Boost.TypeTraits

various authors

Table of Contents

A printer-friendly PDF version of this manual is also available.

Introduction

The Boost type-traits library contains a set of very specific traits classes, each of which encapsulate a single trait from the C++ type system; for example, is a type a pointer or a reference type? Or does a type have a trivial constructor, or a const-qualifier?

The type-traits classes share a unified design: each class inherits from a the type true_type if the type has the specified property and inherits from false_type otherwise.

The type-traits library also contains a set of classes that perform a specific transformation on a type; for example, they can remove a top-level const or volatile qualifier from a type. Each class that performs a transformation defines a single typedef-member `type` that is the result of the transformation.

Background and Tutorial

The following is an updated version of the article "C++ Type traits" by John Maddock and Steve Cleary that appeared in the October 2000 issue of Dr Dobb's Journal.

Generic programming (writing code which works with any data type meeting a set of requirements) has become the method of choice for providing reusable code. However, there are times in generic programming when "generic" just isn't good enough - sometimes the differences between types are too large for an efficient generic implementation. This is when the traits technique becomes important - by encapsulating those properties that need to be considered on a type by type basis inside a traits class, we can minimize the amount of code that has to differ from one type to another, and maximize the amount of generic code.

Consider an example: when working with character strings, one common operation is to determine the length of a null terminated string. Clearly it's possible to write generic code that can do this, but it turns out that there are much more efficient methods available: for example, the C library functions `strlen` and `wcslen` are usually written in assembler, and with suitable hardware support can be considerably faster than a generic version written in C++. The authors of the C++ standard library realized this, and abstracted the properties of `char` and `wchar_t` into the class `char_traits`. Generic code that works with character strings can simply use `char_traits<>::length` to determine the length of a null terminated string, safe in the knowledge that specializations of `char_traits` will use the most appropriate method available to them.

Type Traits

Class `char_traits` is a classic example of a collection of type specific properties wrapped up in a single class - what Nathan Myers termed a *baggage class*[1]. In the Boost type-traits library, we[2] have written a set of very specific traits classes, each of which encapsulate a single trait from the C++ type system; for example, is a type a pointer or a reference type? Or does a type have a trivial constructor, or a const-qualifier? The type-traits classes share a unified design: each class inherits from a the type true_type if the type has the specified property and inherits from false_type otherwise. As we will show, these classes can be used in generic programming to determine the properties of a given type and introduce optimizations that are appropriate for that case.

The type-traits library also contains a set of classes that perform a specific transformation on a type; for example, they can remove a top-level const or volatile qualifier from a type. Each class that performs a transformation defines a single typedef-member `type` that is the result of the transformation. All of the type-traits classes are defined inside namespace `boost`; for brevity, namespace-qualification is omitted in most of the code samples given.

Implementation

There are far too many separate classes contained in the type-traits library to give a full implementation here - see the source code in the Boost library for the full details - however, most of the implementation is fairly repetitive anyway, so here we will just give you a flavor for how some of the classes are implemented. Beginning with possibly the simplest class in the library, `is_void<T>` inherits from `true_type` only if `T` is `void`.

```
template <typename T>
struct is_void : public false_type{};

template <>
struct is_void<void> : public true_type{};
```

Here we define a primary version of the template class `is_void`, and provide a full-specialization when `T` is `void`. While full specialization of a template class is an important technique, sometimes we need a solution that is halfway between a fully generic solution, and a full specialization. This is exactly the situation for which the standards committee defined partial template-class specialization. As an example, consider the class `boost::is_pointer<T>`: here we needed a primary version that handles all the cases where T is not a pointer, and a partial specialization to handle all the cases where T is a pointer:

```
template <typename T>
struct is_pointer : public false_type{};

template <typename T>
struct is_pointer<T*> : public true_type{};
```

The syntax for partial specialization is somewhat arcane and could easily occupy an article in its own right; like full specialization, in order to write a partial specialization for a class, you must first declare the primary template. The partial specialization contains an extra <...> after the class name that contains the partial specialization parameters; these define the types that will bind to that partial specialization rather than the default template. The rules for what can appear in a partial specialization are somewhat convoluted, but as a rule of thumb if you can legally write two function overloads of the form:

```
void foo(T);
void foo(U);
```

Then you can also write a partial specialization of the form:

```
template <typename T>
class c{ /*details*/ };

template <typename T>
class c<U>{ /*details*/ };
```

This rule is by no means foolproof, but it is reasonably simple to remember and close enough to the actual rule to be useful for everyday use.

As a more complex example of partial specialization consider the class `remove_extent<T>`. This class defines a single typedef-member `type` that is the same type as T but with any top-level array bounds removed; this is an example of a traits class that performs a transformation on a type:

```
template <typename T>
struct remove_extent
{ typedef T type; };

template <typename T, std::size_t N>
struct remove_extent<T[N]>
{ typedef T type; };
```

The aim of `remove_extent` is this: imagine a generic algorithm that is passed an array type as a template parameter, `remove_extent` provides a means of determining the underlying type of the array. For example `remove_extent<int[4][5]>::type` would evaluate to the type `int[5]`. This example also shows that the number of template parameters in a partial specialization does not have to match the number in the default template. However, the number of parameters that appear after the class name do have to match the number and type of the parameters in the default template.

Optimized copy

As an example of how the type traits classes can be used, consider the standard library algorithm copy:

```
template<typename Iter1, typename Iter2>
Iter2 copy(Iter1 first, Iter1 last, Iter2 out);
```

Obviously, there's no problem writing a generic version of copy that works for all iterator types `Iter1` and `Iter2`; however, there are some circumstances when the copy operation can best be performed by a call to `memcpy`. In order to implement copy in terms of `memcpy` all of the following conditions need to be met:

- Both of the iterator types `Iter1` and `Iter2` must be pointers.

- Both `Iter1` and `Iter2` must point to the same type - excluding const and volatile-qualifiers.
- The type pointed to by `Iter1` must have a trivial assignment operator.

By trivial assignment operator we mean that the type is either a scalar type[3] or:

- The type has no user defined assignment operator.
- The type does not have any data members that are references.
- All base classes, and all data member objects must have trivial assignment operators.

If all these conditions are met then a type can be copied using `memcpy` rather than using a compiler generated assignment operator. The type-traits library provides a class `has_trivial_assign`, such that `has_trivial_assign<T>::value` is true only if T has a trivial assignment operator. This class "just works" for scalar types, but has to be explicitly specialised for class/struct types that also happen to have a trivial assignment operator. In other words if has_trivial_assign gives the wrong answer, it will give the "safe" wrong answer - that trivial assignment is not allowable.

The code for an optimized version of copy that uses `memcpy` where appropriate is given in the examples. The code begins by defining a template function `do_copy` that performs a "slow but safe" copy. The last parameter passed to this function may be either a `true_type` or a `false_type`. Following that there is an overload of docopy that uses `memcpy`: this time the iterators are required to actually be pointers to the same type, and the final parameter must be a ` true_type. Finally, the version of copy calls docopy`, passing ` has_trivial_assign<value_type>()` as the final parameter: this will dispatch to the optimized version where appropriate, otherwise it will call the "slow but safe version".

Was it worth it?

It has often been repeated in these columns that "premature optimization is the root of all evil" [4]. So the question must be asked: was our optimization premature? To put this in perspective the timings for our version of copy compared a conventional generic copy[5] are shown in table 1.

Clearly the optimization makes a difference in this case; but, to be fair, the timings are loaded to exclude cache miss effects - without this accurate comparison between algorithms becomes difficult. However, perhaps we can add a couple of caveats to the premature optimization rule:

- If you use the right algorithm for the job in the first place then optimization will not be required; in some cases, memcpy is the right algorithm.
- If a component is going to be reused in many places by many people then optimizations may well be worthwhile where they would not be so for a single case - in other words, the likelihood that the optimization will be absolutely necessary somewhere, sometime is that much higher. Just as importantly the perceived value of the stock implementation will be higher: there is no point standardizing an algorithm if users reject it on the grounds that there are better, more heavily optimized versions available.

Table 1. Time taken to copy 1000 elements using `copy<const T*, T*>` (times in micro-seconds)

Version	T	Time
"Optimized" copy	char	0.99
Conventional copy	char	8.07
"Optimized" copy	int	2.52
Conventional copy	int	8.02

Pair of References

The optimized copy example shows how type traits may be used to perform optimization decisions at compile-time. Another important usage of type traits is to allow code to compile that otherwise would not do so unless excessive partial specialization is used. This

is possible by delegating partial specialization to the type traits classes. Our example for this form of usage is a pair that can hold references [6].

First, let us examine the definition of `std::pair`, omitting the comparison operators, default constructor, and template copy constructor for simplicity:

```
template <typename T1, typename T2>
struct pair
{
typedef T1 first_type;
typedef T2 second_type;

T1 first;
T2 second;

pair(const T1 & nfirst, const T2 & nsecond)
:first(nfirst), second(nsecond) { }
};
```

Now, this "pair" cannot hold references as it currently stands, because the constructor would require taking a reference to a reference, which is currently illegal [7]. Let us consider what the constructor's parameters would have to be in order to allow "pair" to hold non-reference types, references, and constant references:

Table 2. Required Constructor Argument Types

Type of `T1`	**Type of parameter to initializing constructor**
T	const T &
T &	T &
const T &	const T &

A little familiarity with the type traits classes allows us to construct a single mapping that allows us to determine the type of parameter from the type of the contained class. The type traits classes provide a transformation add_reference, which adds a reference to its type, unless it is already a reference.

Table 3. Using add_reference to synthesize the correct constructor type

Type of `T1`	**Type of `const T1`**	**Type of `add_reference<const T1>::type`**
T	const T	const T &
T &	T & [8]	T &
const T &	const T &	const T &

This allows us to build a primary template definition for `pair` that can contain non-reference types, reference types, and constant reference types:

```
template <typename T1, typename T2>
struct pair
{
typedef T1 first_type;
typedef T2 second_type;

T1 first;
T2 second;

pair(boost::add_reference<const T1>::type nfirst,
      boost::add_reference<const T2>::type nsecond)
:first(nfirst), second(nsecond) { }
};
```

Add back in the standard comparison operators, default constructor, and template copy constructor (which are all the same), and you have a `std::pair` that can hold reference types!

This same extension could have been done using partial template specialization of `pair`, but to specialize `pair` in this way would require three partial specializations, plus the primary template. Type traits allows us to define a single primary template that adjusts itself auto-magically to any of these partial specializations, instead of a brute-force partial specialization approach. Using type traits in this fashion allows programmers to delegate partial specialization to the type traits classes, resulting in code that is easier to maintain and easier to understand.

Conclusion

We hope that in this article we have been able to give you some idea of what type-traits are all about. A more complete listing of the available classes are in the boost documentation, along with further examples using type traits. Templates have enabled C++ uses to take the advantage of the code reuse that generic programming brings; hopefully this article has shown that generic programming does not have to sink to the lowest common denominator, and that templates can be optimal as well as generic.

Acknowledgements

The authors would like to thank Beman Dawes and Howard Hinnant for their helpful comments when preparing this article.

References

1. Nathan C. Myers, C++ Report, June 1995.

2. The type traits library is based upon contributions by Steve Cleary, Beman Dawes, Howard Hinnant and John Maddock: it can be found at www.boost.org.

3. A scalar type is an arithmetic type (i.e. a built-in integer or floating point type), an enumeration type, a pointer, a pointer to member, or a const- or volatile-qualified version of one of these types.

4. This quote is from Donald Knuth, ACM Computing Surveys, December 1974, pg 268.

5. The test code is available as part of the boost utility library (see algo_opt_examples.cpp), the code was compiled with gcc 2.95 with all optimisations turned on, tests were conducted on a 400MHz Pentium II machine running Microsoft Windows 98.

6. John Maddock and Howard Hinnant have submitted a "compressed_pair" library to Boost, which uses a technique similar to the one described here to hold references. Their pair also uses type traits to determine if any of the types are empty, and will derive instead of contain to conserve space -- hence the name "compressed".

7. This is actually an issue with the C++ Core Language Working Group (issue #106), submitted by Bjarne Stroustrup. The tentative resolution is to allow a "reference to a reference to T" to mean the same thing as a "reference to T", but only in template instantiation, in a method similar to multiple cv-qualifiers.

8. For those of you who are wondering why this shouldn't be const-qualified, remember that references are always implicitly constant (for example, you can't re-assign a reference). Remember also that "const T &" is something completely different. For this reason, cv-qualifiers on template type arguments that are references are ignored.

Type Traits by Category

Type Traits that Describe the Properties of a Type

These traits are all *value traits*, which is to say the traits classes all inherit from integral_constant, and are used to access some numerical property of a type. Often this is a simple true or false Boolean value, but in a few cases may be some other integer value (for example when dealing with type alignments, or array bounds: see `alignment_of`, `rank` and `extent`).

Categorizing a Type

These traits identify what "kind" of type some type `T` is. These are split into two groups: primary traits which are all mutually exclusive, and composite traits that are compositions of one or more primary traits.

For any given type, exactly one primary type trait will inherit from true_type, and all the others will inherit from false_type, in other words these traits are mutually exclusive.

This means that `is_integral<T>::value` and `is_floating_point<T>::value` will only ever be true for built-in types; if you want to check for a user-defined class type that behaves "as if" it is an integral or floating point type, then use the `std::numeric_limits template` instead.

Synopsis:

```
template <class T>
struct is_array;

template <class T>
struct is_class;

template <class T>
struct is_complex;

template <class T>
struct is_enum;

template <class T>
struct is_floating_point;

template <class T>
struct is_function;

template <class T>
struct is_integral;

template <class T>
struct is_member_function_pointer;

template <class T>
struct is_member_object_pointer;

template <class T>
struct is_pointer;

template <class T>
struct is_reference;

template <class T>
struct is_union;

template <class T>
struct is_void;
```

The following traits are made up of the union of one or more type categorizations. A type may belong to more than one of these categories, in addition to one of the primary categories.

```
template <class T>
struct is_arithmetic;

template <class T>
struct is_compound;

template <class T>
struct is_fundamental;

template <class T>
struct is_member_pointer;

template <class T>
struct is_object;

template <class T>
struct is_scalar;
```

General Type Properties

The following templates describe the general properties of a type.

Synopsis:

```
template <class T>
struct alignment_of;

template <class T>
struct has_new_operator;

template <class T>
struct has_nothrow_assign;

template <class T>
struct has_nothrow_constructor;

template <class T>
struct has_nothrow_default_constructor;

template <class T>
struct has_nothrow_copy;

template <class T>
struct has_nothrow_copy_constructor;

template <class T>
struct has_trivial_assign;

template <class T>
struct has_trivial_constructor;

template <class T>
struct has_trivial_default_constructor;

template <class T>
struct has_trivial_copy;

template <class T>
struct has_trivial_copy_constructor;

template <class T>
struct has_trivial_destructor;

template <class T>
struct has_virtual_destructor;

template <class T>
struct is_abstract;

template <class T>
struct is_const;

template <class T>
struct is_empty;

template <class T>
struct is_stateless;

template <class T>
struct is_pod;

template <class T>
struct is_polymorphic;

template <class T>
struct is_signed;
```

```
template <class T>
struct is_unsigned;

template <class T>
struct is_volatile;

template <class T, std::size_t N = 0>
struct extent;

template <class T>
struct rank;
```

Relationships Between Two Types

These templates determine the whether there is a relationship between two types:

Synopsis:

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

template <class Base, class Derived>
struct is_virtual_base_of;

template <class From, class To>
struct is_convertible;

template <class T, class U>
struct is_same;
```

Type Traits that Transform One Type to Another

The following templates transform one type to another, based upon some well-defined rule. Each template has a single member called `type` that is the result of applying the transformation to the template argument `T`.

Synopsis:

```
template <class T>
struct add_const;

template <class T>
struct add_cv;

template <class T>
struct add_pointer;

template <class T>
struct add_reference;

template <class T>
struct add_volatile;

template <class T>
struct decay;

template <class T>
struct floating_point_promotion;

template <class T>
struct integral_promotion;

template <class T>
struct make_signed;

template <class T>
struct make_unsigned;

template <class T>
struct promote;

template <class T>
struct remove_all_extents;

template <class T>
struct remove_const;

template <class T>
struct remove_cv;

template <class T>
struct remove_extent;

template <class T>
struct remove_pointer;

template <class T>
struct remove_reference;

template <class T>
struct remove_volatile;
```

Broken Compiler Workarounds:

For all of these templates support for partial specialization of class templates is required to correctly implement the transformation. On the other hand, practice shows that many of the templates from this category are very useful, and often essential for implementing some generic libraries. Lack of these templates is often one of the major limiting factors in porting those libraries to compilers that do not yet support this language feature. As some of these compilers are going to be around for a while, and at least one of them is very wide-spread, it was decided that the library should provide workarounds where possible.

The basic idea behind the workaround is to manually define full specializations of all type transformation templates for all fundamental types, and all their 1st and 2nd rank cv-[un]qualified derivative pointer types, and to provide a user-level macro that will define all the explicit specializations needed for any user-defined type T.

The first part guarantees the successful compilation of something like this:

```
BOOST_STATIC_ASSERT((is_same<char, remove_reference<char&>::type>::value));
BOOST_STATIC_ASSERT((is_same<char const, remove_reference<char const&>::type>::value));
BOOST_STATIC_ASSERT((is_same<char volatile, remove_reference<char volatile&>::type>::value));
BOOST_STATIC_ASSERT((is_same<char const volatile, remove_reference<char const volat↵
ile&>::type>::value));
BOOST_STATIC_ASSERT((is_same<char*, remove_reference<char*&>::type>::value));
BOOST_STATIC_ASSERT((is_same<char const*, remove_reference<char const*&>::type>::value));
...
BOOST_STATIC_ASSERT((is_same<char const volatile* const volatile* const volatile, remove_refer↵
ence<char const volatile* const volatile* const volatile&>::type>::value));
```

and the second part provides the library's users with a mechanism to make the above code work not only for `char`, `int` or other built-in type, but for their own types as well:

```
namespace myspace{
   struct MyClass {};
}
// declare this at global scope:
BOOST_BROKEN_COMPILER_TYPE_TRAITS_SPECIALIZATION(myspace::MyClass)
// transformations on myspace::MyClass now work:
BOOST_STATIC_ASSERT((is_same<myspace::MyClass, remove_reference<myspace::MyClass&>::type>::value));
BOOST_STATIC_ASSERT((is_same<myspace::MyClass, remove_const<myspace::MyC↵
lass const>::type>::value));
// etc.
```

Note that the macro BOOST_BROKEN_COMPILER_TYPE_TRAITS_SPECIALIZATION evaluates to nothing on those compilers that **do** support partial specialization.

Synthesizing Types with Specific Alignments

Some low level memory management routines need to synthesize a POD type with specific alignment properties. The template `type_with_alignment` finds the smallest type with a specified alignment, while template `aligned_storage` creates a type with a specific size and alignment.

Synopsis

```
template <std::size_t Align>
struct type_with_alignment;

template <std::size_t Size, std::size_t Align>
struct aligned_storage;
```

Decomposing Function Types

The class template function_traits extracts information from function types (see also is_function). This traits class allows you to tell how many arguments a function takes, what those argument types are, and what the return type is.

Synopsis

```
template <std::size_t Align>
struct function_traits;
```

User Defined Specializations

Occationally the end user may need to provide their own specialization for one of the type traits - typically where intrinsic compiler support is required to implement a specific trait fully. These specializations should derive from boost::true_type or boost::false_type as appropriate:

```
#include <boost/type_traits/is_pod.hpp>
#include <boost/type_traits/is_class.hpp>
#include <boost/type_traits/is_union.hpp>

struct my_pod{};
struct my_union
{
   char c;
   int i;
};

namespace boost
{
   template<>
   struct is_pod<my_pod> : public true_type{};

   template<>
   struct is_pod<my_union> : public true_type{};

   template<>
   struct is_union<my_union> : public true_type{};

   template<>
   struct is_class<my_union> : public false_type{};
}
```

Support for Compiler Intrinsics

There are some traits that can not be implemented within the current C++ language: to make these traits "just work" with user defined types, some kind of additional help from the compiler is required. Currently (April 2008) Visual C++ 8 and 9, GNU GCC 4.3 and MWCW 9 provide the necessary intrinsics, and other compilers will no doubt follow in due course.

The Following traits classes always need compiler support to do the right thing for all types (but all have safe fallback positions if this support is unavailable):

- is_union
- is_pod
- has_trivial_constructor
- has_trivial_copy
- has_trivial_assign
- has_trivial_destructor
- has_nothrow_constructor
- has_nothrow_copy

- has_nothrow_assign
- has_virtual_destructor

The following traits classes can't be portably implemented in the C++ language, although in practice, the implementations do in fact do the right thing on all the compilers we know about:

- is_empty
- is_polymorphic

The following traits classes are dependent on one or more of the above:

- is_class
- is_stateless

The hooks for compiler-intrinsic support are defined in boost/type_traits/intrinsics.hpp, adding support for new compilers is simply a matter of defining one of more of the following macros:

Table 4. Macros for Compiler Intrinsics

BOOST_IS_UNION(T)	**Should evaluate to true if T is a union type**
BOOST_IS_POD(T)	Should evaluate to true if T is a POD type
BOOST_IS_EMPTY(T)	Should evaluate to true if T is an empty struct or union
BOOST_HAS_TRIVIAL_CONSTRUCTOR(T)	Should evaluate to true if the default constructor for T is trivial (i.e. has no effect)
BOOST_HAS_TRIVIAL_COPY(T)	Should evaluate to true if T has a trivial copy constructor (and can therefore be replaced by a call to memcpy)
BOOST_HAS_TRIVIAL_ASSIGN(T)	Should evaluate to true if T has a trivial assignment operator (and can therefore be replaced by a call to memcpy)
BOOST_HAS_TRIVIAL_DESTRUCTOR(T)	Should evaluate to true if T has a trivial destructor (i.e. ~T() has no effect)
BOOST_HAS_NOTHROW_CONSTRUCTOR(T)	Should evaluate to true if `T x;` can not throw
BOOST_HAS_NOTHROW_COPY(T)	Should evaluate to true if `T(t)` can not throw
BOOST_HAS_NOTHROW_ASSIGN(T)	Should evaluate to true if `T t, u; t = u` can not throw
BOOST_HAS_VIRTUAL_DESTRUCTOR(T)	Should evaluate to true T has a virtual destructor
BOOST_IS_ABSTRACT(T)	Should evaluate to true if T is an abstract type
BOOST_IS_BASE_OF(T,U)	Should evaluate to true if T is a base class of U
BOOST_IS_CLASS(T)	Should evaluate to true if T is a class type
BOOST_IS_CONVERTIBLE(T,U)	Should evaluate to true if T is convertible to U
BOOST_IS_ENUM(T)	Should evaluate to true is T is an enum
BOOST_IS_POLYMORPHIC(T)	Should evaluate to true if T is a polymorphic type
BOOST_ALIGNMENT_OF(T)	Should evaluate to the alignment requirements of type T.

MPL Interoperability

All the value based traits in this library conform to MPL's requirements for an Integral Constant type: that includes a number of rather intrusive workarounds for broken compilers.

Purely as an implementation detail, this means that `true_type` inherits from `boost::mpl::true_`, `false_type` inherits from `boost::mpl::false_`, and `integral_constant<T, v>` inherits from `boost::mpl::integral_c<T,v>` (provided `T` is not `bool`)

Examples

An Optimized Version of std::copy

Demonstrates a version of `std::copy` that uses `has_trivial_assign` to determine whether to use `memcpy` to optimise the copy operation (see copy_example.cpp):

```
//
// opt::copy
// same semantics as std::copy
// calls memcpy where appropriate.
//

namespace detail{

template<typename I1, typename I2, bool b>
I2 copy_imp(I1 first, I1 last, I2 out, const boost::integral_constant<bool, b>&)
{
   while(first != last)
   {
      *out = *first;
      ++out;
      ++first;
   }
   return out;
}

template<typename T>
T* copy_imp(const T* first, const T* last, T* out, const boost::true_type&)
{
   memcpy(out, first, (last-first)*sizeof(T));
   return out+(last-first);
}


}

template<typename I1, typename I2>
inline I2 copy(I1 first, I1 last, I2 out)
{
   //
   // We can copy with memcpy if T has a trivial assignment operator,
   // and if the iterator arguments are actually pointers (this last
   // requirement we detect with overload resolution):
   //
   typedef typename std::iterator_traits<I1>::value_type value_type;
   return detail::copy_imp(first, last, out, boost::has_trivial_assign<value_type>());
}
```

An Optimised Version of std::fill

Demonstrates a version of `std::fill` that uses `has_trivial_assign` to determine whether to use `memset` to optimise the fill operation (see fill_example.cpp):

```
//
// fill
// same as std::fill, but uses memset where appropriate
//
namespace detail{

template <typename I, typename T, bool b>
void do_fill(I first, I last, const T& val, const boost::integral_constant<bool, b>&)
{
   while(first != last)
   {
      *first = val;
      ++first;
   }
}

template <typename T>
void do_fill(T* first, T* last, const T& val, const boost::true_type&)
{
   std::memset(first, val, last-first);
}

}

template <class I, class T>
inline void fill(I first, I last, const T& val)
{
   //
   // We can do an optimised fill if T has a trivial assignment
   // operator and if it's size is one:
   //
   typedef boost::integral_constant<bool,
      ::boost::has_trivial_assign<T>::value && (sizeof(T) == 1)> truth_type;
   detail::do_fill(first, last, val, truth_type());
}
```

An Example that Omits Destructor Calls For Types with Trivial Destructors

Demonstrates a simple algorithm that uses `__has_trivial_destruct` to determine whether to destructors need to be called (see trivial_destructor_example.cpp):

```
//
// algorithm destroy_array:
// The reverse of std::unitialized_copy, takes a block of
// initialized memory and calls destructors on all objects therein.
//

namespace detail{

template <class T>
void do_destroy_array(T* first, T* last, const boost::false_type&)
{
   while(first != last)
   {
      first->~T();
      ++first;
   }
}

template <class T>
inline void do_destroy_array(T* first, T* last, const boost::true_type&)
{
}

} // namespace detail

template <class T>
inline void destroy_array(T* p1, T* p2)
{
   detail::do_destroy_array(p1, p2, ::boost::has_trivial_destructor<T>());
}
```

An improved Version of std::iter_swap

Demonstrates a version of `std::iter_swap` that use type traits to determine whether an it's arguments are proxying iterators or not, if they're not then it just does a `std::swap` of it's dereferenced arguments (the same as `std::iter_swap` does), however if they are proxying iterators then takes special care over the swap to ensure that the algorithm works correctly for both proxying iterators, and even iterators of different types (see iter_swap_example.cpp):

```
//
// iter_swap:
// tests whether iterator is a proxying iterator or not, and
// uses optimal form accordingly:
//
namespace detail{

template <typename I>
static void do_swap(I one, I two, const boost::false_type&)
{
   typedef typename std::iterator_traits<I>::value_type v_t;
   v_t v = *one;
   *one = *two;
   *two = v;
}
template <typename I>
static void do_swap(I one, I two, const boost::true_type&)
{
   using std::swap;
   swap(*one, *two);
}

}

template <typename I1, typename I2>
inline void iter_swap(I1 one, I2 two)
{
   //
   // See is both arguments are non-proxying iterators,
   // and if both iterator the same type:
   //
   typedef typename std::iterator_traits<I1>::reference r1_t;
   typedef typename std::iterator_traits<I2>::reference r2_t;

   typedef boost::integral_constant<bool,
      ::boost::is_reference<r1_t>::value
      && ::boost::is_reference<r2_t>::value
      && ::boost::is_same<r1_t, r2_t>::value> truth_type;

   detail::do_swap(one, two, truth_type());
}
```

Convert Numeric Types and Enums to double

Demonstrates a conversion of Numeric Types and enum types to double:

```
template<class T>
inline double to_double(T const& value)
{
    typedef typename boost::promote<T>::type promoted;
    return boost::numeric::converter<double,promoted>::convert(value);
}
```

Alphabetical Reference

add_const

```
template <class T>
struct add_const
{
   typedef see-below type;
};
```

type: The same type as `T const` for all `T`.

C++ Standard Reference: 3.9.3.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/add_const.hpp>` or `#include <boost/type_traits.hpp>`

Table 5. Examples

Expression	Result Type
`add_const<int>::type`	`int const`
`add_const<int&>::type`	`int&`
`add_const<int*>::type`	`int* const`
`add_const<int const>::type`	`int const`

add_cv

```
template <class T>
struct add_cv
{
   typedef see-below type;
};
```

type: The same type as `T const volatile` for all `T`.

C++ Standard Reference: 3.9.3.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/add_cv.hpp>` or `#include <boost/type_traits.hpp>`

Table 6. Examples

Expression	Result Type
`add_cv<int>::type`	`int const volatile`
`add_cv<int&>::type`	`int&`
`add_cv<int*>::type`	`int* const volatile`
`add_cv<int const>::type`	`int const volatile`

add_pointer

```
template <class T>
struct add_pointer
{
   typedef see-below type;
};
```

type: The same type as `remove_reference<T>::type*`.

The rationale for this template is that it produces the same type as `TYPEOF(&t)`, where `t` is an object of type `T`.

C++ Standard Reference: 8.3.1.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/add_pointer.hpp>` or `#include <boost/type_traits.hpp>`

Table 7. Examples

Expression	Result Type
`add_pointer<int>::type`	`int*`
`add_pointer<int const&>::type`	`int const*`
`add_pointer<int*>::type`	`int**`
`add_pointer<int*&>::type`	`int**`

add_reference

```
template <class T>
struct add_reference
{
   typedef see-below type;
};
```

type: If `T` is not a reference type then `T&`, otherwise `T`.

C++ Standard Reference: 8.3.2.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/add_reference.hpp>` or `#include <boost/type_traits.hpp>`

Table 8. Examples

Expression	Result Type
`add_reference<int>::type`	`int&`
`add_reference<int const&>::type`	`int const&`
`add_reference<int*>::type`	`int*&`
`add_reference<int*&>::type`	`int*&`

add_volatile

```
template <class T>
struct add_volatile
{
   typedef see-below type;
};
```

type: The same type as `T volatile` for all `T`.

C++ Standard Reference: 3.9.3.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/add_volatile.hpp>` or `#include <boost/type_traits.hpp>`

Table 9. Examples

Expression	Result Type
`add_volatile<int>::type`	`int volatile`
`add_volatile<int&>::type`	`int&`
`add_volatile<int*>::type`	`int* volatile`
`add_volatile<int const>::type`	`int const volatile`

aligned_storage

```
template <std::size_t Size, std::size_t Align>
struct aligned_storage
{
   typedef see-below type;
};
```

type: a built-in or POD type with size `Size` and an alignment that is a multiple of `Align`.

Header: `#include <boost/type_traits/aligned_storage.hpp>` or `#include <boost/type_traits.hpp>`

alignment_of

```
template <class T>
struct alignment_of : public integral_constant<std::size_t, ALIGNOF(T)> {};
```

Inherits: Class template `alignment_of` inherits from `integral_constant<std::size_t, ALIGNOF(T)>`, where `ALIGNOF(T)` is the alignment of type T.

Note: strictly speaking you should only rely on the value of `ALIGNOF(T)` being a multiple of the true alignment of T, although in practice it does compute the correct value in all the cases we know about.

Header: `#include <boost/type_traits/alignment_of.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`alignment_of<int>` inherits from `integral_constant<std::size_t, ALIGNOF(int)>`.

`alignment_of<char>::type` is the type `integral_constant<std::size_t, ALIGNOF(char)>`.

`alignment_of<double>::value` is an integral constant expression with value `ALIGNOF(double)`.

`alignment_of<T>::value_type` is the type `std::size_t`.

decay

```
template <class T>
struct decay
{
   typedef see-below type;
};
```

type: Let `U` be the result of `remove_reference<T>::type`, then if `U` is an array type, the result is `remove_extent<U>::type*`, otherwise if `U` is a function type then the result is `U*`, otherwise the result is `U`.

C++ Standard Reference: 3.9.1.

Header: `#include <boost/type_traits/decay.hpp>` or `#include <boost/type_traits.hpp>`

Table 10. Examples

Expression	Result Type
`decay<int[2][3]>::type`	`int[3]*`
`decay<int(&)[2]>::type`	`int*`
`decay<int(&)(double)>::type`	`int(*)(double)`
`int(*)(double`	`int(*)(double)`
`int(double)`	`int(*)(double)`

extent

```
template <class T, std::size_t N = 0>
struct extent : public integral_constant<std::size_t, EXTENT(T,N)> {};
```

Inherits: Class template extent inherits from `integral_constant<std::size_t, EXTENT(T,N)>`, where `EXTENT(T,N)` is the number of elements in the N'th array dimention of type `T`.

If `T` is not an array type, or if `N > rank<T>::value`, or if the N'th array bound is incomplete, then `EXTENT(T,N)` is zero.

Header: `#include <boost/type_traits/extent.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`extent<int[1]>` inherits from `integral_constant<std::size_t, 1>`.

`extent<double[2][3][4], 1>::type` is the type `integral_constant<std::size_t, 3>`.

`extent<int[4]>::value` is an integral constant expression that evaluates to *4*.

`extent<int[][2]>::value` is an integral constant expression that evaluates to *0*.

`extent<int[][2], 1>::value` is an integral constant expression that evaluates to *2*.

`extent<int*>::value` is an integral constant expression that evaluates to *0*.

`extent<T>::value_type` is the type `std::size_t`.

floating_point_promotion

```
template <class T>
struct floating_point_promotion
{
   typedef see-below type;
};
```

type: If floating point promotion can be applied to an rvalue of type `T`, then applies floating point promotion to `T` and keeps cv-qualifiers of `T`, otherwise leaves `T` unchanged.

C++ Standard Reference: 4.6.

Header: `#include <boost/type_traits/floating_point_promotion.hpp>` or `#include <boost/type_traits.hpp>`

Table 11. Examples

Expression	Result Type
`floating_point_promotion<float const>::type`	`double const`
`floating_point_promotion<float&>::type`	`float&`
`floating_point_promotion<short>::type`	`short`

function_traits

```
template <class F>
struct function_traits
{
   static const std::size_t    arity = see-below;
   typedef see-below           result_type;
   typedef see-below           argN_type;
};
```

The class template function_traits will only compile if:

- The compiler supports partial specialization of class templates.
- The template argument `F` is a *function type*, note that this ***is not*** the same thing as a *pointer to a function*.

Tip

function_traits is intended to introspect only C++ functions of the form R (), R(A1), R (A1, ... etc.) and not function pointers or class member functions. To convert a function pointer type to a suitable type use remove_pointer.

Table 12. Function Traits Members

Member	Description
`function_traits<F>::arity`	An integral constant expression that gives the number of arguments accepted by the function type `F`.
`function_traits<F>::result_type`	The type returned by function type `F`.
`function_traits<F>::argN_type`	The *N*th argument type of function type `F`, where `1 <= N <= arity` of `F`.

Table 13. Examples

Expression	Result
`function_traits<void (void)>::arity`	An integral constant expression that has the value 0.
`function_traits<long (int)>::arity`	An integral constant expression that has the value 1.
`function_traits<long (int, long, double, void*)>::arity`	An integral constant expression that has the value 4.
`function_traits<void (void)>::result_type`	The type `void`.
`function_traits<long (int)>::result_type`	The type `long`.
`function_traits<long (int)>::arg1_type`	The type `int`.
`function_traits<long (int, long, double, void*)>::arg4_type`	The type `void*`.
`function_traits<long (int, long, double, void*)>::arg5_type`	A compiler error: there is no `arg5_type` since there are only four arguments.
`function_traits<long (*)(void)>::arity`	A compiler error: argument type is a *function pointer*, and not a *function type*.

has_new_operator

```
template <class T>
struct has_new_operator : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with an overloaded new-operator then inherits from true_type, otherwise inherits from false_type.

Compiler Compatibility: Not usable with compilers that do not support "substitution failure is not an error" (in which case BOOST_NO_SFINAE will be defined), also known to be broken with the Borland/Codegear compiler.

C++ Standard Reference: 12.5.

Header: `#include <boost/type_traits/has_new_operator.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

Given:

```
class A { void* operator new(std::size_t); };
class B { void* operator new(std::size_t, const std::nothrow&); };
class C { void* operator new(std::size_t, void*); };
class D { void* operator new[](std::size_t); };
class E { void* operator new[](std::size_t, const std::nothrow&); };
class F { void* operator new[](std::size_t, void*); };
```

Then:

`has_new_operator<A>` inherits from `true_type`.

`has_new_operator<B>` inherits from `true_type`.

`has_new_operator<C>` inherits from `true_type`.

`has_new_operator<D>` inherits from `true_type`.

`has_new_operator<E>` inherits from `true_type`.

`has_new_operator<F>` inherits from `true_type`.

has_nothrow_assign

```
template <class T>
struct has_nothrow_assign : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a non-throwing assignment-operator then inherits from true_type, otherwise inherits from false_type. Type `T` must be a complete type.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, `has_nothrow_assign` will never report that a class or struct has a non-throwing assignment-operator; this is always safe, if possibly sub-optimal. Currently (May 2005) only Visual C++ 8 has the necessary compiler support to ensure that this trait "just works".

Header: `#include <boost/type_traits/has_nothrow_assign.hpp>` or `#include <boost/type_traits.hpp>`

has_nothrow_constructor

```
template <class T>
struct has_nothrow_constructor : public true_type-or-false_type {};

template <class T>
struct has_nothrow_default_constructor : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a non-throwing default-constructor then inherits from true_type, otherwise inherits from false_type. Type `T` must be a complete type.

These two traits are synonyms for each other.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, `has_nothrow_constructor` will never report that a class or struct has a non-throwing default-constructor; this is always safe, if possibly sub-optimal. Currently (May 2005) only Visual C++ 8 has the necessary compiler intrinsics to ensure that this trait "just works".

Header: `#include <boost/type_traits/has_nothrow_constructor.hpp>` or `#include <boost/type_traits.hpp>`

has_nothrow_copy

```
template <class T>
struct has_nothrow_copy : public true_type-or-false_type {};

template <class T>
struct has_nothrow_copy_constructor : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a non-throwing copy-constructor then inherits from true_type, otherwise inherits from false_type. Type `T` must be a complete type.

These two traits are synonyms for each other.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, `has_nothrow_copy` will never report that a class or struct has a non-throwing copy-constructor; this is always safe, if possibly sub-optimal. Currently (May 2005) only Visual C++ 8 has the necessary compiler intrinsics to ensure that this trait "just works".

Header: `#include <boost/type_traits/has_nothrow_copy.hpp>` or `#include <boost/type_traits.hpp>`

has_nothrow_copy_constructor

See has_nothrow_copy.

has_nothrow_default_constructor

See has_nothrow_constructor.

has_trivial_assign

```
template <class T>
struct has_trivial_assign : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a trivial assignment-operator then inherits from true_type, otherwise inherits from false_type.

If a type has a trivial assignment-operator then the operator has the same effect as copying the bits of one object to the other: calls to the operator can be safely replaced with a call to `memcpy`.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, has_trivial_assign will never report that a user-defined class or struct has a trivial constructor; this is always safe, if possibly sub-optimal. Currently (May 2005) only MWCW 9 and Visual C++ 8 have the necessary compiler intrinsics to detect user-defined classes with trivial constructors.

C++ Standard Reference: 12.8p11.

Header: `#include <boost/type_traits/has_trivial_assign.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`has_trivial_assign<int>` inherits from `true_type`.

`has_trivial_assign<char*>::type` is the type `true_type`.

`has_trivial_assign<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`has_trivial_assign<MyClass>::value` is an integral constant expression that evaluates to *false*.

`has_trivial_assign<T>::value_type` is the type `bool`.

has_trivial_constructor

```
template <class T>
struct has_trivial_constructor : public true_type-or-false_type {};

template <class T>
struct has_trivial_default_constructor : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a trivial default-constructor then inherits from true_type, otherwise inherits from false_type.

These two traits are synonyms for each other.

If a type has a trivial default-constructor then the constructor have no effect: calls to the constructor can be safely omitted. Note that using meta-programming to omit a call to a single trivial-constructor call is of no benefit whatsoever. However, if loops and/or exception handling code can also be omitted, then some benefit in terms of code size and speed can be obtained.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, has_trivial_constructor will never report that a user-defined class or struct has a trivial constructor; this is always safe, if possibly sub-optimal. Currently (May 2005) only MWCW 9 and Visual C++ 8 have the necessary compiler intrinsics to detect user-defined classes with trivial constructors.

C++ Standard Reference: 12.1p6.

Header: `#include <boost/type_traits/has_trivial_constructor.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`has_trivial_constructor<int>` inherits from `true_type`.

`has_trivial_constructor<char*>::type` is the type `true_type`.

`has_trivial_constructor<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`has_trivial_constructor<MyClass>::value` is an integral constant expression that evaluates to *false*.

`has_trivial_constructor<T>::value_type` is the type `bool`.

has_trivial_copy

```
template <class T>
struct has_trivial_copy : public true_type-or-false_type {};

template <class T>
struct has_trivial_copy_constructor : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a trivial copy-constructor then inherits from true_type, otherwise inherits from false_type.

These two traits are synonyms for each other.

If a type has a trivial copy-constructor then the constructor has the same effect as copying the bits of one object to the other: calls to the constructor can be safely replaced with a call to `memcpy`.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, has_trivial_copy will never report that a user-defined class or struct has a trivial constructor; this is always safe, if possibly sub-optimal. Currently (May 2005) only MWCW 9 and Visual C++ 8 have the necessary compiler intrinsics to detect user-defined classes with trivial constructors.

C++ Standard Reference: 12.8p6.

Header: `#include <boost/type_traits/has_trivial_copy.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`has_trivial_copy<int>` inherits from `true_type`.

`has_trivial_copy<char*>::type` is the type `true_type`.

`has_trivial_copy<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`has_trivial_copy<MyClass>::value` is an integral constant expression that evaluates to *false*.

`has_trivial_copy<T>::value_type` is the type `bool`.

has_trivial_copy_constructor

See has_trivial_copy.

has_trivial_default_constructor

See has_trivial_constructor.

has_trivial_destructor

```
template <class T>
struct has_trivial_destructor : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a trivial destructor then inherits from true_type, otherwise inherits from false_type.

If a type has a trivial destructor then the destructor has no effect: calls to the destructor can be safely omitted. Note that using meta-programming to omit a call to a single trivial-constructor call is of no benefit whatsoever. However, if loops and/or exception handling code can also be omitted, then some benefit in terms of code size and speed can be obtained.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, has_trivial_destructor will never report that a user-defined class or struct has a trivial destructor; this is always safe, if possibly sub-optimal. Currently (May 2005) only MWCW 9 and Visual C++ 8 have the necessary compiler intrinsics to detect user-defined classes with trivial constructors.

C++ Standard Reference: 12.4p3.

Header: `#include <boost/type_traits/has_trivial_destructor.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`has_trivial_destructor<int>` inherits from `true_type`.

`has_trivial_destructor<char*>::type` is the type `true_type`.

`has_trivial_destructor<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`has_trivial_destructor<MyClass>::value` is an integral constant expression that evaluates to *false*.

`has_trivial_destructor<T>::value_type` is the type `bool`.

has_virtual_destructor

```
template <class T>
struct has_virtual_destructor : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) type with a virtual destructor then inherits from true_type, otherwise inherits from false_type.

Compiler Compatibility: This trait is provided for completeness, since it's part of the Technical Report on C++ Library Extensions. However, there is currently no way to portably implement this trait. The default version provided always inherits from false_type, and has to be explicitly specialized for types with virtual destructors unless the compiler used has compiler intrinsics that enable the trait to do the right thing: currently (May 2005) only Visual C++ 8 and GCC-4.3 have the necessary intrinsics.

C++ Standard Reference: 12.4.

Header: `#include <boost/type_traits/has_virtual_destructor.hpp>` or `#include <boost/type_traits.hpp>`

integral_constant

```
template <class T, T val>
struct integral_constant
{
   typedef integral_constant<T, val>  type;
   typedef T                          value_type;
   static const T value = val;
};

typedef integral_constant<bool, true>  true_type;
typedef integral_constant<bool, false> false_type;
```

Class template `integral_constant` is the common base class for all the value-based type traits. The two typedef's `true_type` and `false_type` are provided for convenience: most of the value traits are Boolean properties and so will inherit from one of these.

integral_promotion

```
template <class T>
struct integral_promotion
{
   typedef see-below type;
};
```

type: If integral promotion can be applied to an rvalue of type `T`, then applies integral promotion to `T` and keeps cv-qualifiers of `T`, otherwise leaves `T` unchanged.

C++ Standard Reference: 4.5 except 4.5/3 (integral bit-field).

Header: `#include <boost/type_traits/integral_promotion.hpp>` or `#include <boost/type_traits.hpp>`

Table 14. Examples

Expression	Result Type
`integral_promotion<short const>::type`	`int const`
`integral_promotion<short&>::type`	`short&`
`integral_promotion<enum std::float_round_style>::type`	`int`

is_abstract

```
template <class T>
struct is_abstract : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) abstract type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 10.3.

Header: `#include <boost/type_traits/is_abstract.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: The compiler must support DR337 (as of April 2005: GCC 3.4, VC++ 7.1 (and later), Intel C++ 7 (and later), and Comeau 4.3.2). Otherwise behaves the same as is_polymorphic; this is the "safe fallback position" for which polymorphic types are always regarded as potentially abstract. The macro BOOST_NO_IS_ABSTRACT is used to signify that the implementation is buggy, users should check for this in their own code if the "safe fallback" is not suitable for their particular use-case.

Examples:

Given: `class abc{ virtual ~abc() = 0; };`

`is_abstract<abc>` inherits from `true_type`.

`is_abstract<abc>::type` is the type `true_type`.

`is_abstract<abc const>::value` is an integral constant expression that evaluates to *true*.

`is_abstract<T>::value_type` is the type `bool`.

is_arithmetic

```
template <class T>
struct is_arithmetic : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) arithmetic type then inherits from true_type, otherwise inherits from false_type. Arithmetic types include integral and floating point types (see also is_integral and is_floating_point).

C++ Standard Reference: 3.9.1p8.

Header: `#include <boost/type_traits/is_arithmetic.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_arithmetic<int>` inherits from `true_type`.

`is_arithmetic<char>::type` is the type `true_type`.

`is_arithmetic<double>::value` is an integral constant expression that evaluates to *true*.

`is_arithmetic<T>::value_type` is the type `bool`.

is_array

```
template <class T>
struct is_array : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) array type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 8.3.4.

Header: `#include <boost/type_traits/is_array.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can give the wrong result with function types.

Examples:

`is_array<int[2]>` inherits from `true_type`.

`is_array<char[2][3]>::type` is the type `true_type`.

`is_array<double[]>::value` is an integral constant expression that evaluates to *true*.

`is_array<T>::value_type` is the type `bool`.

is_base_of

```
template <class Base, class Derived>
struct is_base_of : public true_type-or-false_type {};
```

Inherits: If Base is base class of type Derived or if both types are the same then inherits from true_type, otherwise inherits from false_type.

This template will detect non-public base classes, and ambiguous base classes.

Note that `is_base_of<X,X>` will always inherit from true_type. **This is the case even if `X` is not a class type**. This is a change in behaviour from Boost-1.33 in order to track the Technical Report on C++ Library Extensions.

Types `Base` and `Derived` must not be incomplete types.

C++ Standard Reference: 10.

Header: `#include <boost/type_traits/is_base_of.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types. There are some older compilers which will produce compiler errors if `Base` is a private base class of `Derived`, or if `Base` is an ambiguous base of `Derived`. These compilers include Borland C++, older versions of Sun Forte C++, Digital Mars C++, and older versions of EDG based compilers.

Examples:

Given: `class Base{}; class Derived : public Base{};`

`is_base_of<Base, Derived>` inherits from `true_type`.

`is_base_of<Base, Derived>::type` is the type `true_type`.

`is_base_of<Base, Derived>::value` is an integral constant expression that evaluates to *true*.

`is_base_of<Base, Base>::value` is an integral constant expression that evaluates to *true*: a class is regarded as it's own base.

`is_base_of<T, T>::value_type` is the type `bool`.

is_class

```
template <class T>
struct is_class : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) class type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 9.2.

Header: `#include <boost/type_traits/is_class.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: Without (some as yet unspecified) help from the compiler, we cannot distinguish between union and class types, as a result this type will erroneously inherit from true_type for union types. See also is_union. Currently (May 2005) only Visual C++ 8 has the necessary compiler intrinsics to correctly identify union types, and therefore make is_class function correctly.

Examples:

Given: `class MyClass;` then:

`is_class<MyClass>` inherits from `true_type`.

`is_class<MyClass const>::type` is the type `true_type`.

`is_class<MyClass>::value` is an integral constant expression that evaluates to *true*.

`is_class<MyClass&>::value` is an integral constant expression that evaluates to *false*.

`is_class<MyClass*>::value` is an integral constant expression that evaluates to *false*.

`is_class<T>::value_type` is the type `bool`.

is_complex

```
template <class T>
struct is_complex : public true_type-or-false_type {};
```

Inherits: If `T` is a complex number type then true (of type `std::complex<U>` for some type `U`), otherwise false.

C++ Standard Reference: 26.2.

Header: `#include <boost/type_traits/is_complex.hpp>` or `#include <boost/type_traits.hpp>`

is_compound

```
template <class T>
struct is_compound : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) compound type then inherits from true_type, otherwise inherits from false_type. Any type that is not a fundamental type is a compound type (see also is_fundamental).

C++ Standard Reference: 3.9.2.

Header: `#include <boost/type_traits/is_compound.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_compound<MyClass>` inherits from `true_type`.

`is_compound<MyEnum>::type` is the type `true_type`.

`is_compound<int*>::value` is an integral constant expression that evaluates to *true*.

`is_compound<int&>::value` is an integral constant expression that evaluates to *true*.

`is_compound<int>::value` is an integral constant expression that evaluates to *false*.

`is_compound<T>::value_type` is the type `bool`.

is_const

```
template <class T>
struct is_const : public true_type-or-false_type {};
```

Inherits: If T is a (top level) const-qualified type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.3.

Header: `#include <boost/type_traits/is_const.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_const<int const>` inherits from `true_type`.

`is_const<int const volatile>::type` is the type `true_type`.

`is_const<int* const>::value` is an integral constant expression that evaluates to *true*.

`is_const<int const*>::value` is an integral constant expression that evaluates to *false*: the const-qualifier is not at the top level in this case.

`is_const<int const&>::value` is an integral constant expression that evaluates to *false*: the const-qualifier is not at the top level in this case.

`is_const<int>::value` is an integral constant expression that evaluates to *false*.

`is_const<T>::value_type` is the type `bool`.

is_convertible

```
template <class From, class To>
struct is_convertible : public true_type-or-false_type {};
```

Inherits: If an imaginary lvalue of type `From` is convertible to type `To` then inherits from true_type, otherwise inherits from false_type.

Type From must not be an incomplete type.

Type To must not be an incomplete, or function type.

No types are considered to be convertible to array types or abstract-class types.

This template can not detect whether a converting-constructor is `public` or not: if type `To` has a `private` converting constructor from type `From` then instantiating `is_convertible<From, To>` will produce a compiler error. For this reason `is_convertible` can not be used to determine whether a type has a `public` copy-constructor or not.

This template will also produce compiler errors if the conversion is ambiguous, for example:

```
struct A {};
struct B : A {};
struct C : A {};
struct D : B, C {};
// This produces a compiler error, the conversion is ambiguous:
bool const y = boost::is_convertible<D*,A*>::value;
```

C++ Standard Reference: 4 and 8.5.

Compiler Compatibility: This template is currently broken with Borland C++ Builder 5 (and earlier), for constructor-based conversions, and for the Metrowerks 7 (and earlier) compiler in all cases. If the compiler does not support `is_abstract`, then the template parameter `To` must not be an abstract type.

Header: `#include <boost/type_traits/is_convertible.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_convertible<int, double>` inherits from `true_type`.

`is_convertible<const int, double>::type` is the type `true_type`.

`is_convertible<int* const, int*>::value` is an integral constant expression that evaluates to *true*.

`is_convertible<int const*, int*>::value` is an integral constant expression that evaluates to *false*: the conversion would require a `const_cast`.

`is_convertible<int const&, long>::value` is an integral constant expression that evaluates to *true*.

`is_convertible<int, int>::value` is an integral constant expression that evaluates to *false*.

`is_convertible<T, T>::value_type` is the type `bool`.

is_empty

```
template <class T>
struct is_empty : public true_type-or-false_type {};
```

Inherits: If T is an empty class type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 10p5.

Header: `#include <boost/type_traits/is_empty.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: In order to correctly detect empty classes this trait relies on either:

- the compiler implementing zero sized empty base classes, or
- the compiler providing intrinsics to detect empty classes.

Can not be used with incomplete types.

Can not be used with union types, until is_union can be made to work.

If the compiler does not support partial-specialization of class templates, then this template can not be used with abstract types.

Examples:

Given: `struct empty_class {};`

`is_empty<empty_class>` inherits from `true_type`.

`is_empty<empty_class const>::type` is the type `true_type`.

`is_empty<empty_class>::value` is an integral constant expression that evaluates to *true*.

`is_empty<T>::value_type` is the type `bool`.

is_enum

```
template <class T>
struct is_enum : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) enum type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 7.2.

Header: `#include <boost/type_traits/is_enum.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: Requires a correctly functioning is_convertible template; this means that is_enum is currently broken under Borland C++ Builder 5, and for the Metrowerks compiler prior to version 8, other compilers should handle this template just fine.

Examples:

Given: `enum my_enum { one, two };`

`is_enum<my_enum>` inherits from `true_type`.

`is_enum<my_enum const>::type` is the type `true_type`.

`is_enum<my_enum>::value` is an integral constant expression that evaluates to *true*.

`is_enum<my_enum&>::value` is an integral constant expression that evaluates to *false*.

`is_enum<my_enum*>::value` is an integral constant expression that evaluates to *false*.

`is_enum<T>::value_type` is the type `bool`.

is_floating_point

```
template <class T>
struct is_floating_point : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) floating point type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.1p8.

Header: `#include <boost/type_traits/is_floating_point.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_floating_point<float>` inherits from `true_type`.

`is_floating_point<double>::type` is the type `true_type`.

`is_floating_point<long double>::value` is an integral constant expression that evaluates to *true*.

`is_floating_point<T>::value_type` is the type `bool`.

is_function

```
template <class T>
struct is_function : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) function type then inherits from true_type, otherwise inherits from false_type. Note that this template does not detect *pointers to functions*, or *references to functions*, these are detected by is_pointer and is_reference respectively:

```
typedef int f1();      // f1 is of function type.
typedef int (f2*)();   // f2 is a pointer to a function.
typedef int (f3&)();   // f3 is a reference to a function.
```

C++ Standard Reference: 3.9.2p1 and 8.3.5.

Header: `#include <boost/type_traits/is_function.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_function<int (void)>` inherits from `true_type`.

`is_function<long (double, int)>::type` is the type `true_type`.

`is_function<long (double, int)>::value` is an integral constant expression that evaluates to *true*.

`is_function<long (*)(double, int)>::value` is an integral constant expression that evaluates to *false*: the argument in this case is a pointer type, not a function type.

`is_function<long (&)(double, int)>::value` is an integral constant expression that evaluates to *false*: the argument in this case is a reference to a function, not a function type.

`is_function<long (MyClass::*)(double, int)>::value` is an integral constant expression that evaluates to *false*: the argument in this case is a pointer to a member function.

`is_function<T>::value_type` is the type `bool`.

Tip

Don't confuse function-types with pointers to functions:

`typedef int f(double);`

defines a function type,

`f foo;`

declares a prototype for a function of type `f`,

`f* pf = foo;`

`f& fr = foo;`

declares a pointer and a reference to the function `foo`.

If you want to detect whether some type is a pointer-to-function then use:

`is_function<remove_pointer<T>::type>::value && is_pointer<T>::value`

or for pointers to member functions you can just use is_member_function_pointer directly.

is_fundamental

```
template <class T>
struct is_fundamental : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) fundamental type then inherits from true_type, otherwise inherits from false_type. Fundamental types include integral, floating point and void types (see also is_integral, is_floating_point and is_void)

C++ Standard Reference: 3.9.1.

Header: `#include <boost/type_traits/is_fundamental.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_fundamental<int)>` inherits from `true_type`.

`is_fundamental<double const>::type` is the type `true_type`.

`is_fundamental<void>::value` is an integral constant expression that evaluates to *true*.

`is_fundamental<T>::value_type` is the type `bool`.

is_integral

```
template <class T>
struct is_integral : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) integral type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.1p7.

Header: `#include <boost/type_traits/is_integral.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_integral<int>` inherits from `true_type`.

`is_integral<const char>::type` is the type `true_type`.

`is_integral<long>::value` is an integral constant expression that evaluates to *true*.

`is_integral<T>::value_type` is the type `bool`.

is_member_function_pointer

```
template <class T>
struct is_member_function_pointer : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) pointer to a member function then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 8.3.3.

Header: `#include <boost/type_traits/is_member_function_pointer.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_member_function_pointer<int (MyClass::*)(void)>` inherits from `true_type`.

`is_member_function_pointer<int (MyClass::*)(char)>::type` is the type `true_type`.

`is_member_function_pointer<int (MyClass::*)(void)const>::value` is an integral constant expression that evaluates to *true*.

`is_member_function_pointer<int (MyClass::*)>::value` is an integral constant expression that evaluates to *false*: the argument in this case is a pointer to a data member and not a member function, see is_member_object_pointer and is_member_pointer

`is_member_function_pointer<T>::value_type` is the type `bool`.

is_member_object_pointer

```
template <class T>
struct is_member_object_pointer : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) pointer to a member object (a data member) then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 8.3.3.

Header: `#include <boost/type_traits/is_member_object_pointer.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_member_object_pointer<int (MyClass::*)>` inherits from `true_type`.

`is_member_object_pointer<double (MyClass::*)>::type` is the type `true_type`.

`is_member_object_pointer<const int (MyClass::*)>::value` is an integral constant expression that evaluates to *true*.

`is_member_object_pointer<int (MyClass::*)(void)>::value` is an integral constant expression that evaluates to *false*: the argument in this case is a pointer to a member function and not a member object, see is_member_function_pointer and is_member_pointer

`is_member_object_pointer<T>::value_type` is the type `bool`.

is_member_pointer

```
template <class T>
struct is_member_pointer : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) pointer to a member (either a function or a data member) then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 8.3.3.

Header: `#include <boost/type_traits/is_member_pointer.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_member_pointer<int (MyClass::*)>` inherits from `true_type`.

`is_member_pointer<int (MyClass::*)(char)>::type` is the type `true_type`.

`is_member_pointer<int (MyClass::*)(void)const>::value` is an integral constant expression that evaluates to *true*.

`is_member_pointer<T>::value_type` is the type `bool`.

is_object

```
template <class T>
struct is_object : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) object type then inherits from true_type, otherwise inherits from false_type. All types are object types except references, void, and function types.

C++ Standard Reference: 3.9p9.

Header: `#include <boost/type_traits/is_object.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_object<int>` inherits from `true_type`.

`is_object<int*>::type` is the type `true_type`.

`is_object<int (*)(void)>::value` is an integral constant expression that evaluates to *true*.

`is_object<int (MyClass::*)(void)const>::value` is an integral constant expression that evaluates to *true*.

`is_object<int &>::value` is an integral constant expression that evaluates to *false*: reference types are not objects

`is_object<int (double)>::value` is an integral constant expression that evaluates to *false*: function types are not objects

`is_object<const void>::value` is an integral constant expression that evaluates to *false*: void is not an object type

`is_object<T>::value_type` is the type `bool`.

is_pod

```
template <class T>
struct is_pod : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) POD type then inherits from true_type, otherwise inherits from false_type.

POD stands for "Plain old data". Arithmetic types, and enumeration types, a pointers and pointer to members are all PODs. Classes and unions can also be POD's if they have no non-static data members that are of reference or non-POD type, no user defined constructors, no user defined assignment operators, no private or protected non-static data members, no virtual functions and no base classes. Finally, a cv-qualified POD is still a POD, as is an array of PODs.

C++ Standard Reference: 3.9p10 and 9p4 (Note that POD's are also aggregates, see 8.5.1).

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, ispod will never report that a class or struct is a POD; this is always safe, if possibly sub-optimal. Currently (May 2005) only MWCW 9 and Visual C++ 8 have the necessary compiler- intrinsics.

Header: `#include <boost/type_traits/is_pod.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_pod<int>` inherits from `true_type`.

`is_pod<char*>::type` is the type `true_type`.

`is_pod<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`is_pod<MyClass>::value` is an integral constant expression that evaluates to *false*.

`is_pod<T>::value_type` is the type `bool`.

is_pointer

```
template <class T>
struct is_pointer : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) pointer type (includes function pointers, but excludes pointers to members) then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2p2 and 8.3.1.

Header: `#include <boost/type_traits/is_pointer.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_pointer<int*>` inherits from `true_type`.

`is_pointer<char* const>::type` is the type `true_type`.

`is_pointer<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`is_pointer<int (MyClass::*)(long)>::value` is an integral constant expression that evaluates to *false*.

`is_pointer<int (MyClass::*)>::value` is an integral constant expression that evaluates to *false*.

`is_pointer<T>::value_type` is the type `bool`.

Important

`is_pointer` detects "real" pointer types only, and *not* smart pointers. Users should not specialise `is_pointer` for smart pointer types, as doing so may cause Boost (and other third party) code to fail to function correctly. Users wanting a trait to detect smart pointers should create their own. However, note that there is no way in general to auto-magically detect smart pointer types, so such a trait would have to be partially specialised for each supported smart pointer type.

is_polymorphic

```
template <class T>
struct is_polymorphic : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) polymorphic type then inherits from true_type, otherwise inherits from false_type. Type `T` must be a complete type.

C++ Standard Reference: 10.3.

Compiler Compatibility: The implementation requires some knowledge of the compilers ABI, it does actually seem to work with the majority of compilers though.

Header: `#include <boost/type_traits/is_polymorphic.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

Given: `class poly{ virtual ~poly(); };`

`is_polymorphic<poly>` inherits from `true_type`.

`is_polymorphic<poly const>::type` is the type `true_type`.

`is_polymorphic<poly>::value` is an integral constant expression that evaluates to *true*.

`is_polymorphic<T>::value_type` is the type `bool`.

is_same

```
template <class T, class U>
struct is_same : public true_type-or-false_type {};
```

Inherits: If T and U are the same types then inherits from true_type, otherwise inherits from false_type.

Header: `#include <boost/type_traits/is_same.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with abstract, incomplete or function types.

Examples:

`is_same<int, int>` inherits from `true_type`.

`is_same<int, int>::type` is the type `true_type`.

`is_same<int, int>::value` is an integral constant expression that evaluates to *true*.

`is_same<int const, int>::value` is an integral constant expression that evaluates to *false*.

`is_same<int&, int>::value` is an integral constant expression that evaluates to *false*.

`is_same<T, T>::value_type` is the type `bool`.

is_scalar

```
template <class T>
struct is_scalar : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) scalar type then inherits from true_type, otherwise inherits from false_type. Scalar types include integral, floating point, enumeration, pointer, and pointer-to-member types.

C++ Standard Reference: 3.9p10.

Header: `#include <boost/type_traits/is_scalar.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Examples:

`is_scalar<int*>` inherits from `true_type`.

`is_scalar<int>::type` is the type `true_type`.

`is_scalar<double>::value` is an integral constant expression that evaluates to *true*.

`is_scalar<int (*)(long)>::value` is an integral constant expression that evaluates to *true*.

`is_scalar<int (MyClass::*)(long)>::value` is an integral constant expression that evaluates to *true*.

`is_scalar<int (MyClass::*)>::value` is an integral constant expression that evaluates to *true*.

`is_scalar<T>::value_type` is the type `bool`.

is_signed

```
template <class T>
struct is_signed : public true_type-or-false_type {};
```

Inherits: If T is an signed integer type or an enumerated type with an underlying signed integer type, then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.1, 7.2.

Header: `#include <boost/type_traits/is_signed.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_signed<int>` inherits from `true_type`.

`is_signed<int const volatile>::type` is the type `true_type`.

`is_signed<unsigned int>::value` is an integral constant expression that evaluates to *false*.

`is_signed<myclass>::value` is an integral constant expression that evaluates to *false*.

`is_signed<char>::value` is an integral constant expression whose value depends upon the signedness of type `char`.

`is_signed<long long>::value` is an integral constant expression that evaluates to *true*.

`is_signed<T>::value_type` is the type `bool`.

is_stateless

```
template <class T>
struct is_stateless : public true_type-or-false_type {};
```

Inherits: Ff T is a stateless type then inherits from true_type, otherwise from false_type.

Type T must be a complete type.

A stateless type is a type that has no storage and whose constructors and destructors are trivial. That means that `is_stateless` only inherits from true_type if the following expression is `true`:

```
::boost::has_trivial_constructor<T>::value
&& ::boost::has_trivial_copy<T>::value
&& ::boost::has_trivial_destructor<T>::value
&& ::boost::is_class<T>::value
&& ::boost::is_empty<T>::value
```

C++ Standard Reference: 3.9p10.

Header: `#include <boost/type_traits/is_stateless.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template can not be used with function types.

Without some (as yet unspecified) help from the compiler, is_stateless will never report that a class or struct is stateless; this is always safe, if possibly sub-optimal. Currently (May 2005) only MWCW 9 and Visual C++ 8 have the necessary compiler intrinsics to make this template work automatically.

is_reference

```
template <class T>
struct is_reference : public true_type-or-false_type {};
```

Inherits: If T is a reference pointer type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.2 and 8.3.2.

Compiler Compatibility: If the compiler does not support partial-specialization of class templates, then this template may report the wrong result for function types, and for types that are both const and volatile qualified.

Header: `#include <boost/type_traits/is_reference.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_reference<int&>` inherits from `true_type`.

`is_reference<int const&>::type` is the type `true_type`.

`is_reference<int (&)(long)>::value` is an integral constant expression that evaluates to *true* (the argument in this case is a reference to a function).

`is_reference<T>::value_type` is the type `bool`.

is_union

```
template <class T>
struct is_union : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) union type then inherits from true_type, otherwise inherits from false_type. Currently requires some kind of compiler support, otherwise unions are identified as classes.

C++ Standard Reference: 3.9.2 and 9.5.

Compiler Compatibility: Without (some as yet unspecified) help from the compiler, we cannot distinguish between union and class types using only standard C++, as a result this type will never inherit from true_type, unless the user explicitly specializes the template for their user-defined union types, or unless the compiler supplies some unspecified intrinsic that implements this functionality. Currently (May 2005) only Visual C++ 8 has the necessary compiler intrinsics to make this trait "just work" without user intervention.

Header: `#include <boost/type_traits/is_union.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_union<void>` inherits from `true_type`.

`is_union<const void>::type` is the type `true_type`.

`is_union<void>::value` is an integral constant expression that evaluates to *true*.

`is_union<void*>::value` is an integral constant expression that evaluates to *false*.

`is_union<T>::value_type` is the type `bool`.

is_unsigned

```
template <class T>
struct is_unsigned : public true_type-or-false_type {};
```

Inherits: If T is an unsigned integer type or an enumerated type with an underlying unsigned integer type, then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.1, 7.2.

Header: `#include <boost/type_traits/is_unsigned.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_unsigned<unsigned int>` inherits from `true_type`.

`is_unsigned<unsigned int const volatile>::type` is the type `true_type`.

`is_unsigned<int>::value` is an integral constant expression that evaluates to *false*.

`is_unsigned<myclass>::value` is an integral constant expression that evaluates to *false*.

`is_unsigned<char>::value` is an integral constant expression whose value depends upon the signedness of type `char`.

`is_unsigned<unsigned long long>::value` is an integral constant expression that evaluates to *true*.

`is_unsigned<T>::value_type` is the type `bool`.

is_virtual_base_of

```
template <class Base, class Derived>
struct is_virtual_base_of : public true_type-or-false_type {};
```

Inherits: If Base is a virtual base class of type Derived then inherits from true_type, otherwise inherits from false_type.

Types `Base` and `Derived` must not be incomplete types.

C++ Standard Reference: 10.

Header: `#include <boost/type_traits/is_virtual_base_of.hpp>` or `#include <boost/type_traits.hpp>`

Compiler Compatibility: this trait also requires a working is_base_of trait.

Examples:

Given: `class Base{}; class Derived : public virtual Base{};`

`is_virtual_base_of<Base, Derived>` inherits from `true_type`.

`is_virtual_base_of<Base, Derived>::type` is the type `true_type`.

`is_virtual_base_of<Base, Derived>::value` is an integral constant expression that evaluates to *true*.

`is_virtual_base_of<Base, Derived>::value` is an integral constant expression that evaluates to *true*.

`is_virtual_base_of<T, U>::value_type` is the type `bool`.

is_void

```
template <class T>
struct is_void : public true_type-or-false_type {};
```

Inherits: If T is a (possibly cv-qualified) void type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.1p9.

Header: `#include <boost/type_traits/is_void.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_void<void>` inherits from `true_type`.

`is_void<const void>::type` is the type `true_type`.

`is_void<void>::value` is an integral constant expression that evaluates to *true*.

`is_void<void*>::value` is an integral constant expression that evaluates to *false*.

`is_void<T>::value_type` is the type `bool`.

is_volatile

```
template <class T>
struct is_volatile : public true_type-or-false_type {};
```

Inherits: If T is a (top level) volatile-qualified type then inherits from true_type, otherwise inherits from false_type.

C++ Standard Reference: 3.9.3.

Header: `#include <boost/type_traits/is_volatile.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`is_volatile<volatile int>` inherits from `true_type`.

`is_volatile<const volatile int>::type` is the type `true_type`.

`is_volatile<int* volatile>::value` is an integral constant expression that evaluates to *true*.

`is_volatile<int volatile*>::value` is an integral constant expression that evaluates to *false*: the volatile qualifier is not at the top level in this case.

`is_volatile<T>::value_type` is the type `bool`.

make_signed

```
template <class T>
struct make_signed
{
   typedef see-below type;
};
```

type: If T is a signed integer type then the same type as T, if T is an unsigned integer type then the corresponding signed type. Otherwise if T is an enumerated or character type (char or wchar_t) then a signed integer type with the same width as T.

If T has any cv-qualifiers then these are also present on the result type.

Requires: T must be an integer or enumerated type, and must not be the type bool.

C++ Standard Reference: 3.9.1.

Header: `#include <boost/type_traits/make_signed.hpp>` or `#include <boost/type_traits.hpp>`

Table 15. Examples

Expression	Result Type
`make_signed<int>::type`	`int`
`make_signed<unsigned int const>::type`	`int const`
`make_signed<const unsigned long long>::type`	`const long long`
`make_signed<my_enum>::type`	A signed integer type with the same width as the enum.
`make_signed<wchar_t>::type`	A signed integer type with the same width as wchar_t.

make_unsigned

```
template <class T>
struct make_unsigned
{
   typedef see-below type;
};
```

type: If T is a unsigned integer type then the same type as T, if T is an signed integer type then the corresponding unsigned type. Otherwise if T is an enumerated or character type (char or wchar_t) then an unsigned integer type with the same width as T.

If T has any cv-qualifiers then these are also present on the result type.

Requires: T must be an integer or enumerated type, and must not be the type bool.

C++ Standard Reference: 3.9.1.

Header: `#include <boost/type_traits/make_unsigned.hpp>` or `#include <boost/type_traits.hpp>`

Table 16. Examples

Expression	Result Type
`make_unsigned<int>::type`	`unsigned int`
`make_unsigned<unsigned int const>::type`	`unsigned int const`
`make_unsigned<const unsigned long long>::type`	`const unsigned long long`
`make_unsigned<my_enum>::type`	An unsigned integer type with the same width as the enum.
`make_unsigned<wchar_t>::type`	An unsigned integer type with the same width as wchar_t.

promote

```
template <class T>
struct promote
{
   typedef see-below type;
};
```

type: If integral or floating point promotion can be applied to an rvalue of type `T`, then applies integral and floating point promotions to `T` and keeps cv-qualifiers of `T`, otherwise leaves `T` unchanged. See also integral_promotion and floating_point_promotion.

C++ Standard Reference: 4.5 except 4.5/3 (integral bit-field) and 4.6.

Header: `#include <boost/type_traits/promote.hpp>` or `#include <boost/type_traits.hpp>`

Table 17. Examples

Expression	Result Type
`promote<short volatile>::type`	`int volatile`
`promote<float const>::type`	`double const`
`promote<short&>::type`	`short&`

rank

```
template <class T>
struct rank : public integral_constant<std::size_t, RANK(T)> {};
```

Inherits: Class template rank inherits from `integral_constant<std::size_t, RANK(T)>`, where `RANK(T)` is the number of array dimensions in type `T`.

If `T` is not an array type, then `RANK(T)` is zero.

Header: `#include <boost/type_traits/rank.hpp>` or `#include <boost/type_traits.hpp>`

Examples:

`rank<int[]>` inherits from `integral_constant<std::size_t, 1>`.

`rank<double[2][3][4]>::type` is the type `integral_constant<std::size_t, 3>`.

`rank<int[1]>::value` is an integral constant expression that evaluates to *1*.

`rank<int[][2]>::value` is an integral constant expression that evaluates to *2*.

`rank<int*>::value` is an integral constant expression that evaluates to *0*.

`rank<T>::value_type` is the type `std::size_t`.

remove_all_extents

```
template <class T>
struct remove_all_extents
{
   typedef see-below type;
};
```

type: If `T` is an array type, then removes all of the array bounds on `T`, otherwise leaves `T` unchanged.

C++ Standard Reference: 8.3.4.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_all_extents.hpp>` or `#include <boost/type_traits.hpp>`

Table 18. Examples

Expression	Result Type
`remove_all_extents<int>::type`	`int`
`remove_all_extents<int const[2]>::type`	`int const`
`remove_all_extents<int[][2]>::type`	`int`
`remove_all_extents<int[2][3][4]>::type`	`int`
`remove_all_extents<int const*>::type`	`int const*`

remove_const

```
template <class T>
struct remove_const
{
   typedef see-below type;
};
```

type: The same type as `T`, but with any *top level* const-qualifier removed.

C++ Standard Reference: 3.9.3.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_const.hpp>` or `#include <boost/type_traits.hpp>`

Table 19. Examples

Expression	Result Type
`remove_const<int>::type`	`int`
`remove_const<int const>::type`	`int`
`remove_const<int const volatile>::type`	`int volatile`
`remove_const<int const&>::type`	`int const&`
`remove_const<int const*>::type`	`int const*`

remove_cv

```
template <class T>
struct remove_cv
{
   typedef see-below type;
};
```

type: The same type as `T`, but with any *top level* cv-qualifiers removed.

C++ Standard Reference: 3.9.3.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_cv.hpp>` or `#include <boost/type_traits.hpp>`

Table 20. Examples

Expression	Result Type
`remove_cv<int>::type`	`int`
`remove_cv<int const>::type`	`int`
`remove_cv<int const volatile>::type`	`int`
`remove_cv<int const&>::type`	`int const&`
`remove_cv<int const*>::type`	`int const*`

remove_extent

```
template <class T>
struct remove_extent
{
   typedef see-below type;
};
```

type: If `T` is an array type, then removes the topmost array bound, otherwise leaves `T` unchanged.

C++ Standard Reference: 8.3.4.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_extent.hpp>` or `#include <boost/type_traits.hpp>`

Table 21. Examples

Expression	Result Type
`remove_extent<int>::type`	`int`
`remove_extent<int const[2]>::type`	`int const`
`remove_extent<int[2][4]>::type`	`int[4]`
`remove_extent<int[][2]>::type`	`int[2]`
`remove_extent<int const*>::type`	`int const*`

remove_pointer

```
template <class T>
struct remove_pointer
{
   typedef see-below type;
};
```

type: The same type as `T`, but with any pointer modifier removed.

C++ Standard Reference: 8.3.1.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_pointer.hpp>` or `#include <boost/type_traits.hpp>`

Table 22. Examples

Expression	Result Type
`remove_pointer<int>::type`	`int`
`remove_pointer<int const*>::type`	`int const`
`remove_pointer<int const**>::type`	`int const*`
`remove_pointer<int&>::type`	`int&`
`remove_pointer<int*&>::type`	`int*&`

remove_reference

```
template <class T>
struct remove_reference
{
   typedef see-below type;
};
```

type: The same type as `T`, but with any reference modifier removed.

C++ Standard Reference: 8.3.2.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_reference.hpp>` or `#include <boost/type_traits.hpp>`

Table 23. Examples

Expression	Result Type
`remove_reference<int>::type`	`int`
`remove_reference<int const&>::type`	`int const`
`remove_reference<int*>::type`	`int*`
`remove_reference<int*&>::type`	`int*`

remove_volatile

```
template <class T>
struct remove_volatile
{
   typedef see-below type;
};
```

type: The same type as `T`, but with any *top level* volatile-qualifier removed.

C++ Standard Reference: 3.9.3.

Compiler Compatibility: If the compiler does not support partial specialization of class-templates then this template will compile, but the member `type` will always be the same as type `T` except where compiler workarounds have been applied.

Header: `#include <boost/type_traits/remove_volatile.hpp>` or `#include <boost/type_traits.hpp>`

Table 24. Examples

Expression	Result Type
`remove_volatile<int>::type`	`int`
`remove_volatile<int volatile>::type`	`int`
`remove_volatile<int const volatile>::type`	`int const`
`remove_volatile<int volatile&>::type`	`int const&`
`remove_volatile<int volatile*>::type`	`int const*`

type_with_alignment

```
template <std::size_t Align>
struct type_with_alignment
{
   typedef see-below type;
};
```

type: a built-in or POD type with an alignment that is a multiple of `Align`.

Header: `#include <boost/type_traits/type_with_alignment.hpp>` or `#include <boost/type_traits.hpp>`

History

Boost 1.42.0

- Fixed issue #3704.

Credits

This documentation was pulled together by John Maddock, using Boost.Quickbook and Boost.DocBook.

The original version of this library was created by Steve Cleary, Beman Dawes, Howard Hinnant, and John Maddock. John Maddock is the current maintainer of the library.

This version of type traits library is based on contributions by Adobe Systems Inc, David Abrahams, Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, Itay Maman, John Maddock, Thorsten Ottosen, Robert Ramey and Jeremy Siek.

Mat Marcus and Jesse Jones invented, and published a paper describing, the partial specialization workarounds used in this library.

Aleksey Gurtovoy added MPL integration to the library.

The is_convertible template is based on code originally devised by Andrei Alexandrescu, see "Generic<Programming>: Mappings between Types and Values".

The latest version of this library and documentation can be found at www.boost.org. Bugs, suggestions and discussion should be directed to boost@lists.boost.org (see www.boost.org/more/mailing_lists.htm#main for subscription details).