 3 to 2 lines of code:

```
template< class T >  using reference_t
  = conditional_t< is_reference<T>::value, T, add_lvalue_reference_t<T> >;
```

As shown in the proposed wording below, we recommend that aliases be named according to a consistent pattern, namely the name of the aliased trait suffixed by **`_t`**, the conventional suffix denoting a type alias. Thus, for example, the alias for **`add_cv<T>::type`** would be **`add_cv_t`**.

## 3 Proposed wording

Modify [meta.rqmts]/3 of [DuT12] as follows:

A *TransformationTrait* . . . shall define a publicly accessible nested type named **`type`**, which . . . .

Add the following text to the **<type_traits>** synopsis [meta.type.synop] of [DuT12]. At the discretion of the Project Editor, the text may be inserted as a unit or may be distributed/merged among the various trait subclassifications.

```
// 20.9.7.1, const-volatile modifications:
template <class T>
  using remove_const_t    = typename remove_const<T>::type;
template <class T>
  using remove_volatile_t = typename remove_volatile<T>::type;
```

```
template <class T>
  using remove_cv_t       = typename remove_cv<T>::type;
template <class T>
  using add_const_t       = typename add_const<T>::type;
template <class T>
  using add_volatile_t    = typename add_volatile<T>::type;
template <class T>
  using add_cv_t          = typename add_cv<T>::type;

// 20.9.7.2, reference modifications:
template <class T>
  using remove_reference_t     = typename remove_reference<T>::type;
template <class T>
  using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
template <class T>
  using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// 20.9.7.3, sign modifications:
template <class T>
  using make_signed_t   = typename make_signed<T>::type;
template <class T>
  using make_unsigned_t = typename make_unsigned<T>::type;

// 20.9.7.4, array modifications:
template <class T>
  using remove_extent_t      = typename remove_extent<T>::type;
template <class T>
  using remove_all_extents_t = typename remove_all_extents<T>::type;

// 20.9.7.5, pointer modifications:
template <class T>
  using remove_pointer_t = typename remove_pointer<T>::type;
template <class T>
  using add_pointer_t    = typename add_pointer<T>::type;

// 20.9.7.6, other transformations:
template <size_t Len,
          std::size_t Align=default-alignment>  // see 20.9.7.6
  using aligned_storage_t = typename aligned_storage<Len,Align>::type;
template <std::size_t Len, class... Types>
  using aligned_union_t   = typename aligned_union<Len,Types...>::type;
template <class T>
  using decay_t           = typename decay<T>::type;
template <bool b, class T=void>
  using enable_if_t       = typename enable_if<b,T>::type;
template <bool b, class T, class F>
  using conditional_t     = typename conditional<b,T,F>::type;
template <class... T>
  using common_type_t     = typename common_type<T...>::type;
template <class T>
  using underlying_type_t = typename underlying_type<T>::type;
template <class T>
  using result_of_t       = typename result_of<T>::type;
```

## 4 Supplementary proposed wording

The following wording is provided in response to LWG's request that aliases for **::type** members be consistently provided for all the type traits, not only for those classified as *TransformationTraits*. Accordingly, this section provides the specifications needed in order to complete the set.

Add the following text to the **<type_traits>** synopsis [meta.type.synop] of [DuT12]. At the discretion of the Project Editor, the text may be inserted as a unit or may be distributed/merged among the various trait subclassifications.

```
// 20.9.4.1, primary type categories:
template <class T>
  using is_void_t                    = typename is_void<T>::type;
template <class T>
  using is_integral_t                = typename is_integral<T>::type;
template <class T>
  using is_floating_point_t          = typename is_floating_point<T>::type;
template <class T>
  using is_array_t                   = typename is_array<T>::type;
template <class T>
  using is_pointer_t                 = typename is_pointer<T>::type;
template <class T>
  using is_lvalue_reference_t        = typename is_lvalue_reference<T>::type;
template <class T>
  using is_rvalue_reference_t        = typename is_rvalue_reference<T>::type;
template <class T>
  using is_member_object_pointer_t   = typename is_member_object_pointer<T>::type;
template <class T>
  using is_member_function_pointer_t = typename is_member_function_pointer<T>::type;
template <class T>
  using is_enum_t                    = typename is_enum<T>::type;
template <class T>
  using is_union_t                   = typename is_union<T>::type;
template <class T>
  using is_class_t                   = typename is_class<T>::type;
template <class T>
  using is_function_t                = typename is_function<T>::type;

// 20.9.4.2, composite type categories:
template <class T>
  using is_reference_t      = typename is_reference<T>::type;
template <class T>
  using is_arithmetic_t     = typename is_arithmetic<T>::type;
template <class T>
  using is_fundamental_t    = typename is_fundamental<T>::type;
template <class T>
  using is_object_t         = typename is_object<T>::type;
template <class T>
  using is_scalar_t         = typename is_scalar<T>::type;
template <class T>
  using is_compound_t       = typename is_compound<T>::type;
template <class T>
```

```
  using is_member_pointer_t = typename is_member_pointer<T>::type;

// 20.9.4.3, type properties:
template <class T>
  using is_const_t             = typename is_const<T>::type;
template <class T>
  using is_volatile_t          = typename is_volatile<T>::type;
template <class T>
  using is_trivial_t           = typename is_trivial<T>::type;
template <class T>
  using is_trivially_copyable_t = typename is_trivially_copyable<T>::type;
template <class T>
  using is_standard_layout_t   = typename is_standard_layout<T>::type;
template <class T>
  using is_pod_t               = typename is_pod<T>::type;
template <class T>
  using is_literal_type_t      = typename is_literal_type<T>::type;
template <class T>
  using is_empty_t             = typename is_empty<T>::type;
template <class T>
  using is_polymorphic_t       = typename is_polymorphic<T>::type;
template <class T>
  using is_abstract_t          = typename is_abstract<T>::type;

template <class T>
  using is_signed_t   = typename is_signed<T>::type;
template <class T>
  using is_unsigned_t = typename is_unsigned<T>::type;

template <class T, class... Args>
  using is_constructible_t         = typename is_constructible<T, Args...>::type;
template <class T>
  using is_default_constructible_t = typename is_default_constructible<T>::type;
template <class T>
  using is_copy_constructible_t    = typename is_copy_constructible<T>::type;
template <class T>
  using is_move_constructible_t    = typename is_move_constructible<T>::type;

template <class T, class U>
  using is_assignable_t      = typename is_assignable<T, U>::type;
template <class T>
  using is_copy_assignable_t = typename is_copy_assignable<T>::type;
template <class T>
  using is_move_assignable_t = typename is_move_assignable<T>::type;

template <class T>
  using is_destructible_t = typename is_destructible<T>::type;

template <class T, class... Args>
  using is_trivially_constructible_t
  = typename is_trivially_constructible<T, Args...>::type;
template <class T>
  using is_trivially_default_constructible_t
  = typename is_trivially_default_constructible<T>::type;
```

```
template <class T>
  using is_trivially_copy_constructible_t
  = typename is_trivially_copy_constructible<T>::type;
template <class T>
  using is_trivially_move_constructible_t
  = typename is_trivially_move_constructible<T>::type;

template <class T, class U>
  using is_trivially_assignable_t
  = typename is_trivially_assignable<T, U>::type;
template <class T>
  using is_trivially_copy_assignable_t
  = typename is_trivially_copy_assignable<T>::type;
template <class T>
  using is_trivially_move_assignable_t
  = typename is_trivially_move_assignable<T>::type;
template <class T>
  using is_trivially_destructible_t
  = typename is_trivially_destructible<T>::type;

template <class T, class... Args>
  using is_nothrow_constructible_t
  = typename is_nothrow_constructible<T, Args...>::type;
template <class T>
  using is_nothrow_default_constructible_t
  = typename is_nothrow_default_constructible<T>::type;
template <class T>
  using is_nothrow_copy_constructible_t
  = typename is_nothrow_copy_constructible<T>::type;
template <class T>
  using is_nothrow_move_constructible_t
  = typename is_nothrow_move_constructible<T>::type;

template <class T, class U>
  using is_nothrow_assignable_t      = typename is_nothrow_assignable<T, U>::type;
template <class T>
  using is_nothrow_copy_assignable_t = typename is_nothrow_copy_assignable<T>::type;
template <class T>
  using is_nothrow_move_assignable_t = typename is_nothrow_move_assignable<T>::type;

template <class T>
  using is_nothrow_destructible_t = typename is_nothrow_destructible<T>::type;
template <class T>
  using has_virtual_destructor_t  = typename has_virtual_destructor<T>::type;

// 20.9.5, type property queries:
template <class T>
  using alignment_of_t = typename alignment_of<T>::type;
template <class T>
  using rank_t         = typename rank<T>::type;
template <class T, unsigned I = 0>
  using extent_t       = typename extent<T, I>::type;

// 20.9.6, type relations:
```

```
template <class T, class U>
  using is_same_t        = typename is_same<T, U>::type;
template <class Base, class Derived>
  using is_base_of_t     = typename is_base_of<Base, Derived>::type;
template <class From, class To>
  using is_convertible_t = typename is_convertible<From, To>::type;
```

## 5 Acknowledgments

Many thanks to the proofreaders of this paper's early drafts. Thanks also to Stefanus Du Toit for his contributions to the supplementary wording.

## 6 Bibliography

[AG05] David Abrahams and Aleksey Gurtovoy: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2005. ISBN: 0-321-22725-5.

[DuT12] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3485 (post-Portland mailing), 2012-11-02. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf.

## 7 Revision history

| **Version** | **Date** | **Changes** |
|---|---|---|
| 1 | 2013-03-12 | • Published as N3546. |
| 2 | 2013-04-18 | • Corrected **`result_of_t`** definition.<br>• Added supplementary wording requested by LWG.<br>• Acknowledged Stefanus's contribution to supplementary wording.<br>• Published as N3655. |