

---

# Boost.TypeTraits

various authors

Copyright © 2000, 2011 Adobe Systems Inc, David Abrahams, Frederic Bron, Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, Itay Maman, John Maddock, Alexander Nasonov, Thorsten Ottosen, Roman Perepelitsa, Robert Ramey, Jeremy Siek, Robert Stewart and Steven Watanabe

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at [http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt))

## Table of Contents

Introduction .....	4
Background and Tutorial .....	5
Type Traits by Category .....	10
Type Traits that Describe the Properties of a Type .....	10
Categorizing a Type .....	10
General Type Properties .....	12
Relationships Between Two Types .....	14
Operator Type Traits .....	14
Type Traits that Transform One Type to Another .....	25
Synthesizing Types with Specific Alignments .....	27
Decomposing Function Types .....	27
User Defined Specializations .....	28
Support for Compiler Intrinsics .....	29
MPL Interoperability .....	31
Examples .....	32
An Optimized Version of std::copy .....	32
An Optimised Version of std::fill .....	32
An Example that Omits Destructor Calls For Types with Trivial Destructors .....	33
An improved Version of std::iter_swap .....	34
Convert Numeric Types and Enums to double .....	35
Improving std::min with common_type .....	35
Alphabetical Reference .....	37
add_const .....	37
add_cv .....	37
add_lvalue_reference .....	38
add_pointer .....	38
add_reference .....	39
add_rvalue_reference .....	40
add_volatile .....	40
aligned_storage .....	41
alignment_of .....	41
conditional .....	41
common_type .....	42
decay .....	45
extent .....	46
floating_point_promotion .....	47
function_traits .....	47
has_bit_and .....	48
has_bit_and_assign .....	50
has_bit_or .....	52
has_bit_or_assign .....	54
has_bit_xor .....	56

has_bit_xor_assign .....	58
has_complement .....	60
has_dereference .....	62
has_divides .....	64
has_divides_assign .....	66
has_equal_to .....	68
has_greater .....	70
has_greater_equal .....	72
has_left_shift .....	74
has_left_shift_assign .....	76
has_less .....	78
has_less_equal .....	80
has_logical_and .....	82
has_logical_not .....	84
has_logical_or .....	86
has_minus .....	88
has_minus_assign .....	90
has_modulus .....	92
has_modulus_assign .....	94
has_multiplies .....	96
has_multiplies_assign .....	98
has_negate .....	100
has_new_operator .....	102
has_not_equal_to .....	103
has_nothrow_assign .....	104
has_nothrow_constructor .....	105
has_nothrow_copy .....	105
has_nothrow_copy_constructor .....	106
has_nothrow_default_constructor .....	106
has_plus .....	106
has_plus_assign .....	108
has_post_decrement .....	109
has_post_increment .....	111
has_pre_decrement .....	113
has_pre_increment .....	115
has_right_shift .....	117
has_right_shift_assign .....	119
has_trivial_assign .....	121
has_trivial_constructor .....	122
has_trivial_copy .....	122
has_trivial_copy_constructor .....	123
has_trivial_default_constructor .....	123
has_trivial_destructor .....	123
has_unary_minus .....	124
has_unary_plus .....	126
has_virtual_destructor .....	127
integral_constant .....	128
integral_promotion .....	128
is_abstract .....	128
is_arithmetic .....	129
is_array .....	129
is_base_of .....	130
is_class .....	130
is_complex .....	131
is_compound .....	131
is_const .....	132
is_convertible .....	132
is_empty .....	133

is_enum .....	134
is_floating_point .....	134
is_function .....	134
is_fundamental .....	136
is_integral .....	136
is_lvalue_reference .....	136
is_member_function_pointer .....	137
is_member_object_pointer .....	137
is_member_pointer .....	138
is_object .....	138
is_pod .....	139
is_pointer .....	139
is_polymorphic .....	140
is_reference .....	140
is_rvalue_reference .....	141
is_same .....	141
is_scalar .....	142
is_signed .....	142
is_stateless .....	143
is_union .....	143
is_unsigned .....	144
is_virtual_base_of .....	144
is_void .....	145
is_volatile .....	145
make_signed .....	146
make_unsigned .....	146
promote .....	147
rank .....	147
remove_all_extents .....	148
remove_const .....	148
remove_cv .....	149
remove_extent .....	150
remove_pointer .....	150
remove_reference .....	151
remove_volatile .....	151
type_with_alignment .....	152
History .....	153
Credits .....	154
Class Index .....	154
Typedef Index .....	164
Macro Index .....	175
Index .....	185

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 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 `wcslon` 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 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<sup>[3]</sup> 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 `do_copy` 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 `do_copy`, 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 <code>T1</code>	Type of parameter to initializing constructor
<code>T</code>	<code>const T &amp;</code>
<code>T &amp;</code>	<code>T &amp;</code>
<code>const T &amp;</code>	<code>const T &amp;</code>

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 <code>T1</code>	Type of <code>const T1</code>	Type of <code>add_reference&lt;const T1&gt;::type</code>
<code>T</code>	<code>const T</code>	<code>const T &amp;</code>
<code>T &amp;</code>	<code>T &amp; [8]</code>	<code>T &amp;</code>
<code>const T &amp;</code>	<code>const T &amp;</code>	<code>const T &amp;</code>

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++ users 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](http://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_lvalue_reference;

template <class T>
struct is_rvalue_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_reference;

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;
```

## Operator Type Traits

### Introduction

These traits are all *value traits* inheriting from [integral\\_constant](#) and providing a simple true or false boolean value which reflects the fact that given types can or cannot be used with given operators.

For example, `has_plus<int, double>::value` is a `bool` which value is `true` because it is possible to add a `double` to an `int` like in the following code:

```
int i;
double d;
i+d;
```

It is also possible to know if the result of the operator can be used as function argument of a given type. For example, `has_plus<int, double, float>::value` is `true` because it is possible to add a `double` to an `int` and the result (`double`) can be converted to a `float` argument like in the following code:

```
void f(float) { };
int i;
double d;
f(i+d);
```

### Example of application

These traits can be useful to optimize the code for types supporting given operations. For example a function `std::advance` that increases an iterator of a given number of steps could be implemented as follows:

```

#include <boost/type_traits/has_plus_assign.hpp>

namespace detail {
template < class Iterator, class Distance, bool has_plus_assign >
struct advance_impl;

// this is used if += exists (efficient)
template < class Iterator, class Distance >
struct advance_impl<Iterator, Distance, true> {
    void operator()(Iterator &i, Distance n) {
        i+=n;
    }
};

// this is use if += does not exists (less efficient but cannot do better)
template < class Iterator, class Distance >
struct advance_impl<Iterator, Distance, false> {
    void operator()(Iterator &i, Distance n) {
        if (n>0) {
            while (n-- > 0) ++i;
        } else {
            while (n++ < 0) --i;
        }
    }
};
} // namespace detail

template < class Iterator, class Distance >
inline void advance(Iterator &i, Distance n) {
    detail::advance_impl< Iterator, Distance, ::boost::has_plus_assign<Iterator>::value >()(i, n);
}

```

Then the compiler chooses the most efficient implementation according to the type's ability to perform += operation:

```

#include <iostream>

class with {
    int m_i;
public:
    with(int i=0) : m_i(i) { }
    with &operator+=(int rhs) { m_i+=rhs; return *this; }
    operator int const () { return m_i; }
};

class without {
    int m_i;
public:
    without(int i=0) : m_i(i) { }
    without &operator++() { ++m_i; return *this; }
    without &operator--() { --m_i; return *this; }
    operator int const () { return m_i; }
};

int main() {
    with i=0;
    advance(i, 10); // uses +=
    std::cout<<"with: "<<i<<"\n";
    without j=0;
    advance(j, 10); // uses ++
    std::cout<<"without: "<<j<<"\n";
    return 0;
}

```

## Description

The syntax is the following:

```
template < class Rhs, class Ret=dont_care > has_op; // prefix operator
template < class Lhs, class Ret=dont_care > has_op; // postfix operator
template < class Lhs, class Rhs=Lhs, class Ret=dont_care > has_op; // binary operator
```

where:

- `op` represents the operator name
- `Lhs` is the type used at the left hand side of operator `op`,
- `Rhs` is the type used at the right hand side of operator `op`,
- `Ret` is the type for which we want to know if the result of operator `op` can be converted to.

The default behaviour (`Ret=dont_care`) is to not check for the return value of the operator. If `Ret` is different from the default `dont_care`, the return value is checked to be convertible to `Ret`. Convertible to `Ret` means that the return value can be used as argument to a function expecting `Ret`:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs+rhs); // is valid if has_plus<Lhs, Rhs, Ret>::value==true
```

If `Ret=void`, the return type is checked to be exactly `void`.

The following tables give the list of supported binary, prefix and postfix operators.

**Table 4. Supported prefix operators**

prefix operator	trait name
!	<code>has_logical_not</code> < class Rhs, class Ret=dont_care >
+	<code>has_unary_plus</code>
-	<code>has_unary_minus</code> and <code>has_negate</code>
~	<code>has_complement</code>
*	<code>has_dereference</code>
++	<code>has_pre_increment</code>
--	<code>has_pre_decrement</code>

**Table 5. Supported postfix operators**

postfix operator	trait name
++	<code>has_post_increment</code> < class Lhs, class Ret=dont_care >
--	<code>has_post_decrement</code>

**Table 6. Supported binary operators**

binary operator	trait name
+	<code>has_plus &lt; class Lhs, class Rhs=Lhs, class Ret=dont_care &gt;</code>
-	<code>has_minus</code>
*	<code>has_multiplies</code>
/	<code>has_divides</code>
%	<code>has_modulus</code>
+=	<code>has_plus_assign</code>
-=	<code>has_minus_assign</code>
*=	<code>has_multiplies_assign</code>
/=	<code>has_divides_assign</code>
%=	<code>has_modulus_assign</code>
&	<code>has_bit_and</code>
	<code>has_bit_or</code>
^	<code>has_bit_xor</code>
&=	<code>has_bit_and_assign</code>
=	<code>has_bit_or_assign</code>
^=	<code>has_bit_xor_assign</code>
<<	<code>has_left_shift</code>
>>	<code>has_right_shift</code>
<<=	<code>has_left_shift_assign</code>
>>=	<code>has_right_shift_assign</code>
==	<code>has_equal_to</code>
!=	<code>has_not_equal_to</code>
<	<code>has_less</code>
<=	<code>has_less_equal</code>
>	<code>has_greater</code>
>=	<code>has_greater_equal</code>
&&	<code>has_logical_and</code>

binary operator	trait name
	<code>has_logical_or</code>

The following operators are not supported because they could not be implemented using the same technique: `operator=`, `operator->`, `operator&`, `operator[]`, `operator,`, `operator()`, `operator new`.

### cv qualifiers and references

A reference sign `&` in the operator argument is ignored so that `has_plus< int&, double& >::value==has_plus< int, double >::value`. This has been chosen because if the following code works (does not work):

```
int i;
double d;
i+d;
```

the following code also works (does not work):

```
int &ir=i;
double &dr=d;
ir+dr;
```

It was not possible to handle properly the `volatile` qualifier so that any construct using this qualifier has undefined behavior.

As a help, the following tables give the necessary conditions over each trait template argument for the trait value to be true. They are non sufficient conditions because the conditions must be true for all arguments and return type for value to be true.

**Table 7. necessary and non sufficient condition on operator argument for value to be true**

operator declaration	<code>has_op&lt; void &gt;</code>	<code>has_op&lt; Arg &gt;</code> and <code>has_op&lt; Arg&amp; &gt;</code>	<code>has_op&lt; Arg const &gt;</code> and <code>has_op&lt; Arg const&amp; &gt;</code>
<code>operator@(Arg)</code>	false	true	true
<code>operator@(Arg const)</code>	false	true	true
<code>operator@(Arg &amp;)</code>	false	true	false
<code>operator@(Arg const &amp;)</code>	false	true	true

**Table 8. necessary and non sufficient condition on operator return type for value to be true**

operator declaration	<code>has_op&lt; ..., void &gt;</code>	<code>has_op&lt; ..., Ret &gt;</code>	<code>has_op&lt; ..., Ret const &gt;</code>	<code>has_op&lt; ..., Ret &amp; &gt;</code>	<code>has_op&lt; ..., Ret const &amp; &gt;</code>
<code>void operator@(...)</code>	true	false	false	false	false
<code>Ret operator@(...)</code>	false	true	true	false	true
<code>Ret const operator@(...)</code>	false	true	true	false	true
<code>Ret &amp; operator@(...)</code>	false	true	true	true	true
<code>Ret const &amp; operator@(...)</code>	false	true	true	false	true

### Implementation

The implementation consists in only header files. The following headers should included first:

```
#include <boost/type_traits/has_operator.hpp>
```

or

```
#include <boost/type_traits/has_op.hpp>
```

where `op` is the textual name chosen for the wanted operator. The first method includes all operator traits.

All traits are implemented the same way using preprocessor macros to avoid code duplication. The main files are in `boost/type_traits/detail/has_binary_operator.hpp`, `has_prefix_operator.hpp` and `has_postfix_operator.hpp`. The example of prefix operator- is presented below:

```

namespace boost {
namespace detail {

// This namespace ensures that argument-dependent name lookup does not mess things up.
namespace has_unary_minus_impl {

// 1. a function to have an instance of type T without requiring T to be default
// constructible
template <typename T> T &make();

// 2. we provide our operator definition for types that do not have one already

// a type returned from operator- when no such operator is
// found in the type's own namespace (our own operator is used) so that we have
// a means to know that our operator was used
struct no_operator { };

// this class allows implicit conversions and makes the following operator
// definition less-preferred than any other such operators that might be found
// via argument-dependent name lookup
struct any { template <class T> any(T const&); };

// when operator- is not available, this one is used
no_operator operator-(const any&);

// 3. checks if the operator returns void or not
// conditions: Rhs!=void

// we first redefine "operator," so that we have no compilation error if
// operator- returns void and we can use the return type of
// (-rhs, returns_void_t()) to deduce if operator- returns void or not:
// - operator- returns void    -> (-rhs, returns_void_t()) returns returns_void_t
// - operator- returns !=void -> (-rhs, returns_void_t()) returns int
struct returns_void_t { };
template <typename T> int operator,(const T&, returns_void_t);
template <typename T> int operator,(const volatile T&, returns_void_t);

// this intermediate trait has member value of type bool:
// - value==true  -> operator- returns void
// - value==false -> operator- does not return void
template < typename Rhs >
struct operator_returns_void {
    // overloads of function returns_void make the difference
    // yes_type and no_type have different size by construction
    static ::boost::type_traits::yes_type returns_void(returns_void_t);
    static ::boost::type_traits::no_type returns_void(int);
    static const bool value = sizeof(::boost::type_traits::yes_type)==sizeof(re-
turns_void((-make<Rhs>()),returns_void_t())));
};

// 4. checks if the return type is Ret or Ret==dont_care
// conditions: Rhs!=void

struct dont_care { };

template < typename Rhs, typename Ret, bool Returns_void >
struct operator_returns_Ret;

template < typename Rhs >
struct operator_returns_Ret < Rhs, dont_care, true > {

```

```

    static const bool value = true;
};

template < typename Rhs >
struct operator_returns_Ret < Rhs, dont_care, false > {
    static const bool value = true;
};

template < typename Rhs >
struct operator_returns_Ret < Rhs, void, true > {
    static const bool value = true;
};

template < typename Rhs >
struct operator_returns_Ret < Rhs, void, false > {
    static const bool value = false;
};

template < typename Rhs, typename Ret >
struct operator_returns_Ret < Rhs, Ret, true > {
    static const bool value = false;
};

// otherwise checks if it is convertible to Ret using the sizeof trick
// based on overload resolution
// condition: Ret!=void and Ret!=dont_care and the operator does not return void
template < typename Rhs, typename Ret >
struct operator_returns_Ret < Rhs, Ret, false > {
    static ::boost::type_traits::yes_type is_convertible_to_Ret(Ret); // this version is preferred ↓
    for types convertible to Ret
    static ::boost::type_traits::no_type is_convertible_to_Ret(...); // this version is used oth↓
    erwise

    static const bool value = sizeof(is_convertible_to_Ret(-make<Rhs>()))==sizeof(::boost::type_traits::yes_type);
};

// 5. checks for operator existence
// condition: Rhs!=void

// checks if our definition of operator- is used or an other
// existing one;
// this is done with redefinition of "operator," that returns no_operator or has_operator
struct has_operator { };
no_operator operator,(no_operator, has_operator);

template < typename Rhs >
struct operator_exists {
    static ::boost::type_traits::yes_type check(has_operator); // this version is preferred when ↓
    operator exists
    static ::boost::type_traits::no_type check(no_operator); // this version is used otherwise

    static const bool value = sizeof(check(((~make<Rhs>()),make<has_operat↓
    or>()))==sizeof(::boost::type_traits::yes_type);
};

// 6. main trait: to avoid any compilation error, this class behaves
// differently when operator-(Rhs) is forbidden by the standard.
// Forbidden_if is a bool that is:
// - true when the operator-(Rhs) is forbidden by the standard
// (would yield compilation error if used)

```

```

// - false otherwise
template < typename Rhs, typename Ret, bool Forbidden_if >
struct trait_impl1;

template < typename Rhs, typename Ret >
struct trait_impl1 < Rhs, Ret, true > {
    static const bool value = false;
};

template < typename Rhs, typename Ret >
struct trait_impl1 < Rhs, Ret, false > {
    static const bool value =
        ::boost::type_traits::ice_and<
            operator_exists < Rhs >::value,
            operator_returns_Ret < Rhs, Ret, operator_returns_void < Rhs >::value >::value
        >::value
    ;
};

// specialization needs to be declared for the special void case
template < typename Ret >
struct trait_impl1 < void, Ret, false > {
    static const bool value = false;
};

// defines some typedef for convenience
template < typename Rhs, typename Ret >
struct trait_impl {
    typedef typename ::boost::remove_reference<Rhs>::type Rhs_noref;
    typedef typename ::boost::remove_cv<Rhs_noref>::type Rhs_nocv;
    typedef typename ::boost::remove_cv< typename ::boost::remove_reference< typename ::boost::remove_pointer<Rhs_noref>::type >::type >::type Rhs_noptr;
    static const bool value = trait_impl1 < Rhs_noref, Ret, ::boost::is_pointer<Rhs_noref>::value >::value;
};

} // namespace impl
} // namespace detail

// this is the accessible definition of the trait to end user
template < typename Rhs, typename Ret=::boost::detail::has_unary_minus_impl::dont_care >
struct has_unary_minus : ::boost::integral_constant<bool, (::boost::detail::has_unary_minus_impl::trait_impl < Rhs, Ret >::value)> { };

} // namespace boost

```

## Limitation

- Requires a compiler with working SFINAE.

## Known issues

- These traits cannot detect whether the operators are public or not: if an operator is defined as a private member of type T then the instantiation of the corresponding trait will produce a compiler error. For this reason these traits cannot be used to determine whether a type has a public operator or not.

```

struct A { private: A operator-(); };
boost::has_unary_minus<A>::value; // error: A::operator-() is private

```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload because both the existing operator and the one we provide (with argument of type any) need type conversion, so that none is preferred.

```
struct A { };
void operator-(const A&);
struct B { operator A(); };
boost::has_unary_minus<A>::value; // this is fine
boost::has_unary_minus<B>::value; // error: ambiguous overload between
                                   // operator-(const any&) and
                                   // operator-(const A&)
                                   // both need type conversion
```

```
struct B { };
struct A { A(const B&) { } };
void operator-(const A&);
boost::has_unary_minus<A>::value; // this is fine
boost::has_unary_minus<B>::value; // error: ambiguous overload between
                                   // operator-(const any&) and
                                   // operator-(const A&)
                                   // both need type conversion
```

- There is an issue when applying these traits to template classes. If the operator is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. This applies in particular to the containers of the standard library and operator==. Example:

```
#include <boost/type_traits/has_equal_to.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator==(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_equal_to< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g==g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_equal_to< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b==b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## Acknowledgments

Frederic Bron is very thankful to numerous people from the boost mailing list for their kind help and patience. In particular, the following persons have been very helpful for the implementation: Edward Diener, Eric Niebler, Jeffrey Lee Hellrung (Jr.), Robert Stewart, Roman Perepelitsa, Steven Watanabe, Vicente Botet.

## 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_lvalue_reference;

template <class T>
struct add_pointer;

template <class T>
struct add_reference;

template <class T>
struct add_rvalue_reference;

template <class T>
struct add_volatile;

template <bool B, class T, class U>
struct conditional;

template <class... T>
struct common_type;

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 volatile&>::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_referen
ce<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

Occasionally 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 Intrinsic

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 9. Macros for Compiler Ininsics**

<b>BOOST_IS_UNION(T)</b>	<b>Should evaluate to true if T is a union type</b>
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&)
{
    memmove(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 destructors need to be called (see [trivial\\_destructor\\_example.cpp](#)):

```
//
// algorithm destroy_array:
// The reverse of std::uninitialized_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 if both arguments are non-proxying iterators,
    // and if both iterators 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);
}
```

## Improving std::min with common\_type

An improved `std::min` function could be written like this:

```
template <class T, class U>
typename common_type<T, U>::type min(T t, T u)
{
    return t < u ? t : u;
}
```

And now expressions such as:

```
min(1, 2.0)
```

will actually compile and return the correct type!

# 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 10. Examples**

Expression	Result Type
<code>add_const&lt;int&gt;::type</code>	<code>int const</code>
<code>add_const&lt;int&amp;&gt;::type</code>	<code>int&amp;</code>
<code>add_const&lt;int*&gt;::type</code>	<code>int* const</code>
<code>add_const&lt;int const&gt;::type</code>	<code>int const</code>

## 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 11. Examples**

Expression	Result Type
<code>add_cv&lt;int&gt;::type</code>	<code>int const volatile</code>
<code>add_cv&lt;int&amp;&gt;::type</code>	<code>int&amp;</code>
<code>add_cv&lt;int*&gt;::type</code>	<code>int* const volatile</code>
<code>add_cv&lt;int const&gt;::type</code>	<code>int const volatile</code>

## add\_lvalue\_reference

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

**type:** If `T` names an object or function type then the member typedef `type` shall name `T&`; otherwise, if `T` names a type *rvalue reference* to `U` then the member typedef `type` shall name `U&`; otherwise, `type` shall name `T`.

**C++ Standard Reference:** 20.7.6.2.

**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_lvalue_reference.hpp>` or `#include <boost/type_traits.hpp>`

**Table 12. Examples**

Expression	Result Type
<code>add_lvalue_reference&lt;int&gt;::type</code>	<code>int&amp;</code>
<code>add_lvalue_reference&lt;int const&amp;&gt;::type</code>	<code>int const&amp;</code>
<code>add_lvalue_reference&lt;int*&gt;::type</code>	<code>int*&amp;</code>
<code>add_lvalue_reference&lt;int*&amp;&gt;::type</code>	<code>int*&amp;</code>
<code>add_lvalue_reference&lt;int&amp;&amp;&gt;::type</code>	<code>int&amp;</code>
<code>add_lvalue_reference&lt;void&gt;::type</code>	<code>void</code>

## 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 13. Examples**

Expression	Result Type
<code>add_pointer&lt;int&gt;::type</code>	<code>int*</code>
<code>add_pointer&lt;int const&amp;&gt;::type</code>	<code>int const*</code>
<code>add_pointer&lt;int*&gt;::type</code>	<code>int**</code>
<code>add_pointer&lt;int*&amp;&gt;::type</code>	<code>int**</code>

## add\_reference



### Note

This trait has been made obsolete by [add\\_lvalue\\_reference](#) and [add\\_rvalue\\_reference](#), and new code should use these new traits rather than [is\\_reference](#) which is retained for backwards compatibility only.

```
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 14. Examples**

Expression	Result Type
<code>add_reference&lt;int&gt;::type</code>	<code>int&amp;</code>
<code>add_reference&lt;int const&amp;&gt;::type</code>	<code>int const&amp;</code>
<code>add_reference&lt;int*&gt;::type</code>	<code>int*&amp;</code>
<code>add_reference&lt;int*&amp;&gt;::type</code>	<code>int*&amp;</code>

## add\_rvalue\_reference

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

**type:** If T names an object or function type then the member typedef type shall name T&&; otherwise, type shall name T. *[Note: This rule reflects the semantics of reference collapsing. For example, when a type T names a type U&, the type add\_rvalue\_reference<T>::type is not an rvalue reference. -end note].*

**C++ Standard Reference:** 20.7.6.2.

**Compiler Compatibility:** If the compiler does not support partial specialization of class-templates and rvalue references then this template will compile, but the member type will always be the same as type T.

**Header:** #include <boost/type\_traits/add\_rvalue\_reference.hpp> or #include <boost/type\_traits.hpp>

**Table 15. Examples**

Expression	Result Type
add_rvalue_reference<int>::type	int&&
add_rvalue_reference<int const>::type	int const&
add_rvalue_reference<int*>::type	int*&&
add_rvalue_reference<int*&>::type	int*&
add_rvalue_reference<int&&>::type	int&&
add_rvalue_reference<void>::type	void

## 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 16. Examples**

Expression	Result Type
<code>add_volatile&lt;int&gt;::type</code>	<code>int volatile</code>
<code>add_volatile&lt;int&amp;&gt;::type</code>	<code>int&amp;</code>
<code>add_volatile&lt;int*&gt;::type</code>	<code>int* volatile</code>
<code>add_volatile&lt;int const&gt;::type</code>	<code>int const volatile</code>

## 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`.

## conditional

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

```
namespace boost {
    template <bool B, class T, class U> struct conditional;
}
```

If `B` is true, the member typedef type shall equal `T`. If `B` is false, the member typedef type shall equal `U`.

This trait is really just an alias for `boost::mpl::if_c`.

## common\_type

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

```
namespace boost {  
    template <class ...T> struct common_type;  
}
```

`common_type` is a traits class used to deduce a type common to a several types, useful as the return type of functions operating on multiple input types such as in mixed-mode arithmetic..

The nested typedef `::type` could be defined as follows:

```
template <class ...T>  
struct common_type;  
  
template <class T, class U, class ...V>  
struct common_type<T,U,...V> {  
    typedef typename common_type<typename common_type<T, U>::type, V...>::type type;  
};  
  
template <class T>  
struct common_type<T> {  
    typedef T type;  
};  
  
template <class T, class U>  
struct common_type<T, U> {  
    typedef decltype(declval<bool>() ? declval<T>() : declval<U>()) type;  
};
```

All parameter types must be complete. This trait is permitted to be specialized by a user if at least one template parameter is a user-defined type. **Note:** Such specializations are required when only explicit conversions are desired among the `common_type` arguments.

Note that when the compiler does not support variadic templates (and the macro `BOOST_NO_VARIADIC_TEMPLATES` is defined) then the maximum number of template arguments is 3.

## Configuration macros

When the compiler does not support static assertions then the user can select the way static assertions are reported. Define

- `BOOST_COMMON_TYPE_USES_STATIC_ASSERT`: define it if you want to use `Boost.StaticAssert`
- `BOOST_COMMON_TYPE_USES_MPL_ASSERT`: define it if you want to use `Boost.MPL` static assertions

The default behavior is to use `mpl` assertions in this case, but setting `BOOST_COMMON_TYPE_USES_STATIC_ASSERT` may reduce compile times and header dependencies somewhat.

Depending on the static assertion used you will have an hint of the failing assertion either through the symbol or through the text.

When possible `common_type` is implemented using `decltype`. Otherwise when `BOOST_COMMON_TYPE_DONT_USE_TYPEOF` is not defined it uses `Boost.TypeOf`.

## Tutorial

In a nutshell, `common_type` is a trait that takes 1 or more types, and returns a type which all of the types will convert to. The default definition demands this conversion be implicit. However the trait can be specialized for user-defined types which want to limit their inter-type conversions to explicit, and yet still want to interoperate with the `common_type` facility.

### Example:

```
template <class T, class U>
complex<typename common_type<T, U>::type>
operator+(complex<T>, complex<U>);
```

In the above example, "mixed-mode" complex arithmetic is allowed. The return type is described by `common_type`. For example the resulting type of adding a `complex<float>` and `complex<double>` might be a `complex<double>`.

Here is how someone might produce a variadic comparison function:

```
template <class ...T>
typename common_type<T...>::type
min(T... t);
```

This is a very useful and broadly applicable utility.

## How to get the common type of types with explicit conversions?

Another choice for the author of the preceding operator could be

```
template <class T, class U>
typename common_type<complex<T>, complex<U> >::type
operator+(complex<T>, complex<U>);
```

As the default definition of `common_type` demands the conversion be implicit, we need to specialize the trait for complex types as follows.

```
template <class T, class U>
struct common_type<complex<T>, complex<U> > {
    typedef complex< common_type<T, U> > type;
};
```

## How important is the order of the `common_type<>` template arguments?

The order of the template parameters is important.

`common_type<A,B,C>::type` is not equivalent to `common_type<C,A,B>::type`, but to `common_type<common_type<A,B>::type, C>::type`.

Consider

```
struct A {};  
struct B {};  
struct C {  
    C() {}  
    C(A const&) {}  
    C(B const&) {}  
    C& operator=(C const&) {  
        return *this;  
    }  
};
```

The following doesn't compile

```
typedef boost::common_type<A, B, C>::type ABC; // Does not compile
```

while

```
typedef boost::common_type<C, A, B>::type ABC;
```

compiles.

Thus, as `common_type<A,B>::type` is undefined, `common_type<A,B,C>::type` is also undefined.

It is intended that clients who wish for `common_type<A, B>` to be well defined to define it themselves:

```
namespace boost  
{  
  
    template <>  
    struct common_type<A, B> {typedef C type;};  
  
}
```

Now this client can ask for `common_type<A, B, C>` (and get the same answer).

Clients wanting to ask `common_type<A, B, C>` in any order and get the same result need to add in addition:

```
namespace boost  
{  
  
    template <> struct common_type<B, A>  
    : public common_type<A, B> {};  
  
}
```

This is needed as the specialization of `common_type<A, B>` is not be used implicitly for `common_type<B, A>`.

### Can the common\_type of two types be a third type?

Given the preceding example, one might expect `common_type<A,B>::type` to be `C` without any intervention from the user. But the default `common_type<>` implementation doesn't grant that. It is intended that clients who wish for `common_type<A, B>` to be well defined to define it themselves:

```
namespace boost
{

template <>
struct common_type<A, B> {typedef C type;};

template <> struct common_type<B, A>
: public common_type<A, B> {};

}
```

Now this client can ask for `common_type<A, B>`.

## How `common_type` behaves with pointers?

Consider

```
struct C { };
struct B : C { };
struct A : C { };
```

Shouldn't `common_type<A*, B*>::type` be `C*`? I would say yes, but the default implementation will make it ill-formed.

The library could add a specialization for pointers, as

```
namespace boost
{

    template <typename A, typename B>
    struct common_type<A*, B*> {
        typedef common_type<A, B>* type;
    };

}
```

But in the absence of a motivating use cases, we prefer not to add more than the standard specifies.

Of course the user can always make this specialization.

## Can you explain the pros/cons of `common_type` against `Boost.Typeof`?

Even if they appear to be close, `common_type` and `typeof` have different purposes. You use `typeof` to get the type of an expression, while you use `common_type` to set explicitly the type returned of a template function. Both are complementary, and indeed `common_type` is equivalent to `decltype(declval<bool>() ? declval<T>() : declval<U>())`

`common_type` is also similar to `promote_args<class ...T>` in `boost/math/tools/promotion.hpp`, though it is not exactly the same as `promote_args` either. `common_type<T1, T2>::type` simply represents the result of some operation on T1 and T2, and defaults to the type obtained by putting T1 and T2 into a conditional statement.

It is meant to be customizable (via specialization) if this default is not appropriate.

## 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 17. Examples**

Expression	Result Type
<code>decay&lt;int[2][3]&gt;::type</code>	<code>int[3]*</code>
<code>decay&lt;int(&amp;)[2]&gt;::type</code>	<code>int*</code>
<code>decay&lt;int(&amp;)(double)&gt;::type</code>	<code>int(*) (double)</code>
<code>int(*) (double</code>	<code>int(*) (double)</code>
<code>int(double)</code>	<code>int(*) (double)</code>

## 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 dimension of type  $T$ .

If  $T$  is not a (built-in) array type, or if  $N > \text{rank}<T>::\text{value}$ , 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], 0>::type` is the type `integral_constant<std::size_t, 2>`.

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

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

`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<boost::array<int, 3>>::value` is an integral constant expression that evaluates to 0:  
`boost::array` is a class type and **not an array type!**

`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 18. Examples**

Expression	Result Type
<code>floating_point_promotion&lt;float const&gt;::type</code>	<code>double const</code>
<code>floating_point_promotion&lt;float&amp;&gt;::type</code>	<code>float&amp;</code>
<code>floating_point_promotion&lt;short&gt;::type</code>	<code>short</code>

## 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 19. Function Traits Members**

Member	Description
<code>function_traits&lt;F&gt;::arity</code>	An integral constant expression that gives the number of arguments accepted by the function type <i>F</i> .
<code>function_traits&lt;F&gt;::result_type</code>	The type returned by function type <i>F</i> .
<code>function_traits&lt;F&gt;::argN_type</code>	The <i>N</i> th argument type of function type <i>F</i> , where $1 \leq N \leq \text{arity of } F$ .

**Table 20. Examples**

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

## has\_bit\_and

```
template <class Lhs, class Rh= Lhs, class Ret= dont_care>
struct has_bit_and : public true_type-or-false_type {};
```

**Inherits:** If (i) lhs of type *Lhs* and rhs of type *Rhs* can be used in expression `lhs&rhs`, and (ii) `Ret= dont_care` or the result of expression `lhs&rhs` is convertible to *Ret* then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (`Ret= dont_care`) is to not check for the return value of binary operator `&`. If *Ret* is different from the default `dont_care` type, the return value is checked to be convertible to *Ret*. Convertible to *Ret* means that the return value of the operator can be used as argument to a function expecting *Ret*:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs&rhs); // is valid if has_bit_and<Lhs, Rhs, Ret>::value==true
```

If *Ret*=`void`, the return type is checked to be exactly `void`.

**Header:** `#include <boost/type_traits/has_bit_and.hpp>` or `#include <boost/type_traits/has_operator.hpp>`  
or `#include <boost/type_traits.hpp>`

**Examples:**

`has_bit_and<Lhs, Rhs, Ret>::value_type` is the type `bool`.

`has_bit_and<Lhs, Rhs, Ret>::value` is a `bool` integral constant expression.

`has_bit_and<int>::value` is a `bool` integral constant expression that evaluates to `true`.

`has_bit_and<long>` inherits from `true_type`.

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

`has_bit_and<const int, int>` inherits from `true_type`.

`has_bit_and<int, double, bool>` inherits from `false_type`.

`has_bit_and<int, int, std::string>` inherits from `false_type`.

See also: [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator&` is public or not: if `operator&` is defined as a private member of `Lhs` then instantiating `has_bit_and<Lhs>` will produce a compiler error. For this reason `has_bit_and` cannot be used to determine whether a type has a public `operator&` or not.

```
struct A { private: void operator&(const A&); };
boost::has_bit_and<A>::value; // error: A::operator&(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator&(const A&, const A&);
struct B { operator A(); };
boost::has_bit_and<A>::value; // this is fine
boost::has_bit_and<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator&` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_bit_and.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator&(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_bit_and< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g&g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_bit_and< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b&b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_bit\_and\_assign

```

template <class Lhs, class Rh= Lhs, class Ret=dont_care>
struct has_bit_and_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rh can be used in expression lhs&=rhs, and (ii) Ret=dont\_care or the result of expression lhs&=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator&=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs&=rhs); // is valid if has_bit_and_assign<Lhs, Rh, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_bit\_and\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_bit\_and\_assign<Lhs, Rh, Ret>::value\_type is the type bool.

has\_bit\_and\_assign<Lhs, Rh, Ret>::value is a bool integral constant expression.

has\_bit\_and\_assign<int>::value is a bool integral constant expression that evaluates to true.

`has_bit_and_assign<long>` inherits from `true_type`.

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

`has_bit_and_assign<const int, int>` inherits from `false_type`.

`has_bit_and_assign<int, double, bool>` inherits from `false_type`.

`has_bit_and_assign<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator&=` is public or not: if `operator&=` is defined as a private member of `Lhs` then instantiating `has_bit_and_assign<Lhs>` will produce a compiler error. For this reason `has_bit_and_assign` cannot be used to determine whether a type has a public `operator&=` or not.

```
struct A { private: void operator&=(const A&); };  
boost::has_bit_and_assign<A>::value; // error: A::operator&=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator&=(const A&, const A&);  
struct B { operator A(); };  
boost::has_bit_and_assign<A>::value; // this is fine  
boost::has_bit_and_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator&=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_bit_and_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator&=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_bit_and_assign< contains< good > >::value<<'\\n'; // true
    contains<good> g;
    g&=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_bit_and_assign< contains< bad > >::value<<'\\n'; // true, should be false

    contains<bad> b;
    b&=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_bit\_or

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_bit_or : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs|rhs, and (ii) Ret=dont\_care or the result of expression lhs|rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator|. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs|rhs); // is valid if has_bit_or<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_bit\_or.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_bit\_or<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_bit\_or<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_bit_or<int>::value` is a bool integral constant expression that evaluates to true.

`has_bit_or<long>` inherits from `true_type`.

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

`has_bit_or<const int, int>` inherits from `true_type`.

`has_bit_or<int, double, bool>` inherits from `false_type`.

`has_bit_or<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether binary `operator|` is public or not: if `operator|` is defined as a private member of Lhs then instantiating `has_bit_or<Lhs>` will produce a compiler error. For this reason `has_bit_or` cannot be used to determine whether a type has a public `operator|` or not.

```
struct A { private: void operator|(const A&); };
boost::has_bit_or<A>::value; // error: A::operator|(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator|(const A&, const A&);
struct B { operator A(); };
boost::has_bit_or<A>::value; // this is fine
boost::has_bit_or<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator|` is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_bit_or.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator|(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_bit_or< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g|g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_bit_or< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b|b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_bit\_or\_assign

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_bit_or_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs|=rhs, and (ii) Ret=dont\_care or the result of expression lhs|=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator|= . If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs|=rhs); // is valid if has_bit_or_assign<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_bit\_or\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_bit\_or\_assign<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_bit\_or\_assign<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_bit\_or\_assign<int>::value is a bool integral constant expression that evaluates to true.

`has_bit_or_assign<long>` inherits from `true_type`.

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

`has_bit_or_assign<const int, int>` inherits from `false_type`.

`has_bit_or_assign<int, double, bool>` inherits from `false_type`.

`has_bit_or_assign<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator| =` is public or not: if `operator| =` is defined as a private member of `Lhs` then instantiating `has_bit_or_assign<Lhs>` will produce a compiler error. For this reason `has_bit_or_assign` cannot be used to determine whether a type has a public `operator| =` or not.

```
struct A { private: void operator|=(const A&); };  
boost::has_bit_or_assign<A>::value; // error: A::operator|=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator|=(const A&, const A&);  
struct B { operator A(); };  
boost::has_bit_or_assign<A>::value; // this is fine  
boost::has_bit_or_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator| =` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_bit_or_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator|=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_bit_or_assign< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g|=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_bit_or_assign< contains< bad > >::value<<'\n'; // true, should be false

    contains<bad> b;
    b|=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_bit\_xor

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_bit_xor : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression `lhs^rhs`, and (ii) `Ret=dont_care` or the result of expression `lhs^rhs` is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (`Ret=dont_care`) is to not check for the return value of binary operator `^`. If Ret is different from the default `dont_care` type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs^rhs); // is valid if has_bit_xor<Lhs, Rhs, Ret>::value==true

```

If `Ret=void`, the return type is checked to be exactly `void`.

**Header:** `#include <boost/type_traits/has_bit_xor.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

### Examples:

`has_bit_xor<Lhs, Rhs, Ret>::value_type` is the type `bool`.

`has_bit_xor<Lhs, Rhs, Ret>::value` is a `bool` integral constant expression.

`has_bit_xor<int>::value` is a bool integral constant expression that evaluates to true.

`has_bit_xor<long>` inherits from `true_type`.

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

`has_bit_xor<const int, int>` inherits from `true_type`.

`has_bit_xor<int, double, bool>` inherits from `false_type`.

`has_bit_xor<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator^` is public or not: if `operator^` is defined as a private member of Lhs then instantiating `has_bit_xor<Lhs>` will produce a compiler error. For this reason `has_bit_xor` cannot be used to determine whether a type has a public `operator^` or not.

```
struct A { private: void operator^(const A&); };
boost::has_bit_xor<A>::value; // error: A::operator^(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator^(const A&, const A&);
struct B { operator A(); };
boost::has_bit_xor<A>::value; // this is fine
boost::has_bit_xor<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator^` is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_bit_xor.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator^(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_bit_xor< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g^g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_bit_xor< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b^b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_bit\_xor\_assign

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_bit_xor_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs^=rhs, and (ii) Ret=dont\_care or the result of expression lhs^=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator^=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs^=rhs); // is valid if has_bit_xor_assign<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_bit\_xor\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_bit\_xor\_assign<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_bit\_xor\_assign<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_bit\_xor\_assign<int>::value is a bool integral constant expression that evaluates to true.

`has_bit_xor_assign<long>` inherits from `true_type`.

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

`has_bit_xor_assign<const int, int>` inherits from `false_type`.

`has_bit_xor_assign<int, double, bool>` inherits from `false_type`.

`has_bit_xor_assign<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator^=` is public or not: if `operator^=` is defined as a private member of `Lhs` then instantiating `has_bit_xor_assign<Lhs>` will produce a compiler error. For this reason `has_bit_xor_assign` cannot be used to determine whether a type has a public `operator^=` or not.

```
struct A { private: void operator^=(const A&); };
boost::has_bit_xor_assign<A>::value; // error: A::operator^=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator^=(const A&, const A&);
struct B { operator A(); };
boost::has_bit_xor_assign<A>::value; // this is fine
boost::has_bit_xor_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator^=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_bit_xor_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator^=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_bit_xor_assign< contains< good > >::value<<'\\n'; // true
    contains<good> g;
    g^=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_bit_xor_assign< contains< bad > >::value<<'\\n'; // true, should be false

    contains<bad> b;
    b^=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_complement

```

template <class Rhs, class Ret=dont_care>
struct has_complement : public true_type-or-false_type {};

```

**Inherits:** If (i) rhs of type Rhs can be used in expression ~rhs, and (ii) Ret=dont\_care or the result of expression ~rhs is convertible to Ret then inherits from `true_type`, otherwise inherits from `false_type`.

The default behaviour (Ret=dont\_care) is to not check for the return value of prefix operator~. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Rhs rhs;
f(~rhs); // is valid if has_complement<Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_complement.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_complement<Rhs, Ret>::value\_type is the type bool.

has\_complement<Rhs, Ret>::value is a bool integral constant expression.

has\_complement<int>::value is a bool integral constant expression that evaluates to true.

```
has_complement<long> inherits from true_type.  
  
has_complement<int, int> inherits from true_type.  
  
has_complement<int, long> inherits from true_type.  
  
has_complement<const int> inherits from true_type.  
  
has_complement<int*> inherits from false_type.  
  
has_complement<double, double> inherits from false_type.  
  
has_complement<double, int> inherits from false_type.  
  
has_complement<int, std::string> inherits from false_type.
```

See also: [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether prefix `operator~` is public or not: if `operator~` is defined as a private member of `Rhs` then instantiating `has_complement<Rhs>` will produce a compiler error. For this reason `has_complement` cannot be used to determine whether a type has a public `operator~` or not.

```
struct A { private: void operator~(); };  
boost::has_complement<A>::value; // error: A::operator~() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator~(const A&);  
struct B { operator A(); };  
boost::has_complement<A>::value; // this is fine  
boost::has_complement<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator~` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_complement.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator~(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_complement< contains< good > >::value<<'\n'; // true
    contains<good> g;
    ~g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_complement< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    ~b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_dereference

```

template <class Rhs, class Ret=dont_care>
struct has_dereference : public true_type-or-false_type {};

```

**Inherits:** If (i) rhs of type Rhs can be used in expression \*rhs, and (ii) Ret=dont\_care or the result of expression \*rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of prefix operator\*. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Rhs rhs;
f(*rhs); // is valid if has_dereference<Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_dereference.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_dereference<Rhs, Ret>::value\_type is the type bool.

has\_dereference<Rhs, Ret>::value is a bool integral constant expression.

has\_dereference<int\*>::value is a bool integral constant expression that evaluates to true.

`has_dereference<long*>` inherits from `true_type`.

`has_dereference<int*, int>` inherits from `true_type`.

`has_dereference<int*, const int>` inherits from `true_type`.

`has_dereference<int const *>` inherits from `true_type`.

`has_dereference<int * const>` inherits from `true_type`.

`has_dereference<int const * const>` inherits from `true_type`.

`has_dereference<int>` inherits from `false_type`.

`has_dereference<double>` inherits from `false_type`.

`has_dereference<void*>` inherits from `false_type`.

`has_dereference<const int*, int&>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether prefix `operator*` is public or not: if `operator*` is defined as a private member of `Rhs` then instantiating `has_dereference<Rhs>` will produce a compiler error. For this reason `has_dereference` cannot be used to determine whether a type has a public `operator*` or not.

```
struct A { private: void operator*(); };  
boost::has_dereference<A>::value; // error: A::operator*() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator*(const A&);  
struct B { operator A(); };  
boost::has_dereference<A>::value; // this is fine  
boost::has_dereference<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator*` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_dereference.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator*(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_dereference< contains< good > >::value<<'\n'; // true
    contains<good> g;
    *g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_dereference< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    *b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_divides

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_divides : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs/rhs, and (ii) Ret=dont\_care or the result of expression lhs/rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator/. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs/rhs); // is valid if has_divides<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_divides.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_divides<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_divides<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_divides<int>::value is a bool integral constant expression that evaluates to true.

```
has_divides<long> inherits from true_type.

has_divides<int, int, int> inherits from true_type.

has_divides<int, int, long> inherits from true_type.

has_divides<int, double, double> inherits from true_type.

has_divides<int, double, int> inherits from true_type.

has_divides<const int, int>::value inherits from true_type.

has_divides<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator/` is public or not: if `operator/` is defined as a private member of `Lhs` then instantiating `has_divides<Lhs>` will produce a compiler error. For this reason `has_divides` cannot be used to determine whether a type has a public `operator/` or not.

```
struct A { private: void operator/(const A&); };
boost::has_divides<A>::value; // error: A::operator/(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator/(const A&, const A&);
struct B { operator A(); };
boost::has_divides<A>::value; // this is fine
boost::has_divides<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator/` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_divides.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator/(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_divides< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g/g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_divides< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b/b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_divides\_assign

```

template <class Lhs, class Rh= Lhs, class Ret=dont_care>
struct has_divides_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rh can be used in expression lhs/=rhs, and (ii) Ret=dont\_care or the result of expression lhs/=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator/=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs/=rhs); // is valid if has_divides_assign<Lhs, Rh, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_divides\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_divides\_assign<Lhs, Rh, Ret>::value\_type is the type bool.

has\_divides\_assign<Lhs, Rh, Ret>::value is a bool integral constant expression.

has\_divides\_assign<int>::value is a bool integral constant expression that evaluates to true.

```
has_divides_assign<long> inherits from true_type.  
has_divides_assign<int, int, int> inherits from true_type.  
has_divides_assign<int, int, long> inherits from true_type.  
has_divides_assign<int, double, double> inherits from true_type.  
has_divides_assign<int, double, int> inherits from true_type.  
has_divides_assign<const int, int>::value inherits from false_type.  
has_divides_assign<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator/=` is public or not: if `operator/=` is defined as a private member of `Lhs` then instantiating `has_divides_assign<Lhs>` will produce a compiler error. For this reason `has_divides_assign` cannot be used to determine whether a type has a public `operator/=` or not.

```
struct A { private: void operator/=(const A&); };  
boost::has_divides_assign<A>::value; // error: A::operator/=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator/=(const A&, const A&);  
struct B { operator A(); };  
boost::has_divides_assign<A>::value; // this is fine  
boost::has_divides_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator/=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_divides_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator/=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_divides_assign< contains< good > >::value<<'\\n'; // true
    contains<good> g;
    g/=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_divides_assign< contains< bad > >::value<<'\\n'; // true, should be false

    contains<bad> b;
    b/=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_equal\_to

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_equal_to : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs==rhs, and (ii) Ret=dont\_care or the result of expression lhs==rhs is convertible to Ret then inherits from `true_type`, otherwise inherits from `false_type`.

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator==. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs==rhs); // is valid if has_equal_to<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_equal\_to.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_equal\_to<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_equal\_to<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_equal_to<int>::value` is a bool integral constant expression that evaluates to true.

`has_equal_to<long>` inherits from `true_type`.

`has_equal_to<int, int, bool>` inherits from `true_type`.

`has_equal_to<int, double, bool>` inherits from `true_type`.

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

`has_equal_to<int*, int>` inherits from `false_type`.

`has_equal_to<int*, double*>` inherits from `false_type`.

`has_equal_to<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether binary `operator==` is public or not: if `operator==` is defined as a private member of Lhs then instantiating `has_equal_to<Lhs>` will produce a compiler error. For this reason `has_equal_to` cannot be used to determine whether a type has a public `operator==` or not.

```
struct A { private: void operator==(const A&); };
boost::has_equal_to<A>::value; // error: A::operator==(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator==(const A&, const A&);
struct B { operator A(); };
boost::has_equal_to<A>::value; // this is fine
boost::has_equal_to<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator==` is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_equal_to.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator==(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_equal_to< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g==g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_equal_to< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b==b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_greater

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_greater : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs>rhs, and (ii) Ret=dont\_care or the result of expression lhs>rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator>. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs>rhs); // is valid if has_greater<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_greater.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_greater<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_greater<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_greater<int>::value is a bool integral constant expression that evaluates to true.

```
has_greater<long> inherits from true_type.  
  
has_greater<int, int, bool> inherits from true_type.  
  
has_greater<int, double, bool> inherits from true_type.  
  
has_greater<const int> inherits from true_type.  
  
has_greater<int*, int> inherits from false_type.  
  
has_greater<int*, double*> inherits from false_type.  
  
has_greater<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator>` is public or not: if `operator>` is defined as a private member of `Lhs` then instantiating `has_greater<Lhs>` will produce a compiler error. For this reason `has_greater` cannot be used to determine whether a type has a public `operator>` or not.

```
struct A { private: void operator>(const A&); };  
boost::has_greater<A>::value; // error: A::operator>(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator>(const A&, const A&);  
struct B { operator A(); };  
boost::has_greater<A>::value; // this is fine  
boost::has_greater<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator>` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_greater.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator>(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_greater< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g>g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_greater< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b>b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_greater\_equal

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_greater_equal : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs>=rhs, and (ii) Ret=dont\_care or the result of expression lhs>=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator>=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs==rhs); // is valid if has_greater_equal<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_greater\_equal.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_greater\_equal<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_greater\_equal<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_greater\_equal<int>::value is a bool integral constant expression that evaluates to true.

```
has_greater_equal<long> inherits from true_type.  
  
has_greater_equal<int, int, bool> inherits from true_type.  
  
has_greater_equal<int, double, bool> inherits from true_type.  
  
has_greater_equal<const int> inherits from true_type.  
  
has_greater_equal<int*, int> inherits from false_type.  
  
has_greater_equal<int*, double*> inherits from false_type.  
  
has_greater_equal<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator>=` is public or not: if `operator>=` is defined as a private member of `Lhs` then instantiating `has_greater_equal<Lhs>` will produce a compiler error. For this reason `has_greater_equal` cannot be used to determine whether a type has a public `operator>=` or not.

```
struct A { private: void operator>=(const A&); };  
boost::has_greater_equal<A>::value; // error: A::operator>=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator>=(const A&, const A&);  
struct B { operator A(); };  
boost::has_greater_equal<A>::value; // this is fine  
boost::has_greater_equal<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator>=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_greater_equal.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator>=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_greater_equal< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g>=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_greater_equal< contains< bad > >::value<<'\n'; // true, should be false

    contains<bad> b;
    b>=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_left\_shift

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_left_shift : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs<<rhs, and (ii) Ret=dont\_care or the result of expression lhs<<rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator<<. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs<<rhs); // is valid if has_left_shift<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_left\_shift.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_left\_shift<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_left\_shift<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_left_shift<int>::value` is a `bool` integral constant expression that evaluates to `true`.

`has_left_shift<long>` inherits from `true_type`.

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

`has_left_shift<const int, int>` inherits from `true_type`.

`has_left_shift<std::ostream, int>` inherits from `true_type`.

`has_left_shift<std::ostream, char*, std::ostream>` inherits from `true_type`.

`has_left_shift<std::ostream, std::string>` inherits from `true_type`.

`has_left_shift<int, double, bool>` inherits from `false_type`.

`has_left_shift<int, int, std::string>` inherits from `false_type`.

See also: [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether binary `operator<<` is public or not: if `operator<<` is defined as a private member of `Lhs` then instantiating `has_left_shift<Lhs>` will produce a compiler error. For this reason `has_left_shift` cannot be used to determine whether a type has a public `operator<<` or not.

```
struct A { private: void operator<<(const A&); };
boost::has_left_shift<A>::value; // error: A::operator<<(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator<<(const A&, const A&);
struct B { operator A(); };
boost::has_left_shift<A>::value; // this is fine
boost::has_left_shift<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator<<` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```
#include <boost/type_traits/has_left_shift.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator<< (const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_left_shift< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g<<g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_left_shift< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b<<b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_left\_shift\_assign

```
template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_left_shift_assign : public true_type-or-false_type {};
```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs<<=rhs, and (ii) Ret=dont\_care or the result of expression lhs<<=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator<<=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs<<=rhs); // is valid if has_left_shift_assign<Lhs, Rhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** `#include <boost/type_traits/has_left_shift_assign.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

### Examples:

`has_left_shift_assign<Lhs, Rhs, Ret>::value_type` is the type `bool`.

`has_left_shift_assign<Lhs, Rhs, Ret>::value` is a `bool` integral constant expression.

`has_left_shift_assign<int>::value` is a `bool` integral constant expression that evaluates to `true`.

`has_left_shift_assign<long>` inherits from `true_type`.

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

`has_left_shift_assign<const int, int>` inherits from `false_type`.

`has_left_shift_assign<int, double, bool>` inherits from `false_type`.

`has_left_shift_assign<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator<=` is public or not: if `operator<=` is defined as a private member of Lhs then instantiating `has_left_shift_assign<Lhs>` will produce a compiler error. For this reason `has_left_shift_assign` cannot be used to determine whether a type has a public `operator<=` or not.

```
struct A { private: void operator<=(const A&); };  
boost::has_left_shift_assign<A>::value; // error: A::operator<=(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator<=(const A&, const A&);  
struct B { operator A(); };  
boost::has_left_shift_assign<A>::value; // this is fine  
boost::has_left_shift_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator<=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_left_shift_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator<<=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_left_shift_assign< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g<<=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_left_shift_assign< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b<<=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_less

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_less : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs<rhs, and (ii) Ret=dont\_care or the result of expression lhs<rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator<. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs<rhs); // is valid if has_less<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_less.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_less<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_less<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_less<int>::value` is a `bool` integral constant expression that evaluates to `true`.

`has_less<long>` inherits from `true_type`.

`has_less<int, int, bool>` inherits from `true_type`.

`has_less<int, double, bool>` inherits from `true_type`.

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

`has_less<int*, int>` inherits from `false_type`.

`has_less<int*, double*>` inherits from `false_type`.

`has_less<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether `binary operator<` is public or not: if `operator<` is defined as a private member of `Lhs` then instantiating `has_less<Lhs>` will produce a compiler error. For this reason `has_less` cannot be used to determine whether a type has a public `operator<` or not.

```
struct A { private: void operator<(const A&); };
boost::has_less<A>::value; // error: A::operator<(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator<(const A&, const A&);
struct B { operator A(); };
boost::has_less<A>::value; // this is fine
boost::has_less<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator<` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_less.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator<(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_less< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g<g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_less< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b<b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_less\_equal

```

template <class Lhs, class Rh= Lhs, class Ret=dont_care>
struct has_less_equal : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rh can be used in expression lhs<=rhs, and (ii) Ret=dont\_care or the result of expression lhs<=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator<=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rh rhs;
f(lhs<=rhs); // is valid if has_less_equal<Lhs, Rh, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_less\_equal.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_less\_equal<Lhs, Rh, Ret>::value\_type is the type bool.

has\_less\_equal<Lhs, Rh, Ret>::value is a bool integral constant expression.

has\_less\_equal<int>::value is a bool integral constant expression that evaluates to true.

```
has_less_equal<long> inherits from true_type.  
  
has_less_equal<int, int, bool> inherits from true_type.  
  
has_less_equal<int, double, bool> inherits from true_type.  
  
has_less_equal<const int> inherits from true_type.  
  
has_less_equal<int*, int> inherits from false_type.  
  
has_less_equal<int*, double*> inherits from false_type.  
  
has_less_equal<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator<=` is public or not: if `operator<=` is defined as a private member of `Lhs` then instantiating `has_less_equal<Lhs>` will produce a compiler error. For this reason `has_less_equal` cannot be used to determine whether a type has a public `operator<=` or not.

```
struct A { private: void operator<=(const A&); };  
boost::has_less_equal<A>::value; // error: A::operator<=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator<=(const A&, const A&);  
struct B { operator A(); };  
boost::has_less_equal<A>::value; // this is fine  
boost::has_less_equal<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator<=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```
#include <boost/type_traits/has_less_equal.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator<=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_less_equal< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g<=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_less_equal< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b<=b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_logical\_and

```
template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_logical_and : public true_type-or-false_type {};
```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs&&rhs, and (ii) Ret=dont\_care or the result of expression lhs&&rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator&&. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs&&rhs); // is valid if has_logical_and<Lhs, Rhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_logical\_and.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_logical\_and<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_logical\_and<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_logical\_and<int>::value is a bool integral constant expression that evaluates to true.

```
has_logical_and<bool> inherits from true_type.  
  
has_logical_and<int, int, bool> inherits from true_type.  
  
has_logical_and<int, int, long> inherits from true_type.  
  
has_logical_and<int, double, bool> inherits from true_type.  
  
has_logical_and<const int, int>::value inherits from true_type.  
  
has_logical_and<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary operator&& is public or not: if operator&& is defined as a private member of Lhs then instantiating has\_logical\_and<Lhs> will produce a compiler error. For this reason has\_logical\_and cannot be used to determine whether a type has a public operator&& or not.

```
struct A { private: void operator&&(const A&); };  
boost::has_logical_and<A>::value; // error: A::operator&&(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator&&(const A&, const A&);  
struct B { operator A(); };  
boost::has_logical_and<A>::value; // this is fine  
boost::has_logical_and<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator&& is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_logical_and.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator&&(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_logical_and< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g&&g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_logical_and< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b&&b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_logical\_not

```

template <class Rhs, class Ret=dont_care>
struct has_logical_not : public true_type-or-false_type {};

```

**Inherits:** If (i) rhs of type Rhs can be used in expression !rhs, and (ii) Ret=dont\_care or the result of expression !rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of prefix operator!. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Rhs rhs;
f(!rhs); // is valid if has_logical_not<Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_logical\_not.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_logical\_not<Rhs, Ret>::value\_type is the type bool.

has\_logical\_not<Rhs, Ret>::value is a bool integral constant expression.

has\_logical\_not<int>::value is a bool integral constant expression that evaluates to true.

```
has_logical_not<bool> inherits from true_type.  
  
has_logical_not<int, bool> inherits from true_type.  
  
has_logical_not<int, long> inherits from true_type.  
  
has_logical_not<double, double> inherits from true_type.  
  
has_logical_not<double, bool> inherits from true_type.  
  
has_logical_not<const bool> inherits from true_type.  
  
has_logical_not<int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether prefix `operator!` is public or not: if `operator!` is defined as a private member of `Rhs` then instantiating `has_logical_not<Rhs>` will produce a compiler error. For this reason `has_logical_not` cannot be used to determine whether a type has a public `operator!` or not.

```
struct A { private: void operator!(); };  
boost::has_logical_not<A>::value; // error: A::operator!() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator!(const A&);  
struct B { operator A(); };  
boost::has_logical_not<A>::value; // this is fine  
boost::has_logical_not<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator!` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_logical_not.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator!(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_logical_not< contains< good > >::value<<'\n'; // true
    contains<good> g;
    !g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_logical_not< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    !b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_logical\_or

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_logical_or : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs | rhs, and (ii) Ret=dont\_care or the result of expression lhs | rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator |. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs | rhs); // is valid if has_logical_or<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_logical\_or.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_logical\_or<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_logical\_or<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_logical\_or<int>::value is a bool integral constant expression that evaluates to true.

```
has_logical_or<bool> inherits from true_type.  
  
has_logical_or<int, int, bool> inherits from true_type.  
  
has_logical_or<int, int, long> inherits from true_type.  
  
has_logical_or<int, double, bool> inherits from true_type.  
  
has_logical_or<const int, int>::value inherits from true_type.  
  
has_logical_or<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary operator|| is public or not: if operator|| is defined as a private member of Lhs then instantiating has\_logical\_or<Lhs> will produce a compiler error. For this reason has\_logical\_or cannot be used to determine whether a type has a public operator|| or not.

```
struct A { private: void operator||(const A&); };  
boost::has_logical_or<A>::value; // error: A::operator||(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator||(const A&, const A&);  
struct B { operator A(); };  
boost::has_logical_or<A>::value; // this is fine  
boost::has_logical_or<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator|| is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_logical_or.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator||(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_logical_or< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g||g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_logical_or< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b||b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_minus

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_minus : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs-rhs, and (ii) Ret=dont\_care or the result of expression lhs-rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator-. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs-rhs); // is valid if has_minus<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_minus.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_minus<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_minus<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_minus<int>::value is a bool integral constant expression that evaluates to true.

```
has_minus<long> inherits from true_type.  
  
has_minus<int, int, int> inherits from true_type.  
  
has_minus<int, int, long> inherits from true_type.  
  
has_minus<int, double, double> inherits from true_type.  
  
has_minus<int, double, int> inherits from true_type.  
  
has_minus<const int, int>::value inherits from true_type.  
  
has_minus<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator-` is public or not: if `operator-` is defined as a private member of `Lhs` then instantiating `has_minus<Lhs>` will produce a compiler error. For this reason `has_minus` cannot be used to determine whether a type has a public `operator-` or not.

```
struct A { private: void operator-(const A&); };  
boost::has_minus<A>::value; // error: A::operator-(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator-(const A&, const A&);  
struct B { operator A(); };  
boost::has_minus<A>::value; // this is fine  
boost::has_minus<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator-` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_minus.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator-(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_minus< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g-g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_minus< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b-b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_minus\_assign

```

template <class Lhs, class Rh= Lhs, class Ret=dont_care>
struct has_minus_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rh can be used in expression lhs-=rhs, and (ii) Ret=dont\_care or the result of expression lhs-=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator-=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs-=rhs); // is valid if has_minus_assign<Lhs, Rh, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_minus\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_minus\_assign<Lhs, Rh, Ret>::value\_type is the type bool.

has\_minus\_assign<Lhs, Rh, Ret>::value is a bool integral constant expression.

has\_minus\_assign<int>::value is a bool integral constant expression that evaluates to true.

```
has_minus_assign<long> inherits from true_type.  
  
has_minus_assign<int, int, int> inherits from true_type.  
  
has_minus_assign<int, int, long> inherits from true_type.  
  
has_minus_assign<int, double, double> inherits from true_type.  
  
has_minus_assign<int, double, int> inherits from true_type.  
  
has_minus_assign<const int, int>::value inherits from false_type.  
  
has_minus_assign<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator-=` is public or not: if `operator-=` is defined as a private member of `Lhs` then instantiating `has_minus_assign<Lhs>` will produce a compiler error. For this reason `has_minus_assign` cannot be used to determine whether a type has a public `operator-=` or not.

```
struct A { private: void operator-=(const A&); };  
boost::has_minus_assign<A>::value; // error: A::operator-=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator-=(const A&, const A&);  
struct B { operator A(); };  
boost::has_minus_assign<A>::value; // this is fine  
boost::has_minus_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator-=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_minus_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator-=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_minus_assign< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g-=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_minus_assign< contains< bad > >::value<<'\n'; // true, should be false

    contains<bad> b;
    b-=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_modulus

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_modulus : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs%rhs, and (ii) Ret=dont\_care or the result of expression lhs%rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator%. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs%rhs); // is valid if has_modulus<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_modulus.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_modulus<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_modulus<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_modulus<int>::value` is a bool integral constant expression that evaluates to true.

`has_modulus<long>` inherits from `true_type`.

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

`has_modulus<int, int, long>` inherits from `true_type`.

`has_modulus<const int, int>::value` inherits from `true_type`.

`has_modulus<int, double>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator%` is public or not: if `operator%` is defined as a private member of `Lhs` then instantiating `has_modulus<Lhs>` will produce a compiler error. For this reason `has_modulus` cannot be used to determine whether a type has a public `operator%` or not.

```
struct A { private: void operator%(const A&); };  
boost::has_modulus<A>::value; // error: A::operator%(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator%(const A&, const A&);  
struct B { operator A(); };  
boost::has_modulus<A>::value; // this is fine  
boost::has_modulus<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator%` is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_modulus.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator%(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_modulus< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g%g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_modulus< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b%b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_modulus\_assign

```

template <class Lhs, class Rh= Lhs, class Ret=dont_care>
struct has_modulus_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rh can be used in expression lhs%=rhs, and (ii) Ret=dont\_care or the result of expression lhs%=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator%=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs%=rhs); // is valid if has_modulus_assign<Lhs, Rh, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_modulus\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_modulus\_assign<Lhs, Rh, Ret>::value\_type is the type bool.

has\_modulus\_assign<Lhs, Rh, Ret>::value is a bool integral constant expression.

has\_modulus\_assign<int>::value is a bool integral constant expression that evaluates to true.

```
has_modulus_assign<long> inherits from true_type.  
  
has_modulus_assign<int, int, int> inherits from true_type.  
  
has_modulus_assign<int, int, long> inherits from true_type.  
  
has_modulus_assign<const int, int>::value inherits from false_type.  
  
has_modulus_assign<int, double> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary operator%= is public or not: if operator%= is defined as a private member of Lhs then instantiating has\_modulus\_assign<Lhs> will produce a compiler error. For this reason has\_modulus\_assign cannot be used to determine whether a type has a public operator%= or not.

```
struct A { private: void operator%=(const A&); };  
boost::has_modulus_assign<A>::value; // error: A::operator%=(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator%=(const A&, const A&);  
struct B { operator A(); };  
boost::has_modulus_assign<A>::value; // this is fine  
boost::has_modulus_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator%= is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_modulus_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator%=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_modulus_assign< contains< good > >::value<<'\\n'; // true
    contains<good> g;
    g%=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_modulus_assign< contains< bad > >::value<<'\\n'; // true, should be false

    contains<bad> b;
    b%=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_multiplies

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_multiplies : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs\*rhs, and (ii) Ret=dont\_care or the result of expression lhs\*rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator\*. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs*rhs); // is valid if has_multiplies<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_multiplies.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_multiplies<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_multiplies<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_multiplies<int>::value` is a bool integral constant expression that evaluates to true.

`has_multiplies<long>` inherits from `true_type`.

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

`has_multiplies<int, int, long>` inherits from `true_type`.

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

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

`has_multiplies<const int, int>::value` inherits from `true_type`.

`has_multiplies<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator*` is public or not: if `operator*` is defined as a private member of `Lhs` then instantiating `has_multiplies<Lhs>` will produce a compiler error. For this reason `has_multiplies` cannot be used to determine whether a type has a public `operator*` or not.

```
struct A { private: void operator*(const A&); };
boost::has_multiplies<A>::value; // error: A::operator*(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator*(const A&, const A&);
struct B { operator A(); };
boost::has_multiplies<A>::value; // this is fine
boost::has_multiplies<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator*` is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_multiplies.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator*(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_multiplies< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g*g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_multiplies< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b*b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_multiplies\_assign

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_multiplies_assign : public true_type-or-false_type { };

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs\*=rhs, and (ii) Ret=dont\_care or the result of expression lhs\*=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator\*=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs*=rhs); // is valid if has_multiplies_assign<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** `#include <boost/type_traits/has_multiplies_assign.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

### Examples:

`has_multiplies_assign<Lhs, Rhs, Ret>::value_type` is the type `bool`.

`has_multiplies_assign<Lhs, Rhs, Ret>::value` is a `bool` integral constant expression.

`has_multiplies_assign<int>::value` is a `bool` integral constant expression that evaluates to `true`.

`has_multiplies_assign<long>` inherits from `true_type`.

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

`has_multiplies_assign<int, int, long>` inherits from `true_type`.

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

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

`has_multiplies_assign<const int, int>::value` inherits from `false_type`.

`has_multiplies_assign<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

### Limitation:

- Requires a compiler with working SFINAE.

### Known issues:

- This trait cannot detect whether binary `operator*=` is public or not: if `operator*=` is defined as a private member of `Lhs` then instantiating `has_multiplies_assign<Lhs>` will produce a compiler error. For this reason `has_multiplies_assign` cannot be used to determine whether a type has a public `operator*=` or not.

```
struct A { private: void operator*=(const A&); };  
boost::has_multiplies_assign<A>::value; // error: A::operator*=(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator*=(const A&, const A&);  
struct B { operator A(); };  
boost::has_multiplies_assign<A>::value; // this is fine  
boost::has_multiplies_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator*=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_multiplies_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator*=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_multiplies_assign< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g*=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_multiplies_assign< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b*=b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_negate

```

template <class Rhs, class Ret=dont_care>
struct has_negate : public true_type-or-false_type {};

```

**Inherits:** If (i) rhs of type Rhs can be used in expression `-rhs`, and (ii) `Ret=dont_care` or the result of expression `-rhs` is convertible to Ret then inherits from `true_type`, otherwise inherits from `false_type`.

The default behaviour (`Ret=dont_care`) is to not check for the return value of prefix operator `-`. If Ret is different from the default `dont_care` type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Rhs rhs;
f(-rhs); // is valid if has_negate<Rhs, Ret>::value==true

```

If `Ret=void`, the return type is checked to be exactly void.

**Header:** `#include <boost/type_traits/has_negate.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

### Examples:

`has_negate<Rhs, Ret>::value_type` is the type `bool`.

`has_negate<Rhs, Ret>::value` is a `bool` integral constant expression.

`has_negate<int>::value` is a `bool` integral constant expression that evaluates to true.

```
has_negate<long> inherits from true_type.  
has_negate<int, int> inherits from true_type.  
has_negate<int, long> inherits from true_type.  
has_negate<double, double> inherits from true_type.  
has_negate<double, int> inherits from true_type.  
has_negate<const int> inherits from true_type.  
has_negate<int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether prefix `operator-` is public or not: if `operator-` is defined as a private member of `Rhs` then instantiating `has_negate<Rhs>` will produce a compiler error. For this reason `has_negate` cannot be used to determine whether a type has a public `operator-` or not.

```
struct A { private: void operator-(); };  
boost::has_negate<A>::value; // error: A::operator-() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator-(const A&);  
struct B { operator A(); };  
boost::has_negate<A>::value; // this is fine  
boost::has_negate<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator-` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_negate.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator-(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_negate< contains< good > >::value<<'\n'; // true
    contains<good> g;
    -g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_negate< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    -b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## 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\_not\_equal\_to

```
template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_not_equal_to : public true\_type-or-false\_type {};
```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs!=rhs, and (ii) Ret=dont\_care or the result of expression lhs!=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator!=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs!=rhs); // is valid if has_not_equal_to<Lhs, Rhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_not\_equal\_to.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_not\_equal\_to<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_not\_equal\_to<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_not\_equal\_to<int>::value is a bool integral constant expression that evaluates to true.

has\_not\_equal\_to<long> inherits from [true\\_type](#).

has\_not\_equal\_to<int, int, bool> inherits from [true\\_type](#).

has\_not\_equal\_to<int, double, bool> inherits from [true\\_type](#).

has\_not\_equal\_to<const int> inherits from [true\\_type](#).

has\_not\_equal\_to<int\*, int> inherits from [false\\_type](#).

has\_not\_equal\_to<int\*, double\*> inherits from [false\\_type](#).

has\_not\_equal\_to<int, int, std::string> inherits from [false\\_type](#).

**See also:** [Operator Type Traits](#)

### Limitation:

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary operator!= is public or not: if operator!= is defined as a private member of Lhs then instantiating has\_not\_equal\_to<Lhs> will produce a compiler error. For this reason has\_not\_equal\_to cannot be used to determine whether a type has a public operator!= or not.

```
struct A { private: void operator!=(const A&); };
boost::has_not_equal_to<A>::value; // error: A::operator!=(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator!=(const A&, const A&);
struct B { operator A(); };
boost::has_not_equal_to<A>::value; // this is fine
boost::has_not_equal_to<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator!= is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```
#include <boost/type_traits/has_not_equal_to.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator!=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_not_equal_to< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g!=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_not_equal_to< contains< bad > >::value<<'\n'; // true, should be false

    contains<bad> b;
    b!=b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## 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 (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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_HAS_NOTHROW_CONSTRUCTOR` is defined.

**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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_HAS_NOTHROW_COPY` is defined.

**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\_plus

```
template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_plus : public true_type-or-false_type {};
```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs+rhs, and (ii) Ret=dont\_care or the result of expression lhs+rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator+. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs+rhs); // is valid if has_plus<Lhs, Rhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** `#include <boost/type_traits/has_plus.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

### Examples:

`has_plus<Lhs, Rhs, Ret>::value_type` is the type `bool`.

`has_plus<Lhs, Rhs, Ret>::value` is a `bool` integral constant expression.

`has_plus<int>::value` is a `bool` integral constant expression that evaluates to `true`.

`has_plus<long>` inherits from [true\\_type](#).

`has_plus<int, int, int>` inherits from [true\\_type](#).

`has_plus<int, int, long>` inherits from [true\\_type](#).

`has_plus<int, double, double>` inherits from [true\\_type](#).

`has_plus<int, double, int>` inherits from [true\\_type](#).

`has_plus<const int, int>::value` inherits from [true\\_type](#).

`has_plus<int, int, std::string>` inherits from [false\\_type](#).

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator+` is public or not: if `operator+` is defined as a private member of `Lhs` then instantiating `has_plus<Lhs>` will produce a compiler error. For this reason `has_plus` cannot be used to determine whether a type has a public `operator+` or not.

```
struct A { private: void operator+(const A&); };
boost::has_plus<A>::value; // error: A::operator+(const A&) is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator+(const A&, const A&);
struct B { operator A(); };
boost::has_plus<A>::value; // this is fine
boost::has_plus<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator+` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```
#include <boost/type_traits/has_plus.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator+(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_plus< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g+g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_plus< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b+b; // compile time error
    return 0;
}
```

- `volatile` qualifier is not properly handled and would lead to undefined behavior

## has\_plus\_assign

```
template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_plus_assign : public true_type-or-false_type {};
```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs+=rhs, and (ii) Ret=dont\_care or the result of expression lhs+=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator+=". If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs+=rhs); // is valid if has_plus_assign<Lhs, Rhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_plus\_assign.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

```
has_plus_assign<Lhs, Rhs, Ret>::value_type is the type bool.

has_plus_assign<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has_plus_assign<int>::value is a bool integral constant expression that evaluates to true.

has_plus_assign<long> inherits from true_type.

has_plus_assign<int, int, int> inherits from true_type.

has_plus_assign<int, int, long> inherits from true_type.

has_plus_assign<int, double, double> inherits from true_type.

has_plus_assign<int, double, int> inherits from true_type.

has_plus_assign<const int, int>::value inherits from false_type.

has_plus_assign<int, int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

### Limitation:

- Requires a compiler with working SFINAE.

### Known issues:

- This trait cannot detect whether binary operator+= is public or not: if operator+= is defined as a private member of Lhs then instantiating has\_plus\_assign<Lhs> will produce a compiler error. For this reason has\_plus\_assign cannot be used to determine whether a type has a public operator+= or not.

```
struct A { private: void operator+=(const A&); };
boost::has_plus_assign<A>::value; // error: A::operator+=(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator+=(const A&, const A&);
struct B { operator A(); };
boost::has_plus_assign<A>::value; // this is fine
boost::has_plus_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator+= is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```
#include <boost/type_traits/has_plus_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator+=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_plus_assign< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g+=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_plus_assign< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b+=b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_post\_decrement

```
template <class Lhs, class Ret=dont_care>
struct has_post_decrement : public true_type-or-false_type { };
```

**Inherits:** If (i) lhs of type Lhs can be used in expression lhs--, and (ii) Ret=dont\_care or the result of expression lhs-- is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of postfix operator--. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Lhs lhs;
f(lhs--); // is valid if has_post_decrement<Lhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** `#include <boost/type_traits/has_post_decrement.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

**Examples:**

```
has_post_decrement<Lhs, Ret>::value_type is the type bool.

has_post_decrement<Lhs, Ret>::value is a bool integral constant expression.

has_post_decrement<int>::value is a bool integral constant expression that evaluates to true.

has_post_decrement<long> inherits from true_type.

has_post_decrement<int, int> inherits from true_type.

has_post_decrement<int, long> inherits from true_type.

has_post_decrement<double, double> inherits from true_type.

has_post_decrement<double, int> inherits from true_type.

has_post_decrement<bool> inherits from false_type.

has_post_decrement<const int> inherits from false_type.

has_post_decrement<void*> inherits from false_type.

has_post_decrement<int, std::string> inherits from false_type.
```

See also: [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether postfix operator-- is public or not: if operator-- is defined as a private member of Lhs then instantiating has\_post\_decrement<Lhs> will produce a compiler error. For this reason has\_post\_decrement cannot be used to determine whether a type has a public operator-- or not.

```
struct A { private: void operator--(int); };
boost::has_post_decrement<A>::value; // error: A::operator--(int) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator--(const A&, int);
struct B { operator A(); };
boost::has_post_decrement<A>::value; // this is fine
boost::has_post_decrement<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator-- is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_post_decrement.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator--(const contains<T> &lhs, int) {
    return f(lhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_post_decrement< contains< good > >::value<<'\\n'; // true
    contains<good> g;
    g--; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_post_decrement< contains< bad > >::value<<'\\n'; // true, should be false

    contains<bad> b;
    b--; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_post\_increment

```

template <class Lhs, class Ret=dont_care>
struct has_post_increment : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs can be used in expression lhs++, and (ii) Ret=dont\_care or the result of expression lhs++ is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of postfix operator++. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
f(lhs++); // is valid if has_post_increment<Lhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_post\_increment.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_post\_increment<Lhs, Ret>::value\_type is the type bool.

has\_post\_increment<Lhs, Ret>::value is a bool integral constant expression.

has\_post\_increment<int>::value is a bool integral constant expression that evaluates to true.

```
has_post_increment<long> inherits from true_type.  
  
has_post_increment<int, int> inherits from true_type.  
  
has_post_increment<int, long> inherits from true_type.  
  
has_post_increment<double, double> inherits from true_type.  
  
has_post_increment<double, int> inherits from true_type.  
  
has_post_increment<bool> inherits from true_type.  
  
has_post_increment<const int> inherits from false_type.  
  
has_post_increment<void*> inherits from false_type.  
  
has_post_increment<int, std::string> inherits from false_type.
```

See also: [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether postfix operator++ is public or not: if operator++ is defined as a private member of Lhs then instantiating has\_post\_increment<Lhs> will produce a compiler error. For this reason has\_post\_increment cannot be used to determine whether a type has a public operator++ or not.

```
struct A { private: void operator++(int); };  
boost::has_post_increment<A>::value; // error: A::operator++(int) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator++(const A&, int);  
struct B { operator A(); };  
boost::has_post_increment<A>::value; // this is fine  
boost::has_post_increment<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If operator++ is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_post_increment.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator++(const contains<T> &lhs, int) {
    return f(lhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_post_increment< contains< good > >::value<<'\\n'; // true
    contains<good> g;
    g++; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_post_increment< contains< bad > >::value<<'\\n'; // true, should be false

    contains<bad> b;
    b++; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_pre\_decrement

```

template <class Rhs, class Ret=dont_care>
struct has_pre_decrement : public true_type-or-false_type {};

```

**Inherits:** If (i) rhs of type Rhs can be used in expression --rhs, and (ii) Ret=dont\_care or the result of expression --rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of prefix operator--. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Rhs rhs;
f(--rhs); // is valid if has_pre_decrement<Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_pre\_decrement.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_pre\_decrement<Rhs, Ret>::value\_type is the type bool.

has\_pre\_decrement<Rhs, Ret>::value is a bool integral constant expression.

has\_pre\_decrement<int>::value is a bool integral constant expression that evaluates to true.

```
has_pre_decrement<long> inherits from true_type.  
  
has_pre_decrement<int, int> inherits from true_type.  
  
has_pre_decrement<int, long> inherits from true_type.  
  
has_pre_decrement<double, double> inherits from true_type.  
  
has_pre_decrement<double, int> inherits from true_type.  
  
has_pre_decrement<bool> inherits from false_type.  
  
has_pre_decrement<const int> inherits from false_type.  
  
has_pre_decrement<void*> inherits from false_type.  
  
has_pre_decrement<int, std::string> inherits from false_type.
```

See also: [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether prefix `operator--` is public or not: if `operator--` is defined as a private member of `Rhs` then instantiating `has_pre_decrement<Rhs>` will produce a compiler error. For this reason `has_pre_decrement` cannot be used to determine whether a type has a public `operator--` or not.

```
struct A { private: void operator--(); };  
boost::has_pre_decrement<A>::value; // error: A::operator--() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator--(const A&);  
struct B { operator A(); };  
boost::has_pre_decrement<A>::value; // this is fine  
boost::has_pre_decrement<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator--` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_pre_decrement.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator--(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_pre_decrement< contains< good > >::value<<'\n'; // true
    contains<good> g;
    --g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_pre_decrement< contains< bad > >::value<<'\n'; // true, should be false

    contains<bad> b;
    --b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_pre\_increment

```

template <class Rhs, class Ret=dont_care>
struct has_pre_increment : public true_type-or-false_type {};

```

**Inherits:** If (i) rhs of type Rhs can be used in expression ++rhs, and (ii) Ret=dont\_care or the result of expression ++rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of prefix operator++. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Rhs rhs;
f(++rhs); // is valid if has_pre_increment<Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_pre\_increment.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_pre\_increment<Rhs, Ret>::value\_type is the type bool.

has\_pre\_increment<Rhs, Ret>::value is a bool integral constant expression.

has\_pre\_increment<int>::value is a bool integral constant expression that evaluates to true.

```
has_pre_increment<long> inherits from true_type.  
  
has_pre_increment<int, int> inherits from true_type.  
  
has_pre_increment<int, long> inherits from true_type.  
  
has_pre_increment<double, double> inherits from true_type.  
  
has_pre_increment<double, int> inherits from true_type.  
  
has_pre_increment<bool> inherits from true_type.  
  
has_pre_increment<const int> inherits from false_type.  
  
has_pre_increment<void*> inherits from false_type.  
  
has_pre_increment<int, std::string> inherits from false_type.
```

See also: [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether prefix `operator++` is public or not: if `operator++` is defined as a private member of `Rhs` then instantiating `has_pre_increment<Rhs>` will produce a compiler error. For this reason `has_pre_increment` cannot be used to determine whether a type has a public `operator++` or not.

```
struct A { private: void operator++(); };  
boost::has_pre_increment<A>::value; // error: A::operator++() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };  
void operator++(const A&);  
struct B { operator A(); };  
boost::has_pre_increment<A>::value; // this is fine  
boost::has_pre_increment<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator++` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```

#include <boost/type_traits/has_pre_increment.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator++(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_pre_increment< contains< good > >::value<<'\n'; // true
    contains<good> g;
    ++g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_pre_increment< contains< bad > >::value<<'\n'; // true, should be false

    contains<bad> b;
    ++b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_right\_shift

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_right_shift : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs>>rhs, and (ii) Ret=dont\_care or the result of expression lhs>>rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator>>. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs>>rhs); // is valid if has_right_shift<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_right\_shift.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

has\_right\_shift<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_right\_shift<Lhs, Rhs, Ret>::value is a bool integral constant expression.

`has_right_shift<int>::value` is a bool integral constant expression that evaluates to true.

`has_right_shift<long>` inherits from `true_type`.

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

`has_right_shift<const int, int>` inherits from `true_type`.

`has_right_shift<std::istream, int&>` inherits from `true_type`.

`has_right_shift<std::istream, char*, std::ostream>` inherits from `true_type`.

`has_right_shift<std::istream, std::string&>` inherits from `true_type`.

`has_right_shift<int, double, bool>` inherits from `false_type`.

`has_right_shift<int, int, std::string>` inherits from `false_type`.

See also: [Operator Type Traits](#)

#### Limitation:

- Requires a compiler with working SFINAE.

#### Known issues:

- This trait cannot detect whether binary `operator>>` is public or not: if `operator>>` is defined as a private member of Lhs then instantiating `has_right_shift<Lhs>` will produce a compiler error. For this reason `has_right_shift` cannot be used to determine whether a type has a public `operator>>` or not.

```
struct A { private: void operator>>(const A&); };
boost::has_right_shift<A>::value; // error: A::operator>>(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator>>(const A&, const A&);
struct B { operator A(); };
boost::has_right_shift<A>::value; // this is fine
boost::has_right_shift<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator>>` is defined but does not bind for a given template type, it is still detected by the trait which returns true instead of false. Example:

```

#include <boost/type_traits/has_right_shift.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator>>(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_right_shift< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g>>g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_right_shift< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b>>b; // compile time error
    return 0;
}

```

- volatile qualifier is not properly handled and would lead to undefined behavior

## has\_right\_shift\_assign

```

template <class Lhs, class Rhs=Lhs, class Ret=dont_care>
struct has_right_shift_assign : public true_type-or-false_type {};

```

**Inherits:** If (i) lhs of type Lhs and rhs of type Rhs can be used in expression lhs>>=rhs, and (ii) Ret=dont\_care or the result of expression lhs>>=rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of binary operator>>=. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```

void f(Ret);
Lhs lhs;
Rhs rhs;
f(lhs>>=rhs); // is valid if has_right_shift_assign<Lhs, Rhs, Ret>::value==true

```

If Ret=void, the return type is checked to be exactly void.

**Header:** `#include <boost/type_traits/has_right_shift_assign.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

### Examples:

has\_right\_shift\_assign<Lhs, Rhs, Ret>::value\_type is the type bool.

has\_right\_shift\_assign<Lhs, Rhs, Ret>::value is a bool integral constant expression.

has\_right\_shift\_assign<int>::value is a bool integral constant expression that evaluates to true.

`has_right_shift_assign<long>` inherits from `true_type`.

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

`has_right_shift_assign<const int, int>` inherits from `false_type`.

`has_right_shift_assign<int, double, bool>` inherits from `false_type`.

`has_right_shift_assign<int, int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether binary `operator>>=` is public or not: if `operator>>=` is defined as a private member of Lhs then instantiating `has_right_shift_assign<Lhs>` will produce a compiler error. For this reason `has_right_shift_assign` cannot be used to determine whether a type has a public `operator>>=` or not.

```
struct A { private: void operator>>=(const A&); };
boost::has_right_shift_assign<A>::value; // error: A::operator>>=(const A&) is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator>>=(const A&, const A&);
struct B { operator A(); };
boost::has_right_shift_assign<A>::value; // this is fine
boost::has_right_shift_assign<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator>>=` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```
#include <boost/type_traits/has_right_shift_assign.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator>>=(const contains<T> &lhs, const contains<T> &rhs) {
    return f(lhs.data, rhs.data);
}

class bad { };
class good { };
bool f(const good&, const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_right_shift_assign< contains< good > >::value<<'\n'; // true
    contains<good> g;
    g>>=g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_right_shift_assign< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    b>>=b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## 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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_HAS_TRIVIAL_ASSIGN` is defined.

**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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_HAS_TRIVIAL_CONSTRUCTOR` is defined.

**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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_HAS_TRIVIAL_COPY` is defined.

**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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_HAS_TRIVIAL_DESTRUCTOR` is defined.

**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\_unary\_minus

```
template <class Rh, class Ret=dont_care>
struct has_unary_minus : public true_type-or-false_type {};
```

**Inherits:** If (i) `rhs` of type `Rhs` can be used in expression `-rhs`, and (ii) `Ret=dont_care` or the result of expression `-rhs` is convertible to `Ret` then inherits from `true_type`, otherwise inherits from `false_type`.

The default behaviour (`Ret=dont_care`) is to not check for the return value of prefix operator`-`. If `Ret` is different from the default `dont_care` type, the return value is checked to be convertible to `Ret`. Convertible to `Ret` means that the return value of the operator can be used as argument to a function expecting `Ret`:

```
void f(Ret);
Rh rhs;
f(-rhs); // is valid if has_unary_minus<Rh, Ret>::value==true
```

If `Ret=void`, the return type is checked to be exactly `void`.

**Header:** `#include <boost/type_traits/has_unary_minus.hpp>` or `#include <boost/type_traits/has_operator.hpp>` or `#include <boost/type_traits.hpp>`

**Examples:**

`has_unary_minus<Rh, Ret>::value_type` is the type `bool`.

`has_unary_minus<Rh, Ret>::value` is a `bool` integral constant expression.

`has_unary_minus<int>::value` is a `bool` integral constant expression that evaluates to *true*.

`has_unary_minus<long>` inherits from `true_type`.

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

`has_unary_minus<int, long>` inherits from `true_type`.

`has_unary_minus<double, double>` inherits from `true_type`.

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

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

`has_unary_minus<int, std::string>` inherits from `false_type`.

**See also:** [Operator Type Traits](#)

**Limitation:**

- Requires a compiler with working SFINAE.

**Known issues:**

- This trait cannot detect whether prefix `operator-` is public or not: if `operator-` is defined as a private member of `Rhs` then instantiating `has_unary_minus<Rhs>` will produce a compiler error. For this reason `has_unary_minus` cannot be used to determine whether a type has a public `operator-` or not.

```
struct A { private: void operator-(); };
boost::has_unary_minus<A>::value; // error: A::operator-() is private
```

- There is an issue if the operator exists only for type `A` and `B` is convertible to `A`. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator-(const A&);
struct B { operator A(); };
boost::has_unary_minus<A>::value; // this is fine
boost::has_unary_minus<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator-` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```
#include <boost/type_traits/has_unary_minus.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator-(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_unary_minus< contains< good > >::value<<'\n'; // true
    contains<good> g;
    -g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_unary_minus< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    -b; // compile time error
    return 0;
}
```

- `volatile` qualifier is not properly handled and would lead to undefined behavior

## has\_unary\_plus

```
template <class Rhs, class Ret=dont_care>
struct has_unary_plus : public true_type-or-false_type {};
```

**Inherits:** If (i) rhs of type Rhs can be used in expression +rhs, and (ii) Ret=dont\_care or the result of expression +rhs is convertible to Ret then inherits from [true\\_type](#), otherwise inherits from [false\\_type](#).

The default behaviour (Ret=dont\_care) is to not check for the return value of prefix operator+. If Ret is different from the default dont\_care type, the return value is checked to be convertible to Ret. Convertible to Ret means that the return value of the operator can be used as argument to a function expecting Ret:

```
void f(Ret);
Rhs rhs;
f(+rhs); // is valid if has_unary_plus<Rhs, Ret>::value==true
```

If Ret=void, the return type is checked to be exactly void.

**Header:** #include <boost/type\_traits/has\_unary\_plus.hpp> or #include <boost/type\_traits/has\_operator.hpp> or #include <boost/type\_traits.hpp>

### Examples:

```
has_unary_plus<Rhs, Ret>::value_type is the type bool.

has_unary_plus<Rhs, Ret>::value is a bool integral constant expression.

has_unary_plus<int>::value is a bool integral constant expression that evaluates to true.

has_unary_plus<long> inherits from true_type.

has_unary_plus<int, int> inherits from true_type.

has_unary_plus<int, long> inherits from true_type.

has_unary_plus<double, double> inherits from true_type.

has_unary_plus<double, int> inherits from true_type.

has_unary_plus<const int> inherits from true_type.

has_unary_plus<int, std::string> inherits from false_type.
```

**See also:** [Operator Type Traits](#)

### Limitation:

- Requires a compiler with working SFINAE.

### Known issues:

- This trait cannot detect whether prefix operator+ is public or not: if operator+ is defined as a private member of Rhs then instantiating has\_unary\_plus<Rhs> will produce a compiler error. For this reason has\_unary\_plus cannot be used to determine whether a type has a public operator+ or not.

```
struct A { private: void operator+(); };
boost::has_unary_plus<A>::value; // error: A::operator+() is private
```

- There is an issue if the operator exists only for type A and B is convertible to A. In this case, the compiler will report an ambiguous overload.

```
struct A { };
void operator+(const A&);
struct B { operator A(); };
boost::has_unary_plus<A>::value; // this is fine
boost::has_unary_plus<B>::value; // error: ambiguous overload
```

- There is an issue when applying this trait to template classes. If `operator+` is defined but does not bind for a given template type, it is still detected by the trait which returns `true` instead of `false`. Example:

```
#include <boost/type_traits/has_unary_plus.hpp>
#include <iostream>

template <class T>
struct contains { T data; };

template <class T>
bool operator+(const contains<T> &rhs) {
    return f(rhs.data);
}

class bad { };
class good { };
bool f(const good&) { }

int main() {
    std::cout<<std::boolalpha;
    // works fine for contains<good>
    std::cout<<boost::has_unary_plus< contains< good > >::value<<'\n'; // true
    contains<good> g;
    +g; // ok
    // does not work for contains<bad>
    std::cout<<boost::has_unary_plus< contains< bad > >::value<<'\n'; // true, should be false
    contains<bad> b;
    +b; // compile time error
    return 0;
}
```

- volatile qualifier is not properly handled and would lead to undefined behavior

## 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 [intrinsic](#)s that enable the trait to do the right thing: Currently (May 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsic](#)s to ensure that this trait "just works". You may also test to see if the necessary [intrinsic](#)s are available by checking to see if the macro `BOOST_HAS_VIRTUAL_DESTRUCTOR` is defined.

**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 21. Examples**

Expression	Result Type
<code>integral_promotion&lt;short const&gt;::type</code>	<code>int const</code>
<code>integral_promotion&lt;short&amp;&gt;::type</code>	<code>short&amp;</code>
<code>integral_promotion&lt;enum std::float_round_style&gt;::type</code>	<code>int</code>

## 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 class type 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 inherit from [true\\_type](#) if X is a class type. This is a change in behaviour from Boost-1.39.0 in order to track the emerging C++0x standard.

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 (and not a union 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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsic](#)s to ensure

that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_IS_CLASS` is defined.

**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 rvalue 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 `true`.

`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 (and not a union 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 - this latter case can be tested for by checking to see if the macro `BOOST_IS_EMPTY` is defined.

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\_lvalue\_reference

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

**Inherits:** If T is an lvalue reference 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_lvalue_reference.hpp>` or `#include <boost/type_traits.hpp>`

**Examples:**

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

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

`is_lvalue_reference<int const&&>::type` is the type `false_type`.

`is_lvalue_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_lvalue_reference<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, `is_pod` will never report that a class or struct is a POD; this is always safe, if possibly sub-optimal. Currently (May 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsic](#)s to ensure that this trait "just works". You may also test to see if the necessary [intrinsic](#)s are available by checking to see if the macro `BOOST_IS_POD` is defined.

**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\_reference

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

**Inherits:** If T is a reference type (either an lvalue reference or an rvalue reference) 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 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\_rvalue\_reference

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

**Inherits:** If T is an rvalue reference 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_rvalue_reference.hpp>` or `#include <boost/type_traits.hpp>`

**Examples:**

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

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

`is_rvalue_reference<int const>::type` is the type `false_type`.

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

`is_rvalue_reference<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:** If `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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works".

## 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 2011) compilers more recent than Visual C++ 8, GCC-4.3, Greenhills 6.0, Intel-11.0, and Codegear have the necessary compiler [intrinsics](#) to ensure that this trait "just works". You may also test to see if the necessary [intrinsics](#) are available by checking to see if the macro `BOOST_IS_UNION` is defined.

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

### Examples:

Given `union my_union {};` then:

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

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

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

`is_union<my_union*>::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.



## Note

There are a small number of cases where it's simply not possible for this trait to work, and where attempting to instantiate the trait will cause compiler errors (see bug report [#3730](#)). Further more the issues may well be compiler specific. In this situation the user should supply a full specialization of the trait to work around the problem.

### 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<SomeClassType, SomeClassType>::value` is an integral constant expression that evaluates to `true`.

`is_virtual_base_of<NotAClassType, NotAClassType>::value` is an integral constant expression that evaluates to `false`.

`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 22. Examples**

Expression	Result Type
<code>make_signed&lt;int&gt;::type</code>	<code>int</code>
<code>make_signed&lt;unsigned int const&gt;::type</code>	<code>int const</code>
<code>make_signed&lt;const unsigned long long&gt;::type</code>	<code>const long long</code>
<code>make_signed&lt;my_enum&gt;::type</code>	A signed integer type with the same width as the enum.
<code>make_signed&lt;wchar_t&gt;::type</code>	A signed integer type with the same width as <code>wchar_t</code> .

## 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 a 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 23. Examples**

Expression	Result Type
<code>make_unsigned&lt;int&gt;::type</code>	<code>unsigned int</code>
<code>make_unsigned&lt;unsigned int const&gt;::type</code>	<code>unsigned int const</code>
<code>make_unsigned&lt;const unsigned long long&gt;::type</code>	<code>const unsigned long long</code>
<code>make_unsigned&lt;my_enum&gt;::type</code>	An unsigned integer type with the same width as the enum.
<code>make_unsigned&lt;wchar_t&gt;::type</code>	An unsigned integer type with the same width as <code>wchar_t</code> .

## 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 24. Examples**

Expression	Result Type
<code>promote&lt;short volatile&gt;::type</code>	<code>int volatile</code>
<code>promote&lt;float const&gt;::type</code>	<code>double const</code>
<code>promote&lt;short&amp;&gt;::type</code>	<code>short&amp;</code>

## 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 a (built-in) 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<boost::array<int, 3>>::value` is an integral constant expression that evaluates to `0`: `boost::array` is a class type and **not an array type**!

`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 25. Examples**

Expression	Result Type
<code>remove_all_extents&lt;int&gt;::type</code>	<code>int</code>
<code>remove_all_extents&lt;int const[2]&gt;::type</code>	<code>int const</code>
<code>remove_all_extents&lt;int[][2]&gt;::type</code>	<code>int</code>
<code>remove_all_extents&lt;int[2][3][4]&gt;::type</code>	<code>int</code>
<code>remove_all_extents&lt;int const*&gt;::type</code>	<code>int const*</code>

## 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 26. Examples**

Expression	Result Type
<code>remove_const&lt;int&gt;::type</code>	<code>int</code>
<code>remove_const&lt;int const&gt;::type</code>	<code>int</code>
<code>remove_const&lt;int const volatile&gt;::type</code>	<code>int volatile</code>
<code>remove_const&lt;int const&amp;&gt;::type</code>	<code>int const&amp;</code>
<code>remove_const&lt;int const*&gt;::type</code>	<code>int const*</code>

## 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 27. Examples**

Expression	Result Type
<code>remove_cv&lt;int&gt;::type</code>	<code>int</code>
<code>remove_cv&lt;int const&gt;::type</code>	<code>int</code>
<code>remove_cv&lt;int const volatile&gt;::type</code>	<code>int</code>
<code>remove_cv&lt;int const&amp;&gt;::type</code>	<code>int const&amp;</code>
<code>remove_cv&lt;int const*&gt;::type</code>	<code>int const*</code>

## 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 28. Examples**

Expression	Result Type
<code>remove_extent&lt;int&gt;::type</code>	<code>int</code>
<code>remove_extent&lt;int const[2]&gt;::type</code>	<code>int const</code>
<code>remove_extent&lt;int[2][4]&gt;::type</code>	<code>int[4]</code>
<code>remove_extent&lt;int[][2]&gt;::type</code>	<code>int[2]</code>
<code>remove_extent&lt;int const*&gt;::type</code>	<code>int const*</code>

## remove\_pointer

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

**type:** The same type as T, but with any pointer modifier removed. Note that pointers to members are left unchanged: removing the pointer decoration would result in an invalid type.

**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 29. Examples**

Expression	Result Type
<code>remove_pointer&lt;int&gt;::type</code>	<code>int</code>
<code>remove_pointer&lt;int const*&gt;::type</code>	<code>int const</code>
<code>remove_pointer&lt;int const**&gt;::type</code>	<code>int const*</code>
<code>remove_pointer&lt;int&amp;&gt;::type</code>	<code>int&amp;</code>
<code>remove_pointer&lt;int*&amp;&gt;::type</code>	<code>int*&amp;</code>

## 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 30. Examples**

Expression	Result Type
<code>remove_reference&lt;int&gt;::type</code>	<code>int</code>
<code>remove_reference&lt;int const&amp;&gt;::type</code>	<code>int const</code>
<code>remove_reference&lt;int&amp;&amp;&gt;::type</code>	<code>int</code>
<code>remove_reference&lt;int*&gt;::type</code>	<code>int*</code>
<code>remove_reference&lt;int*&amp;&gt;::type</code>	<code>int*</code>

## 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 31. Examples**

Expression	Result Type
<code>remove_volatile&lt;int&gt;::type</code>	<code>int</code>
<code>remove_volatile&lt;int volatile&gt;::type</code>	<code>int</code>
<code>remove_volatile&lt;int const volatile&gt;::type</code>	<code>int const</code>
<code>remove_volatile&lt;int volatile&amp;&gt;::type</code>	<code>int const&amp;</code>
<code>remove_volatile&lt;int volatile*&gt;::type</code>	<code>int const*</code>

## 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 1.47.0

- **Breaking change:** changed [is\\_convertible](#) to C++0x behaviour when possible.
- Fixed issues [#5271](#), [#4530](#).

## Boost 1.45.0

- Added new traits [add\\_rvalue\\_reference](#), [add\\_lvalue\\_reference](#) and [common\\_type](#).
- Minor fixes to [is\\_signed](#), [is\\_unsigned](#) and [is\\_virtual\\_base\\_of](#).

## Boost 1.44.0

- Added support for rvalue references throughout the library, plus two new traits classes [is\\_rvalue\\_reference](#) and [is\\_lvalue\\_reference](#). Fixes [#4407](#) and [#3804](#).
- Fixed ticket [#3621](#).

## 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](http://www.boost.org). Bugs, suggestions and discussion should be directed to [boost@lists.boost.org](mailto:boost@lists.boost.org) (see [www.boost.org/more/mailling\\_lists.htm#main](http://www.boost.org/more/mailling_lists.htm#main) for subscription details).

## Class Index

### A

- add\_const
  - add\_const, 37
- add\_cv
  - add\_cv, 37
- add\_lvalue\_reference
  - add\_lvalue\_reference, 38
- add\_pointer
  - add\_pointer, 38
- add\_reference
  - add\_reference, 39
- add\_rvalue\_reference
  - add\_rvalue\_reference, 40
- add\_volatile
  - add\_volatile, 40
- aligned\_storage
  - aligned\_storage, 41
- alignment\_of
  - alignment\_of, 41
  - integral\_constant, 41
- any
  - Operator Type Traits, 14

### B

- Background and Tutorial
  - is\_pointer, 5
  - is\_void, 5
  - remove\_extent, 5
- BOOST\_ALIGNMENT\_OF
  - Macros for Compiler Intrinsics, 29
- BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION
  - Type Traits that Transform One Type to Another, 25
- BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEOF

- common\_type, 42
- BOOST\_HAS\_NOTHROW\_ASSIGN
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_NOTHROW\_CONSTRUCTOR
  - has\_nothrow\_constructor, 105
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_NOTHROW\_COPY
  - has\_nothrow\_copy, 105
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_TRIVIAL\_ASSIGN
  - has\_trivial\_assign, 121
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR
  - has\_trivial\_constructor, 122
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_TRIVIAL\_COPY
  - has\_trivial\_copy, 122
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_TRIVIAL\_DESTRUCTOR
  - has\_trivial\_destructor, 123
  - Macros for Compiler Intrinsics, 29
- BOOST\_HAS\_VIRTUAL\_DESTRUCTOR
  - has\_virtual\_destructor, 127
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_ABSTRACT
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_BASE\_OF
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_CLASS
  - is\_class, 130
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_CONVERTIBLE
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_EMPTY
  - is\_empty, 133
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_ENUM
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_POD
  - is\_pod, 139
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_POLYMORPHIC
  - Macros for Compiler Intrinsics, 29
- BOOST\_IS\_UNION
  - is\_union, 143
  - Macros for Compiler Intrinsics, 29

## C

- check
  - Operator Type Traits, 14
- common\_type
  - BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEDEF, 42
  - common\_type, 42
  - Improving std::min with common\_type, 35

## D

- decay

- decay, 45
- dont\_care
  - has\_bit\_and, 48
  - has\_bit\_and\_assign, 50
  - has\_bit\_or, 52
  - has\_bit\_or\_assign, 54
  - has\_bit\_xor, 56
  - has\_bit\_xor\_assign, 58
  - has\_complement, 60
  - has\_dereference, 62
  - has\_divides, 64
  - has\_divides\_assign, 66
  - has\_equal\_to, 68
  - has\_greater, 70
  - has\_greater\_equal, 72
  - has\_left\_shift, 74
  - has\_left\_shift\_assign, 76
  - has\_less, 78
  - has\_less\_equal, 80
  - has\_logical\_and, 82
  - has\_logical\_not, 84
  - has\_logical\_or, 86
  - has\_minus, 88
  - has\_minus\_assign, 90
  - has\_modulus, 92
  - has\_modulus\_assign, 94
  - has\_multiplies, 96
  - has\_multiplies\_assign, 98
  - has\_negate, 100
  - has\_not\_equal\_to, 103
  - has\_plus, 106
  - has\_plus\_assign, 108
  - has\_post\_decrement, 109
  - has\_post\_increment, 111
  - has\_pre\_decrement, 113
  - has\_pre\_increment, 115
  - has\_right\_shift, 117
  - has\_right\_shift\_assign, 119
  - has\_unary\_minus, 124
  - has\_unary\_plus, 126
- Operator Type Traits, 14

## E

- extent
  - extent, 46
  - integral\_constant, 46

## F

- false\_type
  - integral\_constant, 128
- floating\_point\_promotion
  - floating\_point\_promotion, 47
- function\_traits
  - function\_traits, 47
  - result\_type, 47

## H

- has\_bit\_and
  - dont\_care, 48
  - has\_bit\_and, 48-49
  - trait, 49
- has\_bit\_and\_assign
  - dont\_care, 50
  - has\_bit\_and\_assign, 50-51
  - trait, 51
- has\_bit\_or
  - dont\_care, 52
  - has\_bit\_or, 52-53
  - trait, 53
- has\_bit\_or\_assign
  - dont\_care, 54
  - has\_bit\_or\_assign, 54-55
  - trait, 55
- has\_bit\_xor
  - dont\_care, 56
  - has\_bit\_xor, 56-57
  - trait, 57
- has\_bit\_xor\_assign
  - dont\_care, 58
  - has\_bit\_xor\_assign, 58-59
  - trait, 59
- has\_complement
  - dont\_care, 60
  - has\_complement, 60-61
  - trait, 61
- has\_dereference
  - dont\_care, 62
  - has\_dereference, 62-63
  - trait, 63
- has\_divides
  - dont\_care, 64
  - has\_divides, 64-65
  - trait, 65
- has\_divides\_assign
  - dont\_care, 66
  - has\_divides\_assign, 66-67
  - trait, 67
- has\_equal\_to
  - dont\_care, 68
  - has\_equal\_to, 68-69
  - Operator Type Traits, 24
  - trait, 69
- has\_greater
  - dont\_care, 70
  - has\_greater, 70-71
  - trait, 71
- has\_greater\_equal
  - dont\_care, 72
  - has\_greater\_equal, 72-73
  - trait, 73
- has\_left\_shift
  - dont\_care, 74
  - has\_left\_shift, 74-75

- trait, 75
- has\_left\_shift\_assign
  - dont\_care, 76
  - has\_left\_shift\_assign, 76-77
  - trait, 77
- has\_less
  - dont\_care, 78
  - has\_less, 78-79
  - trait, 79
- has\_less\_equal
  - dont\_care, 80
  - has\_less\_equal, 80-81
  - trait, 81
- has\_logical\_and
  - dont\_care, 82
  - has\_logical\_and, 82-83
  - trait, 83
- has\_logical\_not
  - dont\_care, 84
  - has\_logical\_not, 84-85
  - trait, 85
- has\_logical\_or
  - dont\_care, 86
  - has\_logical\_or, 86-87
  - trait, 87
- has\_minus
  - dont\_care, 88
  - has\_minus, 88-89
  - trait, 89
- has\_minus\_assign
  - dont\_care, 90
  - has\_minus\_assign, 90-91
  - trait, 91
- has\_modulus
  - dont\_care, 92
  - has\_modulus, 92-93
  - trait, 93
- has\_modulus\_assign
  - dont\_care, 94
  - has\_modulus\_assign, 94-95
  - trait, 95
- has\_multiplies
  - dont\_care, 96
  - has\_multiplies, 96-97
  - trait, 97
- has\_multiplies\_assign
  - dont\_care, 98
  - has\_multiplies\_assign, 98-99
  - trait, 99
- has\_negate
  - dont\_care, 100
  - has\_negate, 100-101
  - trait, 101
- has\_new\_operator
  - has\_new\_operator, 102
- has\_nothrow\_assign
  - has\_nothrow\_assign, 104
- has\_nothrow\_constructor

- BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 105
  - has\_nothrow\_constructor, 105
  - has\_nothrow\_default\_constructor, 105
- has\_nothrow\_copy
  - BOOST\_HAS\_NOTHROW\_COPY, 105
  - has\_nothrow\_copy, 105
  - has\_nothrow\_copy\_constructor, 105
- has\_nothrow\_copy\_constructor
  - has\_nothrow\_copy, 105
- has\_nothrow\_default\_constructor
  - has\_nothrow\_constructor, 105
- has\_not\_equal\_to
  - dont\_care, 103
  - has\_not\_equal\_to, 103-104
  - trait, 104
- has\_operator
  - Operator Type Traits, 14
- has\_plus
  - dont\_care, 106
  - has\_plus, 106-107
  - trait, 107
- has\_plus\_assign
  - dont\_care, 108
  - has\_plus\_assign, 108-109
  - trait, 109
- has\_post\_decrement
  - dont\_care, 109
  - has\_post\_decrement, 109-110
  - trait, 110
- has\_post\_increment
  - dont\_care, 111
  - has\_post\_increment, 111-112
  - trait, 112
- has\_pre\_decrement
  - dont\_care, 113
  - has\_pre\_decrement, 113-114
  - trait, 114
- has\_pre\_increment
  - dont\_care, 115
  - has\_pre\_increment, 115-116
  - trait, 116
- has\_right\_shift
  - dont\_care, 117
  - has\_right\_shift, 117-118
  - trait, 118
- has\_right\_shift\_assign
  - dont\_care, 119
  - has\_right\_shift\_assign, 119-120
  - trait, 120
- has\_trivial\_assign
  - BOOST\_HAS\_TRIVIAL\_ASSIGN, 121
  - has\_trivial\_assign, 121
- has\_trivial\_constructor
  - BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 122
  - has\_trivial\_constructor, 122
  - has\_trivial\_default\_constructor, 122
- has\_trivial\_copy
  - BOOST\_HAS\_TRIVIAL\_COPY, 122

- has\_trivial\_copy, 122
- has\_trivial\_copy\_constructor, 122
- has\_trivial\_copy\_constructor
  - has\_trivial\_copy, 122
- has\_trivial\_default\_constructor
  - has\_trivial\_constructor, 122
- has\_trivial\_destructor
  - BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 123
  - has\_trivial\_destructor, 123
- has\_unary\_minus
  - dont\_care, 124
  - has\_unary\_minus, 124-125
  - Operator Type Traits, 14
  - trait, 125
- has\_unary\_plus
  - dont\_care, 126
  - has\_unary\_plus, 126-127
  - trait, 127
- has\_virtual\_destructor
  - BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 127
  - has\_virtual\_destructor, 127

## I

- Improving std::min with common\_type
  - common\_type, 35
- int
  - is\_function, 134
- integral\_constant
  - alignment\_of, 41
  - extent, 46
  - false\_type, 128
  - integral\_constant, 128
  - Operator Type Traits, 14
  - rank, 147
  - true\_type, 128
- integral\_promotion
  - integral\_promotion, 128
- is\_abstract
  - is\_abstract, 128
- is\_arithmetic
  - is\_arithmetic, 129
- is\_array
  - is\_array, 129
- is\_base\_of
  - is\_base\_of, 130
- is\_class
  - BOOST\_IS\_CLASS, 130
  - is\_class, 130
  - User Defined Specializations, 28
- is\_complex
  - is\_complex, 131
- is\_compound
  - is\_compound, 131
- is\_const
  - is\_const, 132
- is\_convertible
  - is\_convertible, 132

- is\_convertible\_to\_Ret
  - Operator Type Traits, 14
- is\_empty
  - BOOST\_IS\_EMPTY, 133
  - is\_empty, 133
- is\_enum
  - is\_enum, 134
- is\_floating\_point
  - is\_floating\_point, 134
- is\_function
  - int, 134
  - is\_function, 134
- is\_fundamental
  - is\_fundamental, 136
- is\_integral
  - is\_integral, 136
- is\_lvalue\_reference
  - is\_lvalue\_reference, 136
- is\_member\_function\_pointer
  - is\_member\_function\_pointer, 137
- is\_member\_object\_pointer
  - is\_member\_object\_pointer, 137
- is\_member\_pointer
  - is\_member\_pointer, 138
- is\_object
  - is\_object, 138
- is\_pod
  - BOOST\_IS\_POD, 139
  - is\_pod, 139
  - User Defined Specializations, 28
- is\_pointer
  - Background and Tutorial, 5
  - is\_pointer, 139
- is\_polymorphic
  - is\_polymorphic, 140
- is\_reference
  - is\_reference, 140
- is\_rvalue\_reference
  - is\_rvalue\_reference, 141
- is\_same
  - is\_same, 141
- is\_scalar
  - is\_scalar, 142
- is\_signed
  - is\_signed, 142
- is\_stateless
  - is\_stateless, 143
- is\_union
  - BOOST\_IS\_UNION, 143
  - is\_union, 143
  - User Defined Specializations, 28
- is\_unsigned
  - is\_unsigned, 144
- is\_virtual\_base\_of
  - is\_virtual\_base\_of, 144
- is\_void
  - Background and Tutorial, 5
  - is\_void, 145

is\_volatile  
    is\_volatile, 145

## M

### Macros for Compiler Intrinsics

BOOST\_ALIGNMENT\_OF, 29  
BOOST\_HAS\_NOTHROW\_ASSIGN, 29  
BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 29  
BOOST\_HAS\_NOTHROW\_COPY, 29  
BOOST\_HAS\_TRIVIAL\_ASSIGN, 29  
BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 29  
BOOST\_HAS\_TRIVIAL\_COPY, 29  
BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 29  
BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 29  
BOOST\_IS\_ABSTRACT, 29  
BOOST\_IS\_BASE\_OF, 29  
BOOST\_IS\_CLASS, 29  
BOOST\_IS\_CONVERTIBLE, 29  
BOOST\_IS\_EMPTY, 29  
BOOST\_IS\_ENUM, 29  
BOOST\_IS\_POD, 29  
BOOST\_IS\_POLYMORPHIC, 29  
BOOST\_IS\_UNION, 29

make\_signed  
    make\_signed, 146  
make\_unsigned  
    make\_unsigned, 146

## N

no\_operator  
    Operator Type Traits, 14

## O

### Operator Type Traits

any, 14  
check, 14  
dont\_care, 14  
has\_equal\_to, 24  
has\_operator, 14  
has\_unary\_minus, 14  
integral\_constant, 14  
is\_convertible\_to\_Ret, 14  
no\_operator, 14  
operator\_exists, 14  
operator\_returns\_Ret, 14  
operator\_returns\_void, 14  
returns\_void, 14  
returns\_void\_t, 14  
Rhs\_nocv, 14  
Rhs\_noptr, 14  
Rhs\_noref, 14  
trait, 24  
trait\_impl, 14  
trait\_impl1, 14  
operator\_exists  
    Operator Type Traits, 14  
operator\_returns\_Ret

Operator Type Traits, 14  
operator\_returns\_void  
Operator Type Traits, 14

## P

promote  
promote, 147

## R

rank  
integral\_constant, 147  
rank, 147  
remove\_all\_extents  
remove\_all\_extents, 148  
remove\_const  
remove\_const, 148  
remove\_cv  
remove\_cv, 149  
remove\_extent  
Background and Tutorial, 5  
remove\_extent, 150  
remove\_pointer  
remove\_pointer, 150  
remove\_reference  
remove\_reference, 151  
remove\_volatile  
remove\_volatile, 151  
result\_type  
function\_traits, 47  
returns\_void  
Operator Type Traits, 14  
returns\_void\_t  
Operator Type Traits, 14  
Rhs\_nocv  
Operator Type Traits, 14  
Rhs\_noptr  
Operator Type Traits, 14  
Rhs\_noref  
Operator Type Traits, 14

## T

trait  
has\_bit\_and, 49  
has\_bit\_and\_assign, 51  
has\_bit\_or, 53  
has\_bit\_or\_assign, 55  
has\_bit\_xor, 57  
has\_bit\_xor\_assign, 59  
has\_complement, 61  
has\_dereference, 63  
has\_divides, 65  
has\_divides\_assign, 67  
has\_equal\_to, 69  
has\_greater, 71  
has\_greater\_equal, 73  
has\_left\_shift, 75  
has\_left\_shift\_assign, 77

- has\_less, 79
- has\_less\_equal, 81
- has\_logical\_and, 83
- has\_logical\_not, 85
- has\_logical\_or, 87
- has\_minus, 89
- has\_minus\_assign, 91
- has\_modulus, 93
- has\_modulus\_assign, 95
- has\_multiplies, 97
- has\_multiplies\_assign, 99
- has\_negate, 101
- has\_not\_equal\_to, 104
- has\_plus, 107
- has\_plus\_assign, 109
- has\_post\_decrement, 110
- has\_post\_increment, 112
- has\_pre\_decrement, 114
- has\_pre\_increment, 116
- has\_right\_shift, 118
- has\_right\_shift\_assign, 120
- has\_unary\_minus, 125
- has\_unary\_plus, 127
- Operator Type Traits, 24
- trait\_impl
  - Operator Type Traits, 14
- trait\_impl1
  - Operator Type Traits, 14
- true\_type
  - integral\_constant, 128
- Type Traits that Transform One Type to Another
  - BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION, 25
- type\_with\_alignment
  - type\_with\_alignment, 152

## U

User Defined Specializations

- is\_class, 28
- is\_pod, 28
- is\_union, 28

# Typedef Index

## A

- add\_const
  - add\_const, 37
- add\_cv
  - add\_cv, 37
- add\_lvalue\_reference
  - add\_lvalue\_reference, 38
- add\_pointer
  - add\_pointer, 38
- add\_reference
  - add\_reference, 39
- add\_rvalue\_reference
  - add\_rvalue\_reference, 40

add\_volatile  
  add\_volatile, 40  
aligned\_storage  
  aligned\_storage, 41  
alignment\_of  
  alignment\_of, 41  
  integral\_constant, 41  
any  
  Operator Type Traits, 14

## B

Background and Tutorial  
  is\_pointer, 5  
  is\_void, 5  
  remove\_extent, 5  
BOOST\_ALIGNMENT\_OF  
  Macros for Compiler Intrinsics, 29  
BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION  
  Type Traits that Transform One Type to Another, 25  
BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEOF  
  common\_type, 42  
BOOST\_HAS\_NOTHROW\_ASSIGN  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_NOTHROW\_CONSTRUCTOR  
  has\_nothrow\_constructor, 105  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_NOTHROW\_COPY  
  has\_nothrow\_copy, 105  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_ASSIGN  
  has\_trivial\_assign, 121  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR  
  has\_trivial\_constructor, 122  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_COPY  
  has\_trivial\_copy, 122  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_DESTRUCTOR  
  has\_trivial\_destructor, 123  
  Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_VIRTUAL\_DESTRUCTOR  
  has\_virtual\_destructor, 127  
  Macros for Compiler Intrinsics, 29  
BOOST\_IS\_ABSTRACT  
  Macros for Compiler Intrinsics, 29  
BOOST\_IS\_BASE\_OF  
  Macros for Compiler Intrinsics, 29  
BOOST\_IS\_CLASS  
  is\_class, 130  
  Macros for Compiler Intrinsics, 29  
BOOST\_IS\_CONVERTIBLE  
  Macros for Compiler Intrinsics, 29  
BOOST\_IS\_EMPTY  
  is\_empty, 133  
  Macros for Compiler Intrinsics, 29  
BOOST\_IS\_ENUM

Macros for Compiler Intrinsics, 29

BOOST\_IS\_POD

is\_pod, 139

Macros for Compiler Intrinsics, 29

BOOST\_IS\_POLYMORPHIC

Macros for Compiler Intrinsics, 29

BOOST\_IS\_UNION

is\_union, 143

Macros for Compiler Intrinsics, 29

## C

check

Operator Type Traits, 14

common\_type

BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEOF, 42

common\_type, 42

Improving std::min with common\_type, 35

## D

decay

decay, 45

dont\_care

has\_bit\_and, 48

has\_bit\_and\_assign, 50

has\_bit\_or, 52

has\_bit\_or\_assign, 54

has\_bit\_xor, 56

has\_bit\_xor\_assign, 58

has\_complement, 60

has\_dereference, 62

has\_divides, 64

has\_divides\_assign, 66

has\_equal\_to, 68

has\_greater, 70

has\_greater\_equal, 72

has\_left\_shift, 74

has\_left\_shift\_assign, 76

has\_less, 78

has\_less\_equal, 80

has\_logical\_and, 82

has\_logical\_not, 84

has\_logical\_or, 86

has\_minus, 88

has\_minus\_assign, 90

has\_modulus, 92

has\_modulus\_assign, 94

has\_multiplies, 96

has\_multiplies\_assign, 98

has\_negate, 100

has\_not\_equal\_to, 103

has\_plus, 106

has\_plus\_assign, 108

has\_post\_decrement, 109

has\_post\_increment, 111

has\_pre\_decrement, 113

has\_pre\_increment, 115

has\_right\_shift, 117

- has\_right\_shift\_assign, 119
- has\_unary\_minus, 124
- has\_unary\_plus, 126
- Operator Type Traits, 14

## E

extent

- extent, 46
- integral\_constant, 46

## F

false\_type

- integral\_constant, 128

floating\_point\_promotion

- floating\_point\_promotion, 47

function\_traits

- function\_traits, 47
- result\_type, 47

## H

has\_bit\_and

- dont\_care, 48
- has\_bit\_and, 48-49
- trait, 49

has\_bit\_and\_assign

- dont\_care, 50
- has\_bit\_and\_assign, 50-51
- trait, 51

has\_bit\_or

- dont\_care, 52
- has\_bit\_or, 52-53
- trait, 53

has\_bit\_or\_assign

- dont\_care, 54
- has\_bit\_or\_assign, 54-55
- trait, 55

has\_bit\_xor

- dont\_care, 56
- has\_bit\_xor, 56-57
- trait, 57

has\_bit\_xor\_assign

- dont\_care, 58
- has\_bit\_xor\_assign, 58-59
- trait, 59

has\_complement

- dont\_care, 60
- has\_complement, 60-61
- trait, 61

has\_dereference

- dont\_care, 62
- has\_dereference, 62-63
- trait, 63

has\_divides

- dont\_care, 64
- has\_divides, 64-65
- trait, 65

has\_divides\_assign

- dont\_care, 66
- has\_divides\_assign, 66-67
- trait, 67
- has\_equal\_to
  - dont\_care, 68
  - has\_equal\_to, 68-69
  - Operator Type Traits, 24
  - trait, 69
- has\_greater
  - dont\_care, 70
  - has\_greater, 70-71
  - trait, 71
- has\_greater\_equal
  - dont\_care, 72
  - has\_greater\_equal, 72-73
  - trait, 73
- has\_left\_shift
  - dont\_care, 74
  - has\_left\_shift, 74-75
  - trait, 75
- has\_left\_shift\_assign
  - dont\_care, 76
  - has\_left\_shift\_assign, 76-77
  - trait, 77
- has\_less
  - dont\_care, 78
  - has\_less, 78-79
  - trait, 79
- has\_less\_equal
  - dont\_care, 80
  - has\_less\_equal, 80-81
  - trait, 81
- has\_logical\_and
  - dont\_care, 82
  - has\_logical\_and, 82-83
  - trait, 83
- has\_logical\_not
  - dont\_care, 84
  - has\_logical\_not, 84-85
  - trait, 85
- has\_logical\_or
  - dont\_care, 86
  - has\_logical\_or, 86-87
  - trait, 87
- has\_minus
  - dont\_care, 88
  - has\_minus, 88-89
  - trait, 89
- has\_minus\_assign
  - dont\_care, 90
  - has\_minus\_assign, 90-91
  - trait, 91
- has\_modulus
  - dont\_care, 92
  - has\_modulus, 92-93
  - trait, 93
- has\_modulus\_assign
  - dont\_care, 94

- has\_modulus\_assign, 94-95
  - trait, 95
- has\_multiplies
  - dont\_care, 96
  - has\_multiplies, 96-97
  - trait, 97
- has\_multiplies\_assign
  - dont\_care, 98
  - has\_multiplies\_assign, 98-99
  - trait, 99
- has\_negate
  - dont\_care, 100
  - has\_negate, 100-101
  - trait, 101
- has\_new\_operator
  - has\_new\_operator, 102
- has\_nothrow\_assign
  - has\_nothrow\_assign, 104
- has\_nothrow\_constructor
  - BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 105
  - has\_nothrow\_constructor, 105
  - has\_nothrow\_default\_constructor, 105
- has\_nothrow\_copy
  - BOOST\_HAS\_NOTHROW\_COPY, 105
  - has\_nothrow\_copy, 105
  - has\_nothrow\_copy\_constructor, 105
- has\_nothrow\_copy\_constructor
  - has\_nothrow\_copy, 105
- has\_nothrow\_default\_constructor
  - has\_nothrow\_constructor, 105
- has\_not\_equal\_to
  - dont\_care, 103
  - has\_not\_equal\_to, 103-104
  - trait, 104
- has\_operator
  - Operator Type Traits, 14
- has\_plus
  - dont\_care, 106
  - has\_plus, 106-107
  - trait, 107
- has\_plus\_assign
  - dont\_care, 108
  - has\_plus\_assign, 108-109
  - trait, 109
- has\_post\_decrement
  - dont\_care, 109
  - has\_post\_decrement, 109-110
  - trait, 110
- has\_post\_increment
  - dont\_care, 111
  - has\_post\_increment, 111-112
  - trait, 112
- has\_pre\_decrement
  - dont\_care, 113
  - has\_pre\_decrement, 113-114
  - trait, 114
- has\_pre\_increment
  - dont\_care, 115

- has\_pre\_increment, 115-116
  - trait, 116
- has\_right\_shift
  - dont\_care, 117
  - has\_right\_shift, 117-118
  - trait, 118
- has\_right\_shift\_assign
  - dont\_care, 119
  - has\_right\_shift\_assign, 119-120
  - trait, 120
- has\_trivial\_assign
  - BOOST\_HAS\_TRIVIAL\_ASSIGN, 121
  - has\_trivial\_assign, 121
- has\_trivial\_constructor
  - BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 122
  - has\_trivial\_constructor, 122
  - has\_trivial\_default\_constructor, 122
- has\_trivial\_copy
  - BOOST\_HAS\_TRIVIAL\_COPY, 122
  - has\_trivial\_copy, 122
  - has\_trivial\_copy\_constructor, 122
- has\_trivial\_copy\_constructor
  - has\_trivial\_copy, 122
- has\_trivial\_default\_constructor
  - has\_trivial\_constructor, 122
- has\_trivial\_destructor
  - BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 123
  - has\_trivial\_destructor, 123
- has\_unary\_minus
  - dont\_care, 124
  - has\_unary\_minus, 124-125
  - Operator Type Traits, 14
  - trait, 125
- has\_unary\_plus
  - dont\_care, 126
  - has\_unary\_plus, 126-127
  - trait, 127
- has\_virtual\_destructor
  - BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 127
  - has\_virtual\_destructor, 127

## I

- Improving std::min with common\_type
  - common\_type, 35
- int
  - is\_function, 134
- integral\_constant
  - alignment\_of, 41
  - extent, 46
  - false\_type, 128
  - integral\_constant, 128
  - Operator Type Traits, 14
  - rank, 147
  - true\_type, 128
- integral\_promotion
  - integral\_promotion, 128
- is\_abstract

- is\_abstract, 128
- is\_arithmetic
  - is\_arithmetic, 129
- is\_array
  - is\_array, 129
- is\_base\_of
  - is\_base\_of, 130
- is\_class
  - BOOST\_IS\_CLASS, 130
  - is\_class, 130
  - User Defined Specializations, 28
- is\_complex
  - is\_complex, 131
- is\_compound
  - is\_compound, 131
- is\_const
  - is\_const, 132
- is\_convertible
  - is\_convertible, 132
- is\_convertible\_to\_Ret
  - Operator Type Traits, 14
- is\_empty
  - BOOST\_IS\_EMPTY, 133
  - is\_empty, 133
- is\_enum
  - is\_enum, 134
- is\_floating\_point
  - is\_floating\_point, 134
- is\_function
  - int, 134
  - is\_function, 134
- is\_fundamental
  - is\_fundamental, 136
- is\_integral
  - is\_integral, 136
- is\_lvalue\_reference
  - is\_lvalue\_reference, 136
- is\_member\_function\_pointer
  - is\_member\_function\_pointer, 137
- is\_member\_object\_pointer
  - is\_member\_object\_pointer, 137
- is\_member\_pointer
  - is\_member\_pointer, 138
- is\_object
  - is\_object, 138
- is\_pod
  - BOOST\_IS\_POD, 139
  - is\_pod, 139
  - User Defined Specializations, 28
- is\_pointer
  - Background and Tutorial, 5
  - is\_pointer, 139
- is\_polymorphic
  - is\_polymorphic, 140
- is\_reference
  - is\_reference, 140
- is\_rvalue\_reference
  - is\_rvalue\_reference, 141

- is\_same
  - is\_same, 141
- is\_scalar
  - is\_scalar, 142
- is\_signed
  - is\_signed, 142
- is\_stateless
  - is\_stateless, 143
- is\_union
  - BOOST\_IS\_UNION, 143
  - is\_union, 143
  - User Defined Specializations, 28
- is\_unsigned
  - is\_unsigned, 144
- is\_virtual\_base\_of
  - is\_virtual\_base\_of, 144
- is\_void
  - Background and Tutorial, 5
  - is\_void, 145
- is\_volatile
  - is\_volatile, 145

## M

Macros for Compiler Intrinsics

- BOOST\_ALIGNMENT\_OF, 29
- BOOST\_HAS\_NOTHROW\_ASSIGN, 29
- BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 29
- BOOST\_HAS\_NOTHROW\_COPY, 29
- BOOST\_HAS\_TRIVIAL\_ASSIGN, 29
- BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 29
- BOOST\_HAS\_TRIVIAL\_COPY, 29
- BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 29
- BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 29
- BOOST\_IS\_ABSTRACT, 29
- BOOST\_IS\_BASE\_OF, 29
- BOOST\_IS\_CLASS, 29
- BOOST\_IS\_CONVERTIBLE, 29
- BOOST\_IS\_EMPTY, 29
- BOOST\_IS\_ENUM, 29
- BOOST\_IS\_POD, 29
- BOOST\_IS\_POLYMORPHIC, 29
- BOOST\_IS\_UNION, 29

make\_signed

- make\_signed, 146

make\_unsigned

- make\_unsigned, 146

## N

no\_operator

- Operator Type Traits, 14

## O

Operator Type Traits

- any, 14
- check, 14
- dont\_care, 14
- has\_equal\_to, 24

- has\_operator, 14
- has\_unary\_minus, 14
- integral\_constant, 14
- is\_convertible\_to\_Ret, 14
- no\_operator, 14
- operator\_exists, 14
- operator\_returns\_Ret, 14
- operator\_returns\_void, 14
- returns\_void, 14
- returns\_void\_t, 14
- Rhs\_nocv, 14
- Rhs\_noptr, 14
- Rhs\_noref, 14
- trait, 24
- trait\_impl, 14
- trait\_impl1, 14
- operator\_exists
  - Operator Type Traits, 14
- operator\_returns\_Ret
  - Operator Type Traits, 14
- operator\_returns\_void
  - Operator Type Traits, 14

## P

- promote
  - promote, 147

## R

- rank
  - integral\_constant, 147
  - rank, 147
- remove\_all\_extents
  - remove\_all\_extents, 148
- remove\_const
  - remove\_const, 148
- remove\_cv
  - remove\_cv, 149
- remove\_extent
  - Background and Tutorial, 5
  - remove\_extent, 150
- remove\_pointer
  - remove\_pointer, 150
- remove\_reference
  - remove\_reference, 151
- remove\_volatile
  - remove\_volatile, 151
- result\_type
  - function\_traits, 47
- returns\_void
  - Operator Type Traits, 14
- returns\_void\_t
  - Operator Type Traits, 14
- Rhs\_nocv
  - Operator Type Traits, 14
- Rhs\_noptr
  - Operator Type Traits, 14
- Rhs\_noref

Operator Type Traits, 14

## T

trait

- has\_bit\_and, 49
- has\_bit\_and\_assign, 51
- has\_bit\_or, 53
- has\_bit\_or\_assign, 55
- has\_bit\_xor, 57
- has\_bit\_xor\_assign, 59
- has\_complement, 61
- has\_dereference, 63
- has\_divides, 65
- has\_divides\_assign, 67
- has\_equal\_to, 69
- has\_greater, 71
- has\_greater\_equal, 73
- has\_left\_shift, 75
- has\_left\_shift\_assign, 77
- has\_less, 79
- has\_less\_equal, 81
- has\_logical\_and, 83
- has\_logical\_not, 85
- has\_logical\_or, 87
- has\_minus, 89
- has\_minus\_assign, 91
- has\_modulus, 93
- has\_modulus\_assign, 95
- has\_multiplies, 97
- has\_multiplies\_assign, 99
- has\_negate, 101
- has\_not\_equal\_to, 104
- has\_plus, 107
- has\_plus\_assign, 109
- has\_post\_decrement, 110
- has\_post\_increment, 112
- has\_pre\_decrement, 114
- has\_pre\_increment, 116
- has\_right\_shift, 118
- has\_right\_shift\_assign, 120
- has\_unary\_minus, 125
- has\_unary\_plus, 127
- Operator Type Traits, 24

trait\_impl

Operator Type Traits, 14

trait\_impl1

Operator Type Traits, 14

true\_type

integral\_constant, 128

Type Traits that Transform One Type to Another

BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION, 25

type\_with\_alignment

type\_with\_alignment, 152

## U

User Defined Specializations

is\_class, 28

is\_pod, 28  
is\_union, 28

# Macro Index

## A

add\_const  
    add\_const, 37  
add\_cv  
    add\_cv, 37  
add\_lvalue\_reference  
    add\_lvalue\_reference, 38  
add\_pointer  
    add\_pointer, 38  
add\_reference  
    add\_reference, 39  
add\_rvalue\_reference  
    add\_rvalue\_reference, 40  
add\_volatile  
    add\_volatile, 40  
aligned\_storage  
    aligned\_storage, 41  
alignment\_of  
    alignment\_of, 41  
    integral\_constant, 41  
any  
    Operator Type Traits, 14

## B

Background and Tutorial  
    is\_pointer, 5  
    is\_void, 5  
    remove\_extent, 5  
BOOST\_ALIGNMENT\_OF  
    Macros for Compiler Intrinsics, 29  
BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION  
    Type Traits that Transform One Type to Another, 25  
BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEOF  
    common\_type, 42  
BOOST\_HAS\_NOTHROW\_ASSIGN  
    Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_NOTHROW\_CONSTRUCTOR  
    has\_nothrow\_constructor, 105  
    Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_NOTHROW\_COPY  
    has\_nothrow\_copy, 105  
    Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_ASSIGN  
    has\_trivial\_assign, 121  
    Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR  
    has\_trivial\_constructor, 122  
    Macros for Compiler Intrinsics, 29  
BOOST\_HAS\_TRIVIAL\_COPY  
    has\_trivial\_copy, 122  
    Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_TRIVIAL\_DESTRUCTOR**

has\_trivial\_destructor, 123

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_VIRTUAL\_DESTRUCTOR**

has\_virtual\_destructor, 127

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_ABSTRACT**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_BASE\_OF**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_CLASS**

is\_class, 130

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_CONVERTIBLE**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_EMPTY**

is\_empty, 133

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_ENUM**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_POD**

is\_pod, 139

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_POLYMORPHIC**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_UNION**

is\_union, 143

Macros for Compiler Intrinsics, 29

**C****check**

Operator Type Traits, 14

**common\_type**

BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEDEF, 42

common\_type, 42

Improving std::min with common\_type, 35

**D****decay**

decay, 45

**dont\_care**

has\_bit\_and, 48

has\_bit\_and\_assign, 50

has\_bit\_or, 52

has\_bit\_or\_assign, 54

has\_bit\_xor, 56

has\_bit\_xor\_assign, 58

has\_complement, 60

has\_dereference, 62

has\_divides, 64

has\_divides\_assign, 66

has\_equal\_to, 68

has\_greater, 70

has\_greater\_equal, 72

has\_left\_shift, 74

has\_left\_shift\_assign, 76

has\_less, 78

- has\_less\_equal, 80
- has\_logical\_and, 82
- has\_logical\_not, 84
- has\_logical\_or, 86
- has\_minus, 88
- has\_minus\_assign, 90
- has\_modulus, 92
- has\_modulus\_assign, 94
- has\_multiplies, 96
- has\_multiplies\_assign, 98
- has\_negate, 100
- has\_not\_equal\_to, 103
- has\_plus, 106
- has\_plus\_assign, 108
- has\_post\_decrement, 109
- has\_post\_increment, 111
- has\_pre\_decrement, 113
- has\_pre\_increment, 115
- has\_right\_shift, 117
- has\_right\_shift\_assign, 119
- has\_unary\_minus, 124
- has\_unary\_plus, 126
- Operator Type Traits, 14

## E

extent

- extent, 46
- integral\_constant, 46

## F

false\_type

- integral\_constant, 128

floating\_point\_promotion

- floating\_point\_promotion, 47

function\_traits

- function\_traits, 47
- result\_type, 47

## H

has\_bit\_and

- dont\_care, 48
- has\_bit\_and, 48-49
- trait, 49

has\_bit\_and\_assign

- dont\_care, 50
- has\_bit\_and\_assign, 50-51
- trait, 51

has\_bit\_or

- dont\_care, 52
- has\_bit\_or, 52-53
- trait, 53

has\_bit\_or\_assign

- dont\_care, 54
- has\_bit\_or\_assign, 54-55
- trait, 55

has\_bit\_xor

- dont\_care, 56

- has\_bit\_xor, 56-57
  - trait, 57
- has\_bit\_xor\_assign
  - dont\_care, 58
  - has\_bit\_xor\_assign, 58-59
  - trait, 59
- has\_complement
  - dont\_care, 60
  - has\_complement, 60-61
  - trait, 61
- has\_dereference
  - dont\_care, 62
  - has\_dereference, 62-63
  - trait, 63
- has\_divides
  - dont\_care, 64
  - has\_divides, 64-65
  - trait, 65
- has\_divides\_assign
  - dont\_care, 66
  - has\_divides\_assign, 66-67
  - trait, 67
- has\_equal\_to
  - dont\_care, 68
  - has\_equal\_to, 68-69
  - Operator Type Traits, 24
  - trait, 69
- has\_greater
  - dont\_care, 70
  - has\_greater, 70-71
  - trait, 71
- has\_greater\_equal
  - dont\_care, 72
  - has\_greater\_equal, 72-73
  - trait, 73
- has\_left\_shift
  - dont\_care, 74
  - has\_left\_shift, 74-75
  - trait, 75
- has\_left\_shift\_assign
  - dont\_care, 76
  - has\_left\_shift\_assign, 76-77
  - trait, 77
- has\_less
  - dont\_care, 78
  - has\_less, 78-79
  - trait, 79
- has\_less\_equal
  - dont\_care, 80
  - has\_less\_equal, 80-81
  - trait, 81
- has\_logical\_and
  - dont\_care, 82
  - has\_logical\_and, 82-83
  - trait, 83
- has\_logical\_not
  - dont\_care, 84
  - has\_logical\_not, 84-85

- trait, 85
- has\_logical\_or
  - dont\_care, 86
  - has\_logical\_or, 86-87
  - trait, 87
- has\_minus
  - dont\_care, 88
  - has\_minus, 88-89
  - trait, 89
- has\_minus\_assign
  - dont\_care, 90
  - has\_minus\_assign, 90-91
  - trait, 91
- has\_modulus
  - dont\_care, 92
  - has\_modulus, 92-93
  - trait, 93
- has\_modulus\_assign
  - dont\_care, 94
  - has\_modulus\_assign, 94-95
  - trait, 95
- has\_multiplies
  - dont\_care, 96
  - has\_multiplies, 96-97
  - trait, 97
- has\_multiplies\_assign
  - dont\_care, 98
  - has\_multiplies\_assign, 98-99
  - trait, 99
- has\_negate
  - dont\_care, 100
  - has\_negate, 100-101
  - trait, 101
- has\_new\_operator
  - has\_new\_operator, 102
- has\_nothrow\_assign
  - has\_nothrow\_assign, 104
- has\_nothrow\_constructor
  - BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 105
  - has\_nothrow\_constructor, 105
  - has\_nothrow\_default\_constructor, 105
- has\_nothrow\_copy
  - BOOST\_HAS\_NOTHROW\_COPY, 105
  - has\_nothrow\_copy, 105
  - has\_nothrow\_copy\_constructor, 105
- has\_nothrow\_copy\_constructor
  - has\_nothrow\_copy, 105
- has\_nothrow\_default\_constructor
  - has\_nothrow\_constructor, 105
- has\_not\_equal\_to
  - dont\_care, 103
  - has\_not\_equal\_to, 103-104
  - trait, 104
- has\_operator
  - Operator Type Traits, 14
- has\_plus
  - dont\_care, 106
  - has\_plus, 106-107

- trait, 107
- has\_plus\_assign
  - dont\_care, 108
  - has\_plus\_assign, 108-109
  - trait, 109
- has\_post\_decrement
  - dont\_care, 109
  - has\_post\_decrement, 109-110
  - trait, 110
- has\_post\_increment
  - dont\_care, 111
  - has\_post\_increment, 111-112
  - trait, 112
- has\_pre\_decrement
  - dont\_care, 113
  - has\_pre\_decrement, 113-114
  - trait, 114
- has\_pre\_increment
  - dont\_care, 115
  - has\_pre\_increment, 115-116
  - trait, 116
- has\_right\_shift
  - dont\_care, 117
  - has\_right\_shift, 117-118
  - trait, 118
- has\_right\_shift\_assign
  - dont\_care, 119
  - has\_right\_shift\_assign, 119-120
  - trait, 120
- has\_trivial\_assign
  - BOOST\_HAS\_TRIVIAL\_ASSIGN, 121
  - has\_trivial\_assign, 121
- has\_trivial\_constructor
  - BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 122
  - has\_trivial\_constructor, 122
  - has\_trivial\_default\_constructor, 122
- has\_trivial\_copy
  - BOOST\_HAS\_TRIVIAL\_COPY, 122
  - has\_trivial\_copy, 122
  - has\_trivial\_copy\_constructor, 122
- has\_trivial\_copy\_constructor
  - has\_trivial\_copy, 122
- has\_trivial\_default\_constructor
  - has\_trivial\_constructor, 122
- has\_trivial\_destructor
  - BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 123
  - has\_trivial\_destructor, 123
- has\_unary\_minus
  - dont\_care, 124
  - has\_unary\_minus, 124-125
  - Operator Type Traits, 14
  - trait, 125
- has\_unary\_plus
  - dont\_care, 126
  - has\_unary\_plus, 126-127
  - trait, 127
- has\_virtual\_destructor
  - BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 127

has\_virtual\_destructor, 127

## I

Improving std::min with common\_type

common\_type, 35

int

is\_function, 134

integral\_constant

alignment\_of, 41

extent, 46

false\_type, 128

integral\_constant, 128

Operator Type Traits, 14

rank, 147

true\_type, 128

integral\_promotion

integral\_promotion, 128

is\_abstract

is\_abstract, 128

is\_arithmetic

is\_arithmetic, 129

is\_array

is\_array, 129

is\_base\_of

is\_base\_of, 130

is\_class

BOOST\_IS\_CLASS, 130

is\_class, 130

User Defined Specializations, 28

is\_complex

is\_complex, 131

is\_compound

is\_compound, 131

is\_const

is\_const, 132

is\_convertible

is\_convertible, 132

is\_convertible\_to\_Ret

Operator Type Traits, 14

is\_empty

BOOST\_IS\_EMPTY, 133

is\_empty, 133

is\_enum

is\_enum, 134

is\_floating\_point

is\_floating\_point, 134

is\_function

int, 134

is\_function, 134

is\_fundamental

is\_fundamental, 136

is\_integral

is\_integral, 136

is\_lvalue\_reference

is\_lvalue\_reference, 136

is\_member\_function\_pointer

is\_member\_function\_pointer, 137

- is\_member\_object\_pointer
  - is\_member\_object\_pointer, 137
- is\_member\_pointer
  - is\_member\_pointer, 138
- is\_object
  - is\_object, 138
- is\_pod
  - BOOST\_IS\_POD, 139
  - is\_pod, 139
  - User Defined Specializations, 28
- is\_pointer
  - Background and Tutorial, 5
  - is\_pointer, 139
- is\_polymorphic
  - is\_polymorphic, 140
- is\_reference
  - is\_reference, 140
- is\_rvalue\_reference
  - is\_rvalue\_reference, 141
- is\_same
  - is\_same, 141
- is\_scalar
  - is\_scalar, 142
- is\_signed
  - is\_signed, 142
- is\_stateless
  - is\_stateless, 143
- is\_union
  - BOOST\_IS\_UNION, 143
  - is\_union, 143
  - User Defined Specializations, 28
- is\_unsigned
  - is\_unsigned, 144
- is\_virtual\_base\_of
  - is\_virtual\_base\_of, 144
- is\_void
  - Background and Tutorial, 5
  - is\_void, 145
- is\_volatile
  - is\_volatile, 145

## M

Macros for Compiler Intrinsics

- BOOST\_ALIGNMENT\_OF, 29
- BOOST\_HAS\_NOTHROW\_ASSIGN, 29
- BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 29
- BOOST\_HAS\_NOTHROW\_COPY, 29
- BOOST\_HAS\_TRIVIAL\_ASSIGN, 29
- BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 29
- BOOST\_HAS\_TRIVIAL\_COPY, 29
- BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 29
- BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 29
- BOOST\_IS\_ABSTRACT, 29
- BOOST\_IS\_BASE\_OF, 29
- BOOST\_IS\_CLASS, 29
- BOOST\_IS\_CONVERTIBLE, 29
- BOOST\_IS\_EMPTY, 29

BOOST\_IS\_ENUM, 29  
BOOST\_IS\_POD, 29  
BOOST\_IS\_POLYMORPHIC, 29  
BOOST\_IS\_UNION, 29

make\_signed  
    make\_signed, 146  
make\_unsigned  
    make\_unsigned, 146

## N

no\_operator  
    Operator Type Traits, 14

## O

Operator Type Traits  
    any, 14  
    check, 14  
    dont\_care, 14  
    has\_equal\_to, 24  
    has\_operator, 14  
    has\_unary\_minus, 14  
    integral\_constant, 14  
    is\_convertible\_to\_Ret, 14  
    no\_operator, 14  
    operator\_exists, 14  
    operator\_returns\_Ret, 14  
    operator\_returns\_void, 14  
    returns\_void, 14  
    returns\_void\_t, 14  
    Rhs\_nocv, 14  
    Rhs\_noptr, 14  
    Rhs\_noref, 14  
    trait, 24  
    trait\_impl, 14  
    trait\_impl1, 14  
operator\_exists  
    Operator Type Traits, 14  
operator\_returns\_Ret  
    Operator Type Traits, 14  
operator\_returns\_void  
    Operator Type Traits, 14

## P

promote  
    promote, 147

## R

rank  
    integral\_constant, 147  
    rank, 147  
remove\_all\_extents  
    remove\_all\_extents, 148  
remove\_const  
    remove\_const, 148  
remove\_cv  
    remove\_cv, 149  
remove\_extent

- Background and Tutorial, 5
- remove\_extent, 150
- remove\_pointer
  - remove\_pointer, 150
- remove\_reference
  - remove\_reference, 151
- remove\_volatile
  - remove\_volatile, 151
- result\_type
  - function\_traits, 47
- returns\_void
  - Operator Type Traits, 14
- returns\_void\_t
  - Operator Type Traits, 14
- Rhs\_nocv
  - Operator Type Traits, 14
- Rhs\_noptr
  - Operator Type Traits, 14
- Rhs\_noref
  - Operator Type Traits, 14

## T

- trait
  - has\_bit\_and, 49
  - has\_bit\_and\_assign, 51
  - has\_bit\_or, 53
  - has\_bit\_or\_assign, 55
  - has\_bit\_xor, 57
  - has\_bit\_xor\_assign, 59
  - has\_complement, 61
  - has\_dereference, 63
  - has\_divides, 65
  - has\_divides\_assign, 67
  - has\_equal\_to, 69
  - has\_greater, 71
  - has\_greater\_equal, 73
  - has\_left\_shift, 75
  - has\_left\_shift\_assign, 77
  - has\_less, 79
  - has\_less\_equal, 81
  - has\_logical\_and, 83
  - has\_logical\_not, 85
  - has\_logical\_or, 87
  - has\_minus, 89
  - has\_minus\_assign, 91
  - has\_modulus, 93
  - has\_modulus\_assign, 95
  - has\_multiplies, 97
  - has\_multiplies\_assign, 99
  - has\_negate, 101
  - has\_not\_equal\_to, 104
  - has\_plus, 107
  - has\_plus\_assign, 109
  - has\_post\_decrement, 110
  - has\_post\_increment, 112
  - has\_pre\_decrement, 114
  - has\_pre\_increment, 116

- has\_right\_shift, 118
- has\_right\_shift\_assign, 120
- has\_unary\_minus, 125
- has\_unary\_plus, 127
- Operator Type Traits, 24
- trait\_impl
  - Operator Type Traits, 14
- trait\_impl1
  - Operator Type Traits, 14
- true\_type
  - integral\_constant, 128
- Type Traits that Transform One Type to Another
  - BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION, 25
- type\_with\_alignment
  - type\_with\_alignment, 152

## U

User Defined Specializations

- is\_class, 28
- is\_pod, 28
- is\_union, 28

# Index

## A

add\_const

- add\_const, 37

add\_cv

- add\_cv, 37

add\_lvalue\_reference

- add\_lvalue\_reference, 38

add\_pointer

- add\_pointer, 38

add\_reference

- add\_reference, 39

add\_rvalue\_reference

- add\_rvalue\_reference, 40

add\_volatile

- add\_volatile, 40

aligned\_storage

- aligned\_storage, 41

alignment\_of

- alignment\_of, 41
- integral\_constant, 41

any

- Operator Type Traits, 14

## B

Background and Tutorial

- is\_pointer, 5
- is\_void, 5
- remove\_extent, 5

BOOST\_ALIGNMENT\_OF

- Macros for Compiler Intrinsics, 29

BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION

- Type Traits that Transform One Type to Another, 25

**BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEDEF**

common\_type, 42

**BOOST\_HAS\_NOTHROW\_ASSIGN**

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_NOTHROW\_CONSTRUCTOR**

has\_nothrow\_constructor, 105

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_NOTHROW\_COPY**

has\_nothrow\_copy, 105

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_TRIVIAL\_ASSIGN**

has\_trivial\_assign, 121

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR**

has\_trivial\_constructor, 122

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_TRIVIAL\_COPY**

has\_trivial\_copy, 122

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_TRIVIAL\_DESTRUCTOR**

has\_trivial\_destructor, 123

Macros for Compiler Intrinsics, 29

**BOOST\_HAS\_VIRTUAL\_DESTRUCTOR**

has\_virtual\_destructor, 127

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_ABSTRACT**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_BASE\_OF**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_CLASS**

is\_class, 130

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_CONVERTIBLE**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_EMPTY**

is\_empty, 133

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_ENUM**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_POD**

is\_pod, 139

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_POLYMORPHIC**

Macros for Compiler Intrinsics, 29

**BOOST\_IS\_UNION**

is\_union, 143

Macros for Compiler Intrinsics, 29

**C****check**

Operator Type Traits, 14

**common\_type**

BOOST\_COMMON\_TYPE\_DONT\_USE\_TYPEDEF, 42

common\_type, 42

Improving std::min with common\_type, 35

## D

### decay

decay, 45

### dont\_care

has\_bit\_and, 48  
has\_bit\_and\_assign, 50  
has\_bit\_or, 52  
has\_bit\_or\_assign, 54  
has\_bit\_xor, 56  
has\_bit\_xor\_assign, 58  
has\_complement, 60  
has\_dereference, 62  
has\_divides, 64  
has\_divides\_assign, 66  
has\_equal\_to, 68  
has\_greater, 70  
has\_greater\_equal, 72  
has\_left\_shift, 74  
has\_left\_shift\_assign, 76  
has\_less, 78  
has\_less\_equal, 80  
has\_logical\_and, 82  
has\_logical\_not, 84  
has\_logical\_or, 86  
has\_minus, 88  
has\_minus\_assign, 90  
has\_modulus, 92  
has\_modulus\_assign, 94  
has\_multiplies, 96  
has\_multiplies\_assign, 98  
has\_negate, 100  
has\_not\_equal\_to, 103  
has\_plus, 106  
has\_plus\_assign, 108  
has\_post\_decrement, 109  
has\_post\_increment, 111  
has\_pre\_decrement, 113  
has\_pre\_increment, 115  
has\_right\_shift, 117  
has\_right\_shift\_assign, 119  
has\_unary\_minus, 124  
has\_unary\_plus, 126  
Operator Type Traits, 14

## E

### extent

extent, 46  
integral\_constant, 46

## F

### false\_type

integral\_constant, 128

### floating\_point\_promotion

floating\_point\_promotion, 47

### function\_traits

function\_traits, 47  
result\_type, 47

## H

- has\_bit\_and
  - dont\_care, 48
  - has\_bit\_and, 48-49
  - trait, 49
- has\_bit\_and\_assign
  - dont\_care, 50
  - has\_bit\_and\_assign, 50-51
  - trait, 51
- has\_bit\_or
  - dont\_care, 52
  - has\_bit\_or, 52-53
  - trait, 53
- has\_bit\_or\_assign
  - dont\_care, 54
  - has\_bit\_or\_assign, 54-55
  - trait, 55
- has\_bit\_xor
  - dont\_care, 56
  - has\_bit\_xor, 56-57
  - trait, 57
- has\_bit\_xor\_assign
  - dont\_care, 58
  - has\_bit\_xor\_assign, 58-59
  - trait, 59
- has\_complement
  - dont\_care, 60
  - has\_complement, 60-61
  - trait, 61
- has\_dereference
  - dont\_care, 62
  - has\_dereference, 62-63
  - trait, 63
- has\_divides
  - dont\_care, 64
  - has\_divides, 64-65
  - trait, 65
- has\_divides\_assign
  - dont\_care, 66
  - has\_divides\_assign, 66-67
  - trait, 67
- has\_equal\_to
  - dont\_care, 68
  - has\_equal\_to, 68-69
  - Operator Type Traits, 24
  - trait, 69
- has\_greater
  - dont\_care, 70
  - has\_greater, 70-71
  - trait, 71
- has\_greater\_equal
  - dont\_care, 72
  - has\_greater\_equal, 72-73
  - trait, 73
- has\_left\_shift
  - dont\_care, 74
  - has\_left\_shift, 74-75

- trait, 75
- has\_left\_shift\_assign
  - dont\_care, 76
  - has\_left\_shift\_assign, 76-77
  - trait, 77
- has\_less
  - dont\_care, 78
  - has\_less, 78-79
  - trait, 79
- has\_less\_equal
  - dont\_care, 80
  - has\_less\_equal, 80-81
  - trait, 81
- has\_logical\_and
  - dont\_care, 82
  - has\_logical\_and, 82-83
  - trait, 83
- has\_logical\_not
  - dont\_care, 84
  - has\_logical\_not, 84-85
  - trait, 85
- has\_logical\_or
  - dont\_care, 86
  - has\_logical\_or, 86-87
  - trait, 87
- has\_minus
  - dont\_care, 88
  - has\_minus, 88-89
  - trait, 89
- has\_minus\_assign
  - dont\_care, 90
  - has\_minus\_assign, 90-91
  - trait, 91
- has\_modulus
  - dont\_care, 92
  - has\_modulus, 92-93
  - trait, 93
- has\_modulus\_assign
  - dont\_care, 94
  - has\_modulus\_assign, 94-95
  - trait, 95
- has\_multiplies
  - dont\_care, 96
  - has\_multiplies, 96-97
  - trait, 97
- has\_multiplies\_assign
  - dont\_care, 98
  - has\_multiplies\_assign, 98-99
  - trait, 99
- has\_negate
  - dont\_care, 100
  - has\_negate, 100-101
  - trait, 101
- has\_new\_operator
  - has\_new\_operator, 102
- has\_nothrow\_assign
  - has\_nothrow\_assign, 104
- has\_nothrow\_constructor

- BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 105
  - has\_nothrow\_constructor, 105
  - has\_nothrow\_default\_constructor, 105
- has\_nothrow\_copy
  - BOOST\_HAS\_NOTHROW\_COPY, 105
  - has\_nothrow\_copy, 105
  - has\_nothrow\_copy\_constructor, 105
- has\_nothrow\_copy\_constructor
  - has\_nothrow\_copy, 105
- has\_nothrow\_default\_constructor
  - has\_nothrow\_constructor, 105
- has\_not\_equal\_to
  - dont\_care, 103
  - has\_not\_equal\_to, 103-104
  - trait, 104
- has\_operator
  - Operator Type Traits, 14
- has\_plus
  - dont\_care, 106
  - has\_plus, 106-107
  - trait, 107
- has\_plus\_assign
  - dont\_care, 108
  - has\_plus\_assign, 108-109
  - trait, 109
- has\_post\_decrement
  - dont\_care, 109
  - has\_post\_decrement, 109-110
  - trait, 110
- has\_post\_increment
  - dont\_care, 111
  - has\_post\_increment, 111-112
  - trait, 112
- has\_pre\_decrement
  - dont\_care, 113
  - has\_pre\_decrement, 113-114
  - trait, 114
- has\_pre\_increment
  - dont\_care, 115
  - has\_pre\_increment, 115-116
  - trait, 116
- has\_right\_shift
  - dont\_care, 117
  - has\_right\_shift, 117-118
  - trait, 118
- has\_right\_shift\_assign
  - dont\_care, 119
  - has\_right\_shift\_assign, 119-120
  - trait, 120
- has\_trivial\_assign
  - BOOST\_HAS\_TRIVIAL\_ASSIGN, 121
  - has\_trivial\_assign, 121
- has\_trivial\_constructor
  - BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 122
  - has\_trivial\_constructor, 122
  - has\_trivial\_default\_constructor, 122
- has\_trivial\_copy
  - BOOST\_HAS\_TRIVIAL\_COPY, 122

- has\_trivial\_copy, 122
- has\_trivial\_copy\_constructor, 122
- has\_trivial\_copy\_constructor
  - has\_trivial\_copy, 122
- has\_trivial\_default\_constructor
  - has\_trivial\_constructor, 122
- has\_trivial\_destructor
  - BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 123
  - has\_trivial\_destructor, 123
- has\_unary\_minus
  - dont\_care, 124
  - has\_unary\_minus, 124-125
  - Operator Type Traits, 14
  - trait, 125
- has\_unary\_plus
  - dont\_care, 126
  - has\_unary\_plus, 126-127
  - trait, 127
- has\_virtual\_destructor
  - BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 127
  - has\_virtual\_destructor, 127

## I

- Improving std::min with common\_type
  - common\_type, 35
- int
  - is\_function, 134
- integral\_constant
  - alignment\_of, 41
  - extent, 46
  - false\_type, 128
  - integral\_constant, 128
  - Operator Type Traits, 14
  - rank, 147
  - true\_type, 128
- integral\_promotion
  - integral\_promotion, 128
- is\_abstract
  - is\_abstract, 128
- is\_arithmetic
  - is\_arithmetic, 129
- is\_array
  - is\_array, 129
- is\_base\_of
  - is\_base\_of, 130
- is\_class
  - BOOST\_IS\_CLASS, 130
  - is\_class, 130
  - User Defined Specializations, 28
- is\_complex
  - is\_complex, 131
- is\_compound
  - is\_compound, 131
- is\_const
  - is\_const, 132
- is\_convertible
  - is\_convertible, 132

- is\_convertible\_to\_Ret
  - Operator Type Traits, 14
- is\_empty
  - BOOST\_IS\_EMPTY, 133
  - is\_empty, 133
- is\_enum
  - is\_enum, 134
- is\_floating\_point
  - is\_floating\_point, 134
- is\_function
  - int, 134
  - is\_function, 134
- is\_fundamental
  - is\_fundamental, 136
- is\_integral
  - is\_integral, 136
- is\_lvalue\_reference
  - is\_lvalue\_reference, 136
- is\_member\_function\_pointer
  - is\_member\_function\_pointer, 137
- is\_member\_object\_pointer
  - is\_member\_object\_pointer, 137
- is\_member\_pointer
  - is\_member\_pointer, 138
- is\_object
  - is\_object, 138
- is\_pod
  - BOOST\_IS\_POD, 139
  - is\_pod, 139
  - User Defined Specializations, 28
- is\_pointer
  - Background and Tutorial, 5
  - is\_pointer, 139
- is\_polymorphic
  - is\_polymorphic, 140
- is\_reference
  - is\_reference, 140
- is\_rvalue\_reference
  - is\_rvalue\_reference, 141
- is\_same
  - is\_same, 141
- is\_scalar
  - is\_scalar, 142
- is\_signed
  - is\_signed, 142
- is\_stateless
  - is\_stateless, 143
- is\_union
  - BOOST\_IS\_UNION, 143
  - is\_union, 143
  - User Defined Specializations, 28
- is\_unsigned
  - is\_unsigned, 144
- is\_virtual\_base\_of
  - is\_virtual\_base\_of, 144
- is\_void
  - Background and Tutorial, 5
  - is\_void, 145

is\_volatile  
    is\_volatile, 145

## M

### Macros for Compiler Intrinsics

BOOST\_ALIGNMENT\_OF, 29  
BOOST\_HAS\_NOTHROW\_ASSIGN, 29  
BOOST\_HAS\_NOTHROW\_CONSTRUCTOR, 29  
BOOST\_HAS\_NOTHROW\_COPY, 29  
BOOST\_HAS\_TRIVIAL\_ASSIGN, 29  
BOOST\_HAS\_TRIVIAL\_CONSTRUCTOR, 29  
BOOST\_HAS\_TRIVIAL\_COPY, 29  
BOOST\_HAS\_TRIVIAL\_DESTRUCTOR, 29  
BOOST\_HAS\_VIRTUAL\_DESTRUCTOR, 29  
BOOST\_IS\_ABSTRACT, 29  
BOOST\_IS\_BASE\_OF, 29  
BOOST\_IS\_CLASS, 29  
BOOST\_IS\_CONVERTIBLE, 29  
BOOST\_IS\_EMPTY, 29  
BOOST\_IS\_ENUM, 29  
BOOST\_IS\_POD, 29  
BOOST\_IS\_POLYMORPHIC, 29  
BOOST\_IS\_UNION, 29

make\_signed  
    make\_signed, 146  
make\_unsigned  
    make\_unsigned, 146

## N

no\_operator  
    Operator Type Traits, 14

## O

### Operator Type Traits

any, 14  
check, 14  
dont\_care, 14  
has\_equal\_to, 24  
has\_operator, 14  
has\_unary\_minus, 14  
integral\_constant, 14  
is\_convertible\_to\_Ret, 14  
no\_operator, 14  
operator\_exists, 14  
operator\_returns\_Ret, 14  
operator\_returns\_void, 14  
returns\_void, 14  
returns\_void\_t, 14  
Rhs\_nocv, 14  
Rhs\_noptr, 14  
Rhs\_noref, 14  
trait, 24  
trait\_impl, 14  
trait\_impl1, 14  
operator\_exists  
    Operator Type Traits, 14  
operator\_returns\_Ret

Operator Type Traits, 14  
operator\_returns\_void  
Operator Type Traits, 14

## P

promote  
promote, 147

## R

rank  
integral\_constant, 147  
rank, 147  
remove\_all\_extents  
remove\_all\_extents, 148  
remove\_const  
remove\_const, 148  
remove\_cv  
remove\_cv, 149  
remove\_extent  
Background and Tutorial, 5  
remove\_extent, 150  
remove\_pointer  
remove\_pointer, 150  
remove\_reference  
remove\_reference, 151  
remove\_volatile  
remove\_volatile, 151  
result\_type  
function\_traits, 47  
returns\_void  
Operator Type Traits, 14  
returns\_void\_t  
Operator Type Traits, 14  
Rhs\_nocv  
Operator Type Traits, 14  
Rhs\_noptr  
Operator Type Traits, 14  
Rhs\_noref  
Operator Type Traits, 14

## T

trait  
has\_bit\_and, 49  
has\_bit\_and\_assign, 51  
has\_bit\_or, 53  
has\_bit\_or\_assign, 55  
has\_bit\_xor, 57  
has\_bit\_xor\_assign, 59  
has\_complement, 61  
has\_dereference, 63  
has\_divides, 65  
has\_divides\_assign, 67  
has\_equal\_to, 69  
has\_greater, 71  
has\_greater\_equal, 73  
has\_left\_shift, 75  
has\_left\_shift\_assign, 77

- has\_less, 79
- has\_less\_equal, 81
- has\_logical\_and, 83
- has\_logical\_not, 85
- has\_logical\_or, 87
- has\_minus, 89
- has\_minus\_assign, 91
- has\_modulus, 93
- has\_modulus\_assign, 95
- has\_multiplies, 97
- has\_multiplies\_assign, 99
- has\_negate, 101
- has\_not\_equal\_to, 104
- has\_plus, 107
- has\_plus\_assign, 109
- has\_post\_decrement, 110
- has\_post\_increment, 112
- has\_pre\_decrement, 114
- has\_pre\_increment, 116
- has\_right\_shift, 118
- has\_right\_shift\_assign, 120
- has\_unary\_minus, 125
- has\_unary\_plus, 127
- Operator Type Traits, 24
- trait\_impl
  - Operator Type Traits, 14
- trait\_impl1
  - Operator Type Traits, 14
- true\_type
  - integral\_constant, 128
- Type Traits that Transform One Type to Another
  - BOOST\_BROKEN\_COMPILER\_TYPE\_TRAITS\_SPECIALIZATION, 25
- type\_with\_alignment
  - type\_with\_alignment, 152

## U

### User Defined Specializations

- is\_class, 28
- is\_pod, 28
- is\_union, 28