Document Number: N1713=04-0153
Date: October 20, 2004
Reply to: Pete Becker
Dinkumware, Ltd.
petebecker@acm.org

Proposed Resolution to TR1 Issues 3.12, 3.14, and 3.15

This paper proposes a resolution to TR1 issues 3.12, 3.14, and 3.15. It recommends an extensive (although largely mechanical) restructuring of the type-traits portion of TR1, so it contains a rewritten version of that clause rather than a series of edits.

Issue 3.14 is about the way that the various traits are described. The descriptions consist of a code snippet followed by text that describes the required behavior. In most cases this combination ends up overspecifying the template. For example:

```
template <class T> struct remove_reference{
   typedef T type;
};
template <class T> struct remove_reference<T&>{
   typedef T type;
};
```

1 `type` : defined to be a type that is the same as `T`, except any reference qualifier has been removed.

The paragraph marked “1” is the specification for the template `remove_reference`; the code snippet above it is one way to implement it.

The proposed resolution moves the definitions of the type traits into tables which list the name of the trait and its argument list in one column and the required behavior in another column. All of the templates in any table meet the requirements for only one of *UnaryTypeTraits*, *BinaryTypeTraits*, or *TransformationTraits*.

The proposed resolution also rewrites the definitions of *UnaryTypeTrait*, *BinaryTypeTrait*, and *TransformationTrait* to make them applicable to the traits that we added

Issue 3.12 is about the use of `true_type`, `false_type`, and other instantiations of the template `integral_constant`. Many of the type traits templates specify a nested type named `type` which is an instance of `integral_constant`, and also specify a conversion to that type. Issue 3.12 says that each such traits type should inherit from the appropriate instance of `integral_constant`. The revised text in this paper makes that change.

Issue 3.15 recommends changing a "Notes" entry in the requirements for `has_virtual_destructor` into normative text (4.9/6). This change also helps simplify several other descriptions by allowing the header for the third column in table 4 to be "Preconditions", which in turn means the various preconditions in that column don’t have to be separately labeled as such.

Note: I haven’t fixed bad page breaks or bad line breaks. That comes later.

My thanks to John Maddock for reading and commenting on several drafts of this proposal.

4 Metaprogramming and type traits [tr.meta]

1 This clause describes components used by C++ programs, particularly in templates, to: support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time.

2 The following subclauses describe type traits requirements, unary type traits, traits that describe relationships between types, and traits that perform transformations on types, as summarized in Table 1.

Table 1: Type traits library summary

Subclause	Header(s)
4.1 Requirements	
4.5 Unary type traits	<type_traits>
4.6 Relationships between types	<type_traits>
4.7 Transformations between types	<type_traits>

4.1 Requirements [tr.meta.rqmts]

A *UnaryTypeTrait* is a template that describes 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 property being described. It shall be *DefaultConstructible* and derived, directly or indirectly, from an instance of the template `integral_constant` (4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described.

A *BinaryTypeTrait* is a template that describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be *DefaultConstructible* and derived, directly or indirectly, from an instance of the template `integral_constant` (4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described.

A *TransformationTypeTrait* is a template that 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 named `type` which shall be a synonym for the modified type.

4.2 Header <type_traits> synopsis [tr.meta.type.synop]

```
namespace tr1{

// [4.3] helper class:
```

```
template <class T, T v> struct integral_constant;
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

// [4.5.1] primary type categories:
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// [4.5.2] composite type categories:
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// [4.5.3] type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_constructor;
template <class T> struct has_trivial_copy;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_constructor;
template <class T> struct has_nothrow_copy;
template <class T> struct has_nothrow_assign;
template <class T> struct has_virtual_destructor;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// [4.6] type relations:
template <class T, class U> struct is_same;
template <class From, class To> struct is_convertible;
```

```
template <class Base, class Derived> struct is_base_of;

// [4.7.1] const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

// [4.7.2] reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_reference;

// [4.7.3] array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

// [4.7.4] pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

// [4.8] other transformations:
template <std::size_t Len, std::size_t Align> struct aligned_storage;

} // namespace tr1
```

4.3 Helper classes [tr.meta.help]

```
template <class T, T v>
struct integral_constant
{
   static  const T                 value = v;
   typedef T                       value_type;
   typedef integral_constant<T,v> type;
};
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

1 The class template `integral_constant` and its associated typedefs `true_type` and `false_type` are used as base classes to define the interface for various type traits.

4.4 General Requirements [tr.meta.requirements]

1 Tables 2, 3, 4, and 6 define type predicates. Each type predicate `pred<T>` shall be a *UnaryTypeTrait* (4.1), derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`. Each type predicate `pred<T, U>` shall be a *BinaryTypeTrait* (4.1), derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`.

2 Table 5 defines various type queries. Each type query shall be a *UnaryTypeTrait* (4.1), derived directly or indirectly from `integral_constant<std::size_t, value>`, where `value` is the value of the property being queried.

3 Tables 7, 8, 9, and 10 define type transformations. Each transformation shall be a *TransformationTrait* (4.1).

4 Table 11 defines a template that can be instantiated to define a type with a specific alignment and size.

4.5 Unary Type Traits [tr.meta.unary]

1 This sub-clause contains templates that may be used to query the properties of a type at compile time.

2 For all of the class templates `X` declared in this clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of `X` require that the argument must be a complete type.

4.5.1 Primary Type Categories [tr.meta.unary.cat]

1 The primary type categories correspond to the descriptions given in section [basic.types] of the C++ standard.

2 For any given type `T`, exactly one of the primary type categories shall have its member `value` evaluate to `true`.

3 For any given type `T`, the result of applying one of these templates to `T`, and to *cv-qualified* `T` shall yield the same result.

4 The behavior of a program that adds specializations for any of the class templates defined in this clause is undefined.

Table 2: Primary Type Category Predicates

Template	Condition	Comments
`template <class T> struct is_void;`	`T` is `void` or a *cv-qualified* `void`	
`template <class T> struct is_integral;`	`T` is an integral type ([basic.fundamental])	
`template <class T> struct is_floating_point;`	`T` is a floating point type ([basic.fundamental])	
`template <class T> struct is_array;`	`T` is an array type ([basic.compound])	[*Note*: class template `array`, described in clause **??** of this technical report, is *not* an array type. *—end note*]
`template <class T> struct is_pointer;`	`T` is a pointer type ([basic.compound])	Pointer type here includes all function pointer types but not pointers to members or member functions.
`template <class T> struct is_reference;`	`T` is a reference type ([basic.fundamental])	Includes reference to a function type.
`template <class T> struct is_member_object_pointer;`	`T` is a pointer to data member	
`template <class T> struct is_member_function_pointer;`	`T` is a pointer to member function	
`template <class T> struct is_enum;`	`T` is an enumeration type ([basic.compound])	
`template <class T> struct is_union;`	`T` is a union type ([basic.compound])	

`template <class T> struct is_class;`	T is a class type ([basic.compound]) but not a union type	
`template <class T> struct is_function;`	T is a function type ([basic.compound])	

4.5.2 Composite type traits [tr.meta.unary.comp]

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section [basic.types].

2 For any given type T, the result of applying one of these templates to T, and to *cv-qualified* T shall yield the same result.

3 The behavior of a program that adds specializations for any of the class templates defined in this clause is undefined.

Table 3: Composite Type Category Predicates

Template	Condition	Comments
`template <class T> struct is_arithmetic;`	T is an arithmetic type ([basic.fundamental])	
`template <class T> struct is_fundamental;`	T is a fundamental type ([basic.fundamental])	
`template <class T> struct is_object;`	T is an object type ([basic.types])	
`template <class T> struct is_scalar;`	T is a scalar type ([basic.types])	
`template <class T> struct is_compound;`	T is a compound type ([basic.compound])	
`template <class T> struct is_member_pointer;`	T is a pointer to a member or member function	

4.5.3 Type properties [tr.meta.unary.prop]

1 These templates provide access to some of the more important properties of types; they reveal information which is available to the compiler, but which would not otherwise be detectable in C++ code.

2 It is unspecified whether the library defines any full or partial specialisations of any of these templates. A program may specialise any of these templates on a user-defined type, provided the semantics of the specialisation match those given for the template in its description.

Table 4: Type Property Predicates

Template	Condition	Preconditions
`template <class T> struct is_const;`	T is const-qualified ([basic.qualifier])	

`template <class T>` `struct is_volatile;`	T is volatile-qualified ([basic.qualifier])	
`template <class T>` `struct is_pod;`	T is a POD type ([basic.type])	T shall be a complete type.
`template <class T>` `struct is_empty;`	T is an empty class (10)	T shall be a complete type.
`template <class T>` `struct is_polymorphic;`	T is a polymorphic class (10.3)	T shall be a complete type.
`template <class T>` `struct is_abstract;`	T is an abstract class (10.4)	T shall be a complete type.
`template <class T>` `struct has_trivial_constructor;`	The default constructor for T is trivial (12.1)	T shall be a complete type.
`template <class T>` `struct has_trivial_copy;`	The copy constructor for T is trivial (12.8)	T shall be a complete type.
`template <class T>` `struct has_trivial_assign;`	The assignment operator for T is trivial (12.8)	T shall be a complete type.
`template <class T>` `struct has_trivial_destructor;`	The destructor for T is trivial (12.4)	T shall be a complete type.
`template <class T>` `struct has_nothrow_constructor;`	The default constructor for T has an empty exception specification or can otherwise be deduced never to throw an exception	T shall be a complete type.
`template <class T>` `struct has_nothrow_copy;`	The copy constructor for T has an empty exception specification or can otherwise be deduced never to throw an exception	T shall be a complete type.
`template <class T>` `struct has_nothrow_assign;`	The assignment operator for T has an empty exception specification or can otherwise be deduced never to throw an exception	T shall be a complete type.
`template <class T>` `struct has_virtual_destructor;`	T has a virtual destructor (12.4)	T shall be a complete type.
`template <class T>` `struct is_signed;`	T is a signed integral type ([basic.fundamental])	
`template <class T>` `struct is_unsigned;`	T is an unsigned integral type ([basic.fundamental])	

Table 5: Type Property Queries

Template	**value**

`template <class T> struct alignment_of;`	An integer value representing the number of bytes of the alignment of objects of type `T`; an object of type `T` may be allocated at an address that is a multiple of its alignment ([basic.types]). *Precondition:* `T` shall be a complete type.
`template <class T> struct rank;`	An integer value representing the rank of objects of type `T` (8.3.4). [*Note*: The term "rank" here is used to describe the number of dimensions of an array type. *—end note*]
`template <class T, unsigned I = 0> struct extent;`	An integer value representing the extent (dimension) of the `I`'th bound of objects of type T (8.3.4). If the type `T` is not an array type, has rank of less than `I`, or if `I == 0` and `T` is of type "array of unknown bound of U," then `value` shall evaluate to zero; otherwise `value` shall evaluate to the number of elements in the `I`'th array bound of `T`. [*Note*: The term "extent" here is used to describe the number of elements in an array type *—end note*]

3 [*Example*:

```
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);
```

—end example]

4 [*Example*:

```
 // the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert(extent<int[][4]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[][4], 1>::value) == 4);
```

—end example]

4.6 Relationships between types [tr.meta.rel]

Table 6: Type Relationship Predicates

Template	Condition	Comments
`template <class T, class U> struct is_same;`	`T` and `U` name the same type	

`template <class From, class To> struct is_convertible;`	An imaginary lvalue of type `From` is implicitly convertible to type `To` (4.0)	Special conversions involving string-literals and null-pointer constants are not considered (4.2, 4.10 and 4.11). No function-parameter adjustments (8.3.5) are made to type `To` when determining whether `From` is convertible to `To`; this implies that if type `To` is a function type or an array type, then the condition is false. See below.
`template <class Base, class Derived> struct is_base_of;`	`Base` is a base class of `Derived` ([class.derived]) or `Base` and `Derived` name the same type	*Preconditions:* `Base` and `Derived` shall be complete types.

1 The expression `is_convertible<From,To>::value` is ill-formed if:

- Type `From` is a void or incomplete type ([basic.types]).
- Type `To` is an incomplete, void or abstract type ([basic.types]).
- The conversion is ambiguous, for example if type `From` has multiple base classes of type `To` ([class.member.lookup]).
- Type `To` is of class type and the conversion would invoke a non-public constructor of `To` ([class.access] and [class.conv.ctor]).
- Type `From` is of class type and the conversion would invoke a non-public conversion operator of `From` ([class.access] and [class.conv.fct]).

4.7 Transformations between types [tr.meta.trans]

1 This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this header shall be a `TransformationTrait` (4.1).

4.7.1 Const-volatile modifications [tr.meta.trans.cv]

Table 7: Const-volatile modifications

Template	Comments
`template <class T> struct remove_const;`	The member typedef `type` shall be the same as `T` except that any top level const-qualifier has been removed. [*Example:* `remove_const<const volatile int>::type` evaluates to `volatile int`, whereas `remove_const<const int*>` is `const int*`. —*end example*]

template <class T> struct remove_volatile;	The member typedef type shall be the same as T except that any top level volatile-qualifier has been removed. [*Example:* remove_const<const volatile int>::type evaluates to const int, whereas remove_const<volatile int*> is volatile int*. *—end example*]
template <class T> struct remove_cv;	The member typedef type shall be the same as T except that any top level cv-qualifier has been removed. [*Example:* remove_cv<const volatile int>::type evaluates to int, where as remove_cv<const volatile int*> is const volatile int*. *—end example*]
template <class T> struct add_const;	If T is a reference, function, or top level const-qualified type, then type shall be the same type as T, otherwise T const.
template <class T> struct add_volatile;	If T is a reference, function, or top level volatile-qualified type, then type shall be the same type as T, otherwise T volatile.
template <class T> struct add_cv;	The member typedef type shall be the same type as add_const<add_volatile<T>::type>::type.

4.7.2 Reference modifications [tr.meta.trans.ref]

Table 8: Reference modifications

Template	Comments
template <class T> struct remove_reference;	The member typedef type shall be the same as T, except any reference qualifier has been removed.
template <class T> struct add_reference;	If T is a reference type, then the member typedef type shall be T, otherwise T& .

4.7.3 Array modifications [tr.meta.trans.arr]

Table 9: Array modifications

Template	Comments
template <class T> struct remove_extent;	If T is "array of U", the member typedef type shall be U, otherwise T. For multidimensional arrays, only the first array dimension is removed. For a type "array of const U", the resulting type is const U.
template <class T> struct remove_all_extents;	If T is "multi-dimensional array of U", the resulting member typedef type is U, otherwise T.

1 [*Example*

```
// the following assertions hold:
assert((is_same<remove_extent<int>::type, int>::value));
assert((is_same<remove_extent<int[2]>::type, int>::value));
assert((is_same<remove_extent<int[2][3]>::type, int[3]>::value));
assert((is_same<remove_extent<int[][3]>::type, int[3]>::value));
```

—end example]

2 [*Example*

```
// the following assertions hold:
assert((is_same<remove_all_extents<int>::type, int>::value));
assert((is_same<remove_all_extents<int[2]>::type, int>::value));
assert((is_same<remove_all_extents<int[2][3]>::type, int>::value));
assert((is_same<remove_all_extents<int[][3]>::type, int>::value));
```

—end example]

4.7.4 Pointer modifications [tr.meta.trans.ptr]

Table 10: Pointer modifications

Template	Comments
`template <class T> struct remove_pointer;`	The member typedef `type` shall be the same as `T`, except any top level indirection has been removed. Note: pointers to members are left unchanged by `remove_pointer`.
`template <class T> struct add_pointer;`	The member typedef `type` shall be the same as `remove_reference<T>::type*` if `T` is a reference type, otherwise `T*`.

4.8 Other transformations [tr.meta.trans.other]

Table 11: Other transformations

Template	Comments
`template <std::size_t Len, std::size_t Align> struct aligned_storage;`	The member typedef `type` shall be a POD type with size *Len* and alignment *Align*, suitable for use as uninitialized storage for any object of a type whose size is *Len* and whose alignment is *Align*.

4.9 Implementation requirements [tr.meta.req]

1 The behaviour of all the class templates defined in `<type_traits>` shall conform to the specifications given, except where noted below.

2 [*Note:* The latitude granted to implementers in this clause is temporary, and is expected to be removed in future revisions of this document. *—end note*]

3 If there is no means by which the implementation can differentiate between class and union types, then the class templates `is_class` and `is_union` need not be provided.

4 If there is no means by which the implementation can detect polymorphic types, then the class template `is_polymorphic` need not be provided.

5 If there is no means by which the implementation can detect abstract types, then the class template `is_abstract` need not be provided.

6 If there is no means by which an implementation can determine whether a type `T` has a virtual destructor, *e.g.* a pure library implementation with no compiler support, then `has_virtual_destructor<T>` shall be derived, directly or indirectly, from `false_type` (4.1).

7 It is unspecified under what circumstances, if any, `is_empty<T>::value` evaluates to `true`.

8 It is unspecified under what circumstances, if any, `is_pod<T>::value` evaluates to `true`, except that, for all types `T`:

```
is_pod<T>::value == is_pod<remove_extent<T>::type>::value
is_pod<T>::value == is_pod<T const volatile>::value
is_pod<T>::value >= (is_scalar<T>::value || is_void<T>::value)
```

9 It is unspecified under what circumstances, if any, `has_trivial_*<T>::value` evaluates to `true`, except that:

```
has_trivial_*<T>::value ==
    has_trivial_*<remove_extent<T>::type>::value
has_trivial_*<T>::value >=
    is_pod<T>::value
```

10 It is unspecified under what circumstances, if any, `has_nothrow_*<T>::value` evaluates to `true`.

11 There are trait templates whose semantics do not require their argument(s) to be completely defined, nor does such completeness in any way affect the exact definition of the traits class template specializations. However, in the absence of compiler support these traits cannot be implemented without causing implicit instantiation of their arguments; in particular: `is_class`, `is_enum`, and `is_scalar`. For these templates, it is unspecified whether their template argument(s) are implicitly instantiated when the traits class is itself instantiated.