Document number: P1016R0
Revises:
Date: 2018-04-06
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: LEWG
Reply to: Vincent Reverdy and Robert J. Brunner
University of Illinois at Urbana-Champaign
vince.rev@gmail.com

A few additional type manipulation utilities

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Abstract

We introduce additional type traits to the standard library. Most of these types traits have been used again and again in the implementation of a library dedicated to the creation of custom overload sets that will be proposed for standardization in a separate proposal. These type traits focus on fives domains: the removal of several pointers, the manipulation of qualifiers, the use of conditional inheritance, the categorization of callable types, and a few additional type aliases and template variables as metaprogramming helpers.

Contents

1 Proposal [proposal]

1.1 Introduction [proposal.intro]

Since their introduction with C++11, the type traits of the standard library have been of great help for template metaprogramming. They contributed to the standardization of common metaprogramming patterns, such as SFINAE with `enable_if`, and since C++17 with `void_t`. In this paper, we introduce new type traits corresponding to metaprogramming patterns that turned out to be very useful for the implementation of a tool to build custom overload sets. This tool will be proposed for standardization in a separate paper. We believe that the listed type traits are of common use and could benefit the entire community. The new type traits focus on the five following areas:

— removal of all pointers: `remove_all_pointers` inspired from `remove_all_extents`

— manipulation of qualifiers: `copy_*` and `clone_*` type traits

— conditional inheritance: `blank` struct helper, `is_inheritable` and `inherit_if`

— callable categorization: `is_closure`, `is_functor`, `is_function_object` and `is_callable`

— utility type aliases and template variables: `index_constant`, `type_t`, `false_v` and `true_v`

1.2 Impact on the standard [proposal.impact]

This proposal is a pure library extension. It does not require changes to any standard classes or functions. All the extensions belong to the `<type_traits>` header.

1.3 Motivations and design decisions [proposal.design]

1.3.1 Pointers removal [proposal.design.ptr]

```
// Pointers removal
template <class T> struct remove_all_pointers;

// Type alias
template <class T> using remove_all_pointers_t = typename remove_all_pointers<T>::type;
```

The current standard library includes two type traits to manipulate extents: `remove_extent` which removes the first array dimension, and `remove_all_extents` which removes all dimensions. However, for pointers, only one is provided: `remove_pointer` which removes one pointer. However for the same reason that it can be useful to remove all dimensions, it can sometimes be useful to remove all pointers and access the “raw” type. Also, in the context of qualifiers manipulation (see (1.3.2)), it makes sense to provide tools to transform a `int***` into a `double***` by transferring all pointers from one type to another: `copy_all_pointers` and `clone_all_pointers`. In this context, being able to remove all pointers seems to be a natural addition to the standard library, for completeness. For all these reasons, we propose to introduce the type trait: `remove_all_pointers`.

1.3.2 Qualifiers manipulation [proposal.design.qual]

```
// Qualifiers manipulation
template <class From, class To> struct copy_const;
template <class From, class To> struct clone_const;
template <class From, class To> struct copy_volatile;
template <class From, class To> struct clone_volatile;
template <class From, class To> struct copy_cv;
template <class From, class To> struct clone_cv;
```

```
template <class From, class To> struct copy_reference;
template <class From, class To> struct clone_reference;
template <class From, class To> struct copy_signedness;
template <class From, class To> struct copy_extent;
template <class From, class To> struct clone_extent;
template <class From, class To> struct copy_all_extents;
template <class From, class To> struct clone_all_extents;
template <class From, class To> struct copy_pointer;
template <class From, class To> struct clone_pointer;
template <class From, class To> struct copy_all_pointers;
template <class From, class To> struct clone_all_pointers;
template <class From, class To> struct copy_cvref;
template <class From, class To> struct clone_cvref;

// Type aliases
template <class F, class T> using copy_const_t = typename copy_const<F, T>::type;
template <class F, class T> using clone_const_t = typename clone_const<F, T>::type;
template <class F, class T> using copy_volatile_t = typename copy_volatile<F, T>::type;
template <class F, class T> using clone_volatile_t = typename clone_volatile<F, T>::type;
template <class F, class T> using copy_cv_t = typename copy_cv<F, T>::type;
template <class F, class T> using clone_cv_t = typename clone_cv<F, T>::type;
template <class F, class T> using copy_reference_t = typename copy_reference<F, T>::type;
template <class F, class T> using clone_reference_t = typename clone_reference<F, T>::type;
template <class F, class T> using copy_signedness_t = typename copy_signedness<F, T>::type;
template <class F, class T> using copy_extent_t = typename copy_extent<F, T>::type;
template <class F, class T> using clone_extent_t = typename clone_extent<F, T>::type;
template <class F, class T> using copy_all_extents_t = typename copy_all_extents<F, T>::type;
template <class F, class T> using clone_all_extents_t = typename clone_all_extents<F, T>::type;
template <class F, class T> using copy_pointer_t = typename copy_pointer<F, T>::type;
template <class F, class T> using clone_pointer_t = typename clone_pointer<F, T>::type;
template <class F, class T> using copy_all_pointers_t = typename copy_all_pointers<F, T>::type;
template <class F, class T> using clone_all_pointers_t = typename clone_all_pointers<F, T>::type;
template <class F, class T> using copy_cvref_t = typename copy_cvref<F, T>::type;
template <class F, class T> using clone_cvref_t = typename clone_cvref<F, T>::type;
```

In the heavy template metaprogramming involved in the building of custom overload sets, one pattern happened to be very useful: being able to transfer the qualifiers of one type to another one. For example, to transform a `const int&` into a `const double&`, a `int[1][2][3]` into a `double[1][2][3]`, or an `int***` to a `double***`. It can be also used in a function taking a universal reference as an input, to qualify another type based on the qualification of the input:

```
template <class T> void f(T&& x) {
    // An integer with the same qualification as the input
    using integer = std::copy_cvref_t<T&&, int>;
    /* function contents */
}
```

or to make a type `const` depending on another type:

```
template <class T> struct foo {
    // Data members
    T a;
    std::copy_const_t<T, int> n;
    std::copy_const_t<T, double> x;
    /* class contents */
};
```

Another uses are illustrated in P0847R0, where `copy_cvref_t` is called `like_t`.

For completeness, qualifier manipulators are added to all existing categories of type transformations: cv (2.1.7.1), reference (2.1.7.2), sign (2.1.7.3), array (2.1.7.4) and pointer (2.1.7.5). Additionally, depending on the behavior regarding the second template parameter, two kinds of qualifier parameters are introduced: the copiers `copy_*` and the cloners `clone_*`. The difference is that the copiers directly copy the qualifiers of the first argument to the second, while cloners first discard the qualifiers of the second argument. For example `copy_cv_t<volatile int, const double>` evaluates to `const volatile double` while `clone_cv_t<volatile int, const double>` evaluates to `volatile double`, and `copy_all_pointers_t<int***, double*>` evaluates to `double****` while `clone_all_pointers_t<int***, double*>` evaluates to `double***`.

The complete list of proposed `copy_*` and `clone_*` traits is:

— const-volatile modifications: `copy_const`, `clone_const`, `copy_volatile`, `clone_volatile`, `copy_cv`, `clone_cv`

— reference modifications: `copy_reference`, `clone_reference`

— sign modifications: `copy_signedness`

— array modifications: `copy_extent`, `clone_extent`, `copy_all_extents`, `clone_all_extents`

— pointer modifications: `copy_pointer`, `clone_pointer`, `copy_all_pointers`, `clone_all_pointers`

— other transformations: `copy_cvref`, `clone_cvref`

As a note, in the same way `remove_pointer` deals with cv-qualified pointers, `copy_pointer`, `clone_pointer`, `copy_all_pointers`, `clone_all_pointers` copy the cv-qualifiers of pointers. Also `copy_signedness` is preferred over `copy_sign` to avoid confusion with the existing mathematical function `copysign`. Finally, `clone_signedness` is not introduced, because `remove_sign` does not exist, and does not seem to be a relevant type trait to introduce, the only interesting use case being to transform a `signed char` or an `unsigned char` into a `char`. The difference between `copy_signedness` and a hypothetical `clone_signedness` would be the following: `copy_signedness_t<char, unsigned char>` would evaluate to `unsigned char` while `clone_signedness_t<char, unsigned char>` would evaluate to `char`. In both cases `copy/clone_signedness_t<unsigned int, int>` would evaluate to `unsigned int` and `copy/clone_signedness_t<signed int, unsigned int>` would evaluate to `signed int`.

1.3.3 Conditional inheritance [proposal.design.inher]

```
// Inheritance
struct blank;
template <class T> struct is_inheritable;
template <bool b, class T> struct inherit_if;

// Type alias and variable template
template <class T> inline constexpr bool is_inheritable_v = is_inheritable<T>::value;
template <bool b, class T> using inherit_if_t = typename inherit_if<b, T>::type;
```

The standard library currently does not provide tools dedicated to the use of conditional inheritance. Conditional inheritance consists in conditionally inheriting from a class depending on a condition, usually related to the template parameters of the derived class. To facilitate it, we propose to introduce the following tools:

— `blank`: an generic empty class

— `is_inheritable`: a trait to detect whether one can inherit from a class

— `inherit_if`: a trait to inherit from a class if a condition is satisfied

The use of `inherit_if` can be illustrated by the following:

```
template <class T>
```

```
struct foo: std::inherit_if_t<std::is_class_v<T> && !std::is_final_v<T>, T> {
    /* class contents */
};

template <class T>
struct bar: std::inherit_if_t<std::is_integral_v<T>, foo<T>> {
    /* class contents */
};
```

where `foo<T>` derives from `T` if `T` is a non-final, non-union class, and `bar<T>` derives from `foo<T>` if `T` is an integral type.

In practice, `inherit_if` can be defined as:

```
struct blank {};
template <class b, class T> struct inherit_if {
    using type = conditional_t<b, T, blank>;
};
```

Even though in that context, `blank` and `is_inheritable` are related to `inherit_if`, they can be seen as independent and relying on other motivations. Checking if one can derive from a type or not is a useful information by itself, and that is the reason behind `is_inheritable`. Then, as shown by the last block of code, `inherit_if`, as many other tools that can be developed in the metaprogramming world, requires an empty class. Since C++17, the standard proposes such as class: `monostate` in the `<variant>` header. As an empty class is a very useful tool to have, there are three main solutions:

— make `monostate` the universally accepted empty class provided by the standard library by putting it in a less specific header such as `<type_traits>` or `<utility>`

— keep `monostate` specific to `variant` and introduce a new universally accepted empty class provided by the standard library such as `blank` (the name being inspired from The Boost C++ Libraries)

— introduce an different empty class for each specific use, and change the name of `blank` for `empty_base`

1.3.4 Callable categorization [proposal.design.call]

```
// Callable categorization
template <class T> struct is_closure;
template <class T> struct is_functor;
template <class T> struct is_function_object;
template <class T> struct is_callable;

// Variable templates
template <class T> inline constexpr bool is_closure_v = is_closure<T>::value;
template <class T> inline constexpr bool is_functor_v = is_functor<T>::value;
template <class T> inline constexpr bool is_function_object_v = is_function_object<T>::value;
template <class T> inline constexpr bool is_callable_v = is_callable<T>::value;
```

With `is_function` and `is_member_function_pointer`, the standard already allow to detect two types of callables. Additionally, `is_invocable` and its variations allows to check whether a callable can be called with a given set of arguments. However, the standard library does not currently provide a way to check if a type is a callable although the concept is defined in [func.def]: "A callable type is a function object type or a pointer to member". Function object types are themselves defined in [function.objects]: "A function object type is an object type that can be the type of the postfix-expression in a function call". To build custom overload sets, a far more fine-grained categorization of callable types was necessary. As these tools are of general interest and allow to detect types that are already defined in the existing wording of the standard, we propose them for standardization. Furthermore P0604R0 was already discussing the idea of introducing an `is_callable` trait additionally to `is_invocable`.

In this context, we propose to introduce the following traits:

— `is_closure`: to detect closure types as already defined in the standard in [expr.prim.lambda.closure]

— `is_functor`: to detect (possibly union) class types that have an overloaded `operator()`

— `is_function_object`: to detect function object types as already defined in the standard in [function.objects]

— `is_callable`: to detect callable types as already defined in the standard in [func.def]

Even though functors are currently not formally defined in the standard, functors are a common concept in programming languages. Also, just introducing `is_function_object` could be very misleading for users since function objects can be usually thought as functors although in the C++ standard function objects is a more general notion, including for example function pointers. The ambiguity has been confirmed informally on StackOverflow, where general users would expect a function object to be a functor, while experts would be aware of the difference. The introduction of both traits break the ambiguity.

1.3.5 General helpers [proposal.design.utils]

```
// Helpers
template <size_t I> using index_constant = integral_constant<size_t, I>;
template <class T, class...> using type_t = T;
template <class...> inline constexpr bool false_v = false;
template <class...> inline constexpr bool true_v = true;
```

In the same way `index_sequence<I...>` is defined to be `integer_sequence<size_t, I...>`, we propose to introduce `index_constant` for completeness and because using an `integral_constant` of type `size_t` is a common pattern to specify a constant size.

In C++17, the standard introduces `void_t`, a very useful tool to detect ill-formed types in SFINAE contexts. `void_t` maps an arbitrary sequence of types to `void`. We suggest the introduction of a more general version of this tool, to map an arbitrary sequence of types, to a given type. In that sense, `void_t` could be thought of as:

```
template <class... X> using void_t = type_t<void, X...>;
```

`type_t` is just as useful as `void_t`, and be used in contexts where one wants to detect ill-formed types in SFINAE contexts, but also return a type based on this information.

Additionally two template variables were suggested on future-proposals: `false_v` and `true_v`. Even though a simpler definition is possible, they can be thought of as:

```
template <class... X> inline constexpr bool false_v = type_t<false_type, X...>::value;
template <class... X> inline constexpr bool true_v = type_t<true_type, X...>::value;
```

An example of use is the content of a `static_assert` in a template context such as illustrated here and here. A basic use case of `false_v` is, when in a template context, for example a template function, a `false` would produce a hard error. By making the `false` statement dependent on the template parameters, it is possible to delay instantiation and prevent the hard error as in the example below:

```
// Example provided by Arthur O'Dwyer
template<class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor {
    public:
    using oalloc_t = OuterAlloc;
    using ialloc_t = scoped_allocator_adaptor<InnerAllocs...>;

    private:
    oalloc_t _outer;
    ialloc_t _inner;
```

```
    using _otraits = allocator_traits<OuterAlloc>;

    public:
    template<class... Args>
    void construct(value_type* p, Args&&... args) {
        if constexpr (!uses_allocator_v<value_type, ialloc_t>) {
            _otraits::construct(_outer, p, std::forward<Args>(args)...);
        } else if constexpr (is_constructible_v<value_type, alloc_arg_t, ialloc_t, Args&&...>) {
            _otraits::construct(_outer, p, allocator_arg, _inner, std::forward<Args>(args)...);
        } else if constexpr (is_constructible_v<value_type, Args&&..., ialloc_t&>) {
            _otraits::construct(_outer, p, std::forward<Args>(args)..., _inner);
        } else {
            // static_assert(false, "value_type is not constructible from args"); // not working
            static_assert(false_v<Args&&...>, "value_type is not constructible from args");
        }
    }
};
```

1.4 Technical specification [proposal.spec]

See the wording (part 2).

1.5 Discussion and open questions [proposal.discussion]

1.5.1 Bikeshedding [proposal.discussion.bikeshed]

While some names are straightforward and follow existing patterns in standard library, the following names are the most likely to be debated:

- `copy_*`
- `clone_*`
- `copy_signedness`
- `blank`
- `is_inheritable`
- `inherit_if`
- `type_t`
- `false_v`
- `true_v`

1.5.2 Questions [proposal.discussion.questions]

Finally, the following questions should be answered:

- `blank`: is `monostate` the universal empty class of the standard library, is a new one required, or is a specific `empty_base` necessary?
- `is_functor`: should classes with `private` or `protected` function call operators be considered functors, on only those with at least one `public operator()`?
- `is_callable`: should reference to callables and cv-qualified callables be themselves considered callables?

1.6 Acknowledgements [proposal.ackwldgmnts]

The authors would like to thank the participants to the related discussion on the future-proposals group, as well as Daniel Krügler for the mail exchange about `is_callable`, and Arthur O'Dwyer about `false_v` and `true_v`. This work has been made possible thanks to the National Science Foundation through the awards CCF-1647432 and SI2-SSE-1642411.

1.7 References [proposal.references]

N4727, Working Draft, Standard for Programming Language C++, Richard Smith, *ISO/IEC JTC1/SC22/WG21* (February 2018)

P0847R0, Deducing this, Gasper Azman et al., *ISO/IEC JTC1/SC22/WG21* (February 2018)

P0604R0, Resolving GB 55, US 84, US 85, US 86, Daniel Krügler et al., *ISO/IEC JTC1/SC22/WG21* (March 2017)

General purpose utilities for template metaprogramming and type manipulation, ISO C++ Standard - Future Proposals, *Google Groups* (March 2018)

Boost blank, Eric Friedman, *The Boost C++ Libraries* (2003)

Unit testing highly templated library, Guillaume Racicot, *StackOverflow* (November 2016)

Are function pointers function objects in C++?, Vincent Reverdy, *StackOverflow* (March 2018)

Use-cases for `false_v`, Arthur O'Dwyer, *Blog post* (April 2018)

2 Wording [wording]

2.1 Metaprogramming and type traits [meta]

2.1.1 Requirements [meta.rqmts]

1 No modification.

2.1.2 Header <type_traits> synopsis [meta.type.synop]

1 Add the following to the synopsis of <type_traits>:

```
namespace std {
    // 2.1.3, helper classes
    struct blank;
    template <size_t I> using index_constant = integral_constant<size_t, I>;

    // 2.1.4.1, primary type categories
    template <class T> struct is_closure;

    template <class T>
    inline constexpr bool is_closure_v = is_closure<T>::value;

    // 2.1.4.2, composite type categories
    template <class T> struct is_functor;
    template <class T> struct is_function_object;
    template <class T> struct is_callable;

    template <class T>
    inline constexpr bool is_functor_v = is_functor<T>::value;
    template <class T>
    inline constexpr bool is_function_object_v = is_function_object<T>::value;
    template <class T>
    inline constexpr bool is_callable_v = is_callable<T>::value;

    // 2.1.4.3, type properties
    template <class T> struct is_inheritable;

    template <class T>
    inline constexpr bool is_inheritable_v = is_inheritable<T>::value;

    // 2.1.5, type property queries

    // 2.1.6, type relations

    // 2.1.7.1, const-volatile modifications
    template <class From, class To> struct copy_const;
    template <class From, class To> struct clone_const;
    template <class From, class To> struct copy_volatile;
    template <class From, class To> struct clone_volatile;
    template <class From, class To> struct copy_cv;
    template <class From, class To> struct clone_cv;
```

```
template <class From, class To>
using copy_const_t = typename copy_const<From, To>::type;
template <class From, class To>
using clone_const_t = typename clone_const<From, To>::type;
template <class From, class To>
using copy_volatile_t = typename copy_volatile<From, To>::type;
template <class From, class To>
using clone_volatile_t = typename clone_volatile<From, To>::type;
template <class From, class To>
using copy_cv_t = typename copy_cv<From, To>::type;
template <class From, class To>
using clone_cv_t = typename clone_cv<From, To>::type;

// 2.1.7.2, reference modifications
template <class From, class To> struct copy_reference;
template <class From, class To> struct clone_reference;

template <class From, class To>
using copy_reference_t = typename copy_reference<From, To>::type;
template <class From, class To>
using clone_reference_t = typename clone_reference<From, To>::type;

// 2.1.7.3, sign modifications
template <class From, class To> struct copy_signedness;

template <class From, class To>
using copy_signedness_t = typename copy_signedness<From, To>::type;

// 2.1.7.4, array modifications
template <class From, class To> struct copy_extent;
template <class From, class To> struct clone_extent;
template <class From, class To> struct copy_all_extents;
template <class From, class To> struct clone_all_extents;

template <class From, class To>
using copy_extent_t = typename copy_extent<From, To>::type;
template <class From, class To>
using clone_extent_t = typename clone_extent<From, To>::type;
template <class From, class To>
using copy_all_extents_t = typename copy_all_extents<From, To>::type;
template <class From, class To>
using clone_all_extents_t = typename clone_all_extents<From, To>::type;

// 2.1.7.5, pointer modifications
template <class T> struct remove_all_pointers;
template <class From, class To> struct copy_pointer;
template <class From, class To> struct clone_pointer;
template <class From, class To> struct copy_all_pointers;
template <class From, class To> struct clone_all_pointers;

template <class T>
using remove_all_pointers_t = typename remove_all_pointers<T>::type;
template <class From, class To>
using copy_pointer_t = typename copy_pointer<From, To>::type;
template <class From, class To>
```

```
    using clone_pointer_t = typename clone_pointer<From, To>::type;
    template <class From, class To>
    using copy_all_pointers_t = typename copy_all_pointers<From, To>::type;
    template <class From, class To>
    using clone_all_pointers_t = typename clone_all_pointers<From, To>::type;

    // 2.1.7.6, other transformations
    template <class From, class To> struct copy_cvref;
    template <class From, class To> struct clone_cvref;
    template <bool b, class T> struct inherit_if;

    template <class From, class To>
    using copy_cvref_t = typename copy_cvref<From, To>::type;
    template <class From, class To>
    using clone_cvref_t = typename clone_cvref<From, To>::type;
    template <bool b, class T>
    using inherit_if_t = typename inherit_if<b, T>::type;

    template <class T, class...> using type_t = T;
    template <class...> inline constexpr bool false_v = false;
    template <class...> inline constexpr bool true_v = true;

    // 2.1.8, logical operator traits

    // 2.1.9, endian
}
```

2.1.3 Helper classes [meta.help]

```
namespace std {
    struct blank {};
}
```

1 The class template `blank` provides an empty class to be used in a wide range of context such as being a placeholder in conditional inheritance.

2.1.4 Unary type traits [meta.unary]

1 No modification.

2.1.4.1 Primary type categories [meta.unary.cat]

1 Add the following to the table “Primary type category predicates”:

Table 1 — Primary type category predicates

Template	Condition	Comments
`template<class T>` `struct is_closure;`	`T` is a closure type (see [expr.prim.lambda.closure]).	

2.1.4.2 Composite type traits [meta.unary.comp]

1 Add the following to the table “Composite type category predicates”:

Table 2 — Composite type category predicates

Template	Condition	Comments
template<class T> struct is_functor;	T is a (possibly union) class type with an overloaded operator().	
template<class T> struct is_function_object;	T is a function object (see [function.objects]).	This includes pointers to functions.
template<class T> struct is_callable;	T is a callable type (see [func.def]).	

2.1.4.3 Type properties [meta.unary.prop]

1 Add the following to the table "Type property predicates":

Table 3 — Type property predicates

Template	Condition	Preconditions
template<class T> struct is_inheritable;	It is possible to create a class U for which is_base_of_v<T, U> is true.	

2.1.5 Type property queries [meta.unary.prop.query]

1 No modification.

2.1.6 Relationships between types [meta.rel]

1 No modification.

2.1.7 Transformations between types [meta.trans]

2.1.7.1 Const-volatile modifications [meta.trans.cv]

1 Add the following to the table "Const-volatile modifications":

Table 4 — Const-volatile modifications

Template	Comments
template<class From, class To> struct copy_const;	The member typedef type names the same type as add_const_t<To> if is_const_v<From>, and To otherwise.
template<class From, class To> struct clone_const;	The member typedef type names the same type as copy_const_t<From, remove_const_t<To>>.
template<class From, class To> struct copy_volatile;	The member typedef type names the same type as add_volatile_t<To> if is_volatile_v<From>, and To otherwise.
template<class From, class To> struct clone_volatile;	The member typedef type names the same type as copy_volatile_t<From, remove_volatile_t<To>>.
template<class From, class To> struct copy_cv;	The member typedef type names the same type as copy_const_t<From, copy_volatile_t<From, To>>.
template<class From, class To> struct clone_cv;	The member typedef type names the same type as copy_cv_t<From, remove_cv_t<To>>.

2.1.7.2 Reference modifications [meta.trans.ref]

1 Add the following to the table “Reference modifications”:

Table 5 — Reference modifications

Template	Comments
`template<class From, class To> struct copy_reference;`	The member typedef `type` names the same type as `add_rvalue_reference_t<To>` if `is_rvalue_reference_v<From>`, `add_lvalue_reference_t<To>` if `is_lvalue_reference_v<From>`, and `To` otherwise.
`template<class From, class To> struct clone_reference;`	The member typedef `type` names the same type as `copy_reference_t<From, remove_reference_t<To>>`.

2.1.7.3 Sign modifications [meta.trans.sign]

1 Add the following to the table “Sign modifications”:

Table 6 — Sign modifications

Template	Comments
`template<class From, class To> struct copy_signedness;`	The member typedef `type` names the same type as `make_signed_t<To>` if `is_same_v<From, make_signed_t<From>>`, `make_unsigned_t<To>` if `is_same_v<From, make_unsigned_t<From>>`, and `To` otherwise. *Requires:* `From` and `To` shall be (possibly cv-qualified) integral types or enumerations but not `bool` types.

2.1.7.4 Array modifications [meta.trans.arr]

1 Add the following to the table “Array modifications”:

Table 7 — Array modifications

Template	Comments
`template<class From, class To> struct copy_extent;`	The member typedef `type` names the same type as `To[extent_v<From>]` if `rank_v<From> > 0 && extent_v<From> > 0`, `To[]` if `rank_v<From> > 0 && extent_v<From> == 0`, and `To` otherwise. *Requires:* `To` shall not be an array of unknown bound along its first dimension if `From` is an array of unknown bound along its first dimension.
`template<class From, class To> struct clone_extent;`	The member typedef `type` names the same type as `copy_extent_t<From, remove_extent_t<To>>`. *Requires:* `From` and `To` shall not be arrays of unknown bounds along their first dimension at the same time.

Table 7 — Array modifications (continued)

Template	Comments
`template<class From, class To>` `struct copy_all_extents;`	The member typedef `type` names the same type as `copy_extent_t<From, copy_all_extents_t<std::remove_extent_t<From>, To>>` if `rank_v<From> > 0`, and `To` otherwise. *Requires:* `From` and `To` shall not be arrays of unknown bounds along their first dimension at the same time.
`template<class From, class To>` `struct clone_all_extents;`	The member typedef `type` names the same type as `copy_all_extents_t<From, remove_all_extents_t<To>>`.

2.1.7.5 Pointer modifications [meta.trans.ptr]

1 Add the following to the table "Pointer modifications":

Table 8 — Pointer modifications

Template	Comments
`template<class T>` `struct remove_all_pointers;`	The member typedef `type` names the same type as `remove_all_pointers_t<remove_pointer_t<T>>` if `is_pointer_v<T>`, and `T` otherwise.
`template<class From, class To>` `struct copy_pointer;`	The member typedef `type` names the same type as `copy_cv_t<From, add_pointer_t<To>>` if `is_pointer_v<From>`, and `To` otherwise.
`template<class From, class To>` `struct clone_pointer;`	The member typedef `type` names the same type as `copy_pointer_t<From, remove_pointer_t<To>>`.
`template<class From, class To>` `struct copy_all_pointers;`	The member typedef `type` names the same type as `copy_pointer_t<From, copy_all_pointers_t<std::remove_pointer_t<From>, To>>` if `is_pointer_v<From>`, and `To` otherwise.
`template<class From, class To>` `struct clone_all_pointers;`	The member typedef `type` names the same type as `copy_all_pointers_t<From, remove_all_pointers_t<To>>`.

2.1.7.6 Other transformations [meta.trans.other]

1 Add the following to the table "Other transformations":

Table 9 — Other transformations

Template	Comments
`template<class From, class To>` `struct copy_cvref;`	The member typedef `type` names the same type as `copy_reference_t<From, copy_reference_t<To, copy_cv_t<remove_reference_t<From>, remove_reference_t<To>>>>`.
`template<class From, class To>` `struct clone_cvref;`	The member typedef `type` names the same type as `copy_cvref_t<From, remove_cvref<To>>`.

Table 9 — Other transformations (continued)

Template	**Comments**
`template <bool, class T>` `struct inherit_if;`	The member typedef `type` names the same type as `conditional_t<b, T, blank>>`. [*Example:* For `template <class T> struct derived: inherit_if_t<is_integral_v<T>, base> {};`, `is_base_of_v<base, derived>` will evaluate to `true` if `T` is an integral type, and to `false` otherwise. — *end example*]

2.1.8 Logical operator traits [meta.logical]

1 No modification.

2.1.9 Endian [meta.endian]

1 No modification.

A few additional type manipulation utilities

Vincent Reverdy

Summary

What?

Additional type traits for the `<type_traits>` header and corresponding to common metaprogramming patterns. Originally developed for a library to create custom overload sets (to be proposed separately).

Overview

5 domains: pointers removal, qualifiers manipulation, inheritance, callables and helpers.

Pointers removal

`remove_all_pointers`

Qualifiers copy

`copy_const`
`copy_volatile`
`copy_cv`
`copy_reference`
`copy_signedness`
`copy_extent`
`copy_all_extents`
`copy_pointer`
`copy_all_pointers`
`copy_cvref`

Qualifiers cloning

`clone_const`
`clone_volatile`
`clone_cv`
`clone_reference`
`clone_extent`
`clone_all_extents`
`clone_pointer`
`clone_all_pointers`
`clone_cvref`

Conditional inheritance

`blank`
`is_inheritable`
`inherit_if`

Callable categorization

`is_closure`
`is_functor`
`is_function_object`
`is_callable`

Helpers

`index_constant`
`type_t`
`false_v`
`true_v`

Pointers removal

Current pointer and extent transformation traits

```
template <class T> struct add_pointer;
template <class T> struct remove_pointer;
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

template <class T> using add_pointer_t = typename add_pointer<T>::type;
template <class T> using remove_pointer_t = typename remove_pointer<T>::type;
template <class T> using remove_extent_t = typename remove_extent<T>::type;
template <class T> using remove_all_extents_t = typename remove_all_extents<T>::type;
```

Synopsis of the proposed additions

```
template <class T> struct remove_all_pointers;
template <class T> using remove_all_pointers_t = typename remove_all_pointers<T>::type;
```

Motivations

- Symmetry with `remove_extent` and `remove_all_extents`
- As useful as `remove_all_extents`
- Completeness with qualifier manipulation traits (see next section)

Example

```
// Arrays
using arr0_t = int[2][3][4];
using arr1_t = remove_extent_t<arr0_t>;        // int[3][4]
using type_a = remove_all_extents_t<arr0_t>;  // int

// Pointers
using ptr0_t = int***;
using ptr1_t = remove_pointer_t<ptr0_t>;       // int**
using type_p = remove_all_pointers_t<ptr0_t>; // int
```

Qualifiers manipulation: synopsis

```
template <class From, class To> struct copy_const;
template <class From, class To> struct clone_const;
template <class From, class To> struct copy_volatile;
template <class From, class To> struct clone_volatile;
template <class From, class To> struct copy_cv;
template <class From, class To> struct clone_cv;
template <class From, class To> struct copy_reference;
template <class From, class To> struct clone_reference;
template <class From, class To> struct copy_signedness;
template <class From, class To> struct copy_extent;
template <class From, class To> struct clone_extent;
template <class From, class To> struct copy_all_extents;
template <class From, class To> struct clone_all_extents;
template <class From, class To> struct copy_pointer;
template <class From, class To> struct clone_pointer;
template <class From, class To> struct copy_all_pointers;
template <class From, class To> struct clone_all_pointers;
template <class From, class To> struct copy_cvref;
template <class From, class To> struct clone_cvref;

template <class F, class T> using copy_const_t = typename copy_const<F, T>::type;
template <class F, class T> using clone_const_t = typename clone_const<F, T>::type;
template <class F, class T> using copy_volatile_t = typename copy_volatile<F, T>::type;
template <class F, class T> using clone_volatile_t = typename clone_volatile<F, T>::type;
template <class F, class T> using copy_cv_t = typename copy_cv<F, T>::type;
template <class F, class T> using clone_cv_t = typename clone_cv<F, T>::type;
template <class F, class T> using copy_reference_t = typename copy_reference<F, T>::type;
template <class F, class T> using clone_reference_t = typename clone_reference<F, T>::type;
template <class F, class T> using copy_signedness_t = typename copy_signedness<F, T>::type;
template <class F, class T> using copy_extent_t = typename copy_extent<F, T>::type;
template <class F, class T> using clone_extent_t = typename clone_extent<F, T>::type;
template <class F, class T> using copy_all_extents_t = typename copy_all_extents<F, T>::type;
template <class F, class T> using clone_all_extents_t = typename clone_all_extents<F, T>::type;
template <class F, class T> using copy_pointer_t = typename copy_pointer<F, T>::type;
template <class F, class T> using clone_pointer_t = typename clone_pointer<F, T>::type;
template <class F, class T> using copy_all_pointers_t = typename copy_all_pointers<F, T>::type;
template <class F, class T> using clone_all_pointers_t = typename clone_all_pointers<F, T>::type;
template <class F, class T> using copy_cvref_t = typename copy_cvref<F, T>::type;
template <class F, class T> using clone_cvref_t = typename clone_cvref<F, T>::type;
```

Qualifiers manipulation: design

Functionality

Apply qualifiers or attributes of one type to another type.

Example

```
// Copy cv qualifiers
using type0 = copy_cv_t<const int, double>;                    // const double
using type1 = copy_cv_t<volatile int, double>;                 // volatile double
using type2 = copy_cv_t<const volatile int, double>;           // const volatile double

// Copy cv-ref qualifiers
using type3 = copy_cvref_t<int&, double>;                      // double&
using type4 = copy_cvref_t<volatile int&, double>;             // volatile double&
using type5 = copy_cvref_t<const volatile int&&, double>;      // const volatile double&&

// Copy vs clone
using type6 = copy_cvref_t<volatile int&, const double>;       // const volatile double&
using type7 = clone_cvref_t<volatile int&, const double>;      // volatile double&
using type8 = copy_all_pointers_t<int**, double***>;           // double*****;
using type9 = clone_all_pointers_t<int**, double***>;          // double**;
```

Design

- Two types of transformations: `copy_*` and `clone_*`
- `copy_*`: add the given qualifiers/attributes of `From` to `To`
- `clone_*`: apply the given qualifiers/attributes of `From` to `To` by first removing the given qualifiers/attributes of `To`
- Same list as existing transformation traits `add_*` and `remove_*`

Qualifiers manipulation: overview

Overview

	cv	reference	sign	array	pointer	cvref
remove_*	remove_const remove_volatile remove_cv	remove_reference		remove_extent remove_all_extents	remove_pointer remove_all_pointers	remove_cvref
add_*	add_const add_volatile add_cv	add_lvalue_reference add_rvalue_reference			add_pointer	
make_*			make_signed make_unsigned			
copy_*	copy_const copy_volatile copy_cv	copy_reference	copy_signedness	copy_extent copy_all_extents	copy_pointer copy_all_pointers	copy_cvref
clone_*	clone_const clone_volatile clone_cv	clone_reference		clone_extent clone_all_extents	clone_pointer clone_all_pointers	clone_cvref

Qualifiers manipulation: examples

Use case: manipulation of universal refs

```
template <class T, class U>
void f(T&& x, U&& y) {
    using type = T&&;
    using other = clone_cvref_t<T&&, U&&>;
    /* function contents */
}
```

Use case: in class templates

```
template <class T>
class foo {
    T a;
    copy_cvref_t<T, int> n;
    copy_cvref_t<T, double> x;
    /* class contents */
};
```

Use case: storing the qualifiers of a type

```
struct placeholder {};

template <class T>
struct qualifiers {
    using type = copy_cvref_t<T, placeholder>;
};

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

Use case: C array conversion

```
int array1[5][4][3][2];
using array_type = decltype(array1);
copy_all_extents_t<array_type, double> array2;
```

Use case: P0847R0: Deducing this

```
// Example provided in P0847R0
template <class From, class To> using like_t = clone_cvref_t<From, To>;

struct B {
    template <typename Self>
    auto&& f(Self&& this self) {
        return forward<Self>(*this).i;               // ok if Self and *this are the same type
                                                     // compile other if Self is a derived type
        return forward<like_t<Self, B>>(*this).i;    // always ok
        return forward_like<Self>(*this).i;          // always ok
    }
};
```

Qualifiers manipulation: about signedness

Current sign manipulators

- make_signed
- make_unsigned
- is_same_v<char, signed char> and is_same_v<char, unsigned char> are both false: therefore, contrarily to other integral types once make_signed or make_unsigned has been applied to char it is impossible to recover it easily (a remove_sign trait would be necessary)

Proposed behavior

```
using type0 = copy_signedness_t<unsigned int, char>;          // unsigned char
using type1 = copy_signedness_t<signed int, char>;            // signed char
using type2 = copy_signedness_t<char, unsigned int>;          // unsigned int
using type3 = copy_signedness_t<unsigned char, unsigned int>; // unsigned int
using type4 = copy_signedness_t<signed char, unsigned int>;   // signed int
using type5 = copy_signedness_t<char, unsigned char>;         // unsigned char

// using type6 = clone_signedness_t<char, unsigned char>;     // char (hypothetical)

using type7 = copy_signedness_t<signed char, unsigned int>;
// is equivalent to "make_signed_t<unsigned int>" since "signed unsigned int" would not compile
```

Qualifiers manipulation: discussion and open questions

Bikeshedding

- Alternative names for copy_*?
- Alternative names for clone_*?
- Alternative names for copy_signedness?

Remarks on copy_reference

copy_reference_t<T&, U&>, copy_reference_t<T&&, U&>, copy_reference_t<T&, U&&> and copy_reference_t<T&&, U&&> use reference collapsing rules to compute the resulting type. As clone_reference first removes the ref-qualifier of the second type, there is no need for reference collapsing in this case.

Remarks on copy_pointer

copy/clone_pointer and copy/clone_all_pointer copy cv-qualification of pointers:

```
using type = int* const* volatile** const volatile*;
using other = copy_all_pointers_t<type, double>; // double* const* volatile** const volatile*
```

Remarks on copy_signedness

- clone_signedness is not introduced because remove_sign does not exist
- The name copy_signedness is chosen because copysign already exists
- copy_signedness does not add a sign keyword (contrarily to the others copy_*) but uses make_signed and make_unsigned instead

Inheritance: summary

Synopsis of the proposed additions

```
struct blank;
template <class T> struct is_inheritable;
template <bool b, class T> struct inherit_if;

template <class T> inline constexpr bool is_inheritable_v = is_inheritable<T>::value;
template <bool b, class T> using inherit_if_t = typename inherit_if<b, T>::type;
```

blank: a general purpose empty class

```
struct blank {};
```

is_inheritable: to check if it is possible to inherit from a class

```
template <class T> struct is_inheritable: bool_constant<is_class_v<T> && !is_final_v<T>> {};
```

inherit_if: to enable the conditional inheritance pattern

```
template <bool b, class T> struct inherit_if {using type = conditional_t<b, T, blank>;};
```

inherit_if: use case

```
// Inherit if possible
template <class T>
struct foo: inherit_if_t<is_inheritable_v<T>, T> {
    /* class contents */
};

// Inheritance based on the properties of T
template <class T>
struct bar: inherit_if_t<is_integral_v<T>, foo<T>> {
    /* class contents */
};
```

Inheritance: discussion and open questions

Bikeshedding

- Alternative names for `blank`?
- Alternative names for `is_inheritable`?
- Alternative names for `inherit_if`?

Remarks on `blank`: need for a standardized empty class

3 mains options:

- `blank`: introduce a new empty class to become the standardized universal empty class for metaprogramming (the name being inspired from The Boost C++ Libraries)
- `monostate`: make `monostate` the standardized universal empty class (in that case it should be put in `<utility>` or `<type_traits>` instead of `<variant>`)
- `empty_base`: introduce a specialized empty class in the context of conditional inheritance so that `inherit_if_t` corresponds to `conditional_t<b, T, empty_base>`

Callables categorization: summary

Current traits related to callables

```
template <class T> struct is_function;
template <class T> struct is_member_function_pointer;
template <class T> struct is_member_object_pointer;

template <class Fn, class... ArgTypes> struct is_invocable;
template <class R, class Fn, class... ArgTypes> struct is_invocable_r;
template <class Fn, class... ArgTypes> struct is_nothrow_invocable;
template <class R, class Fn, class... ArgTypes> struct is_nothrow_invocable_r;
template <class Fn, class... ArgTypes> struct invoke_result;
template <class F, class... A> invoke_result_t<F, A...> invoke(F&& f, A&&... a) noexcept(/*..*/);
```

Synopsis of the proposed additions

```
template <class T> struct is_closure;
template <class T> struct is_functor;
template <class T> struct is_function_object;
template <class T> struct is_callable;

template <class T> inline constexpr bool is_closure_v = is_closure<T>::value;
template <class T> inline constexpr bool is_functor_v = is_functor<T>::value;
template <class T> inline constexpr bool is_function_object_v = is_function_object<T>::value;
template <class T> inline constexpr bool is_callable_v = is_callable<T>::value;
```

Callables categorization: motivations

Main motivations

- Make the custom overload set proposal possible (to be proposed separately)
- Detect types that are already defined in the standard but do not have traits yet
- Complete existing traits such as `is_function` and `is_member_function_pointer`
- Have a trait associated with the concept satisfied by the first argument of `invoke` (suggested by P0604R0: Resolving GB 55, US 84, US 85, US 86 but left for later)
- Distinguish between the different categories of callable types

Examples of use cases

```
// Definition and use of the callable concept
template <class T>
concept Callable = is_callable_v<T>::value;
template <Callable F, class... Args>
invoke_result_t<F, Args...> invoke(F&& f, Args&&... args) noexcept(/*...*/);

// Inheriting from a functor
template <class F, class = void> class functor;
template <class F>
class functor<F, enable_if_t<is_functor_v<F> && is_inheritable_v<F>>>
: F {public: using F::operator(); /* Only working for an inheritable functor */};

// Check if a template class will have a unique key
template <class T, class Key>
struct wrapper {
    using key_t = Key;
    template <class T> constexpr wrapper(const T& value, const Key& key) {}
};
wrapper(4, []{});
constexpr bool b = is_closure_v<typename decltype(wrapper)::key_t>;
```

Callables categorization: definitions

Existing definition of a closure type [expr.prim.lambda.closure]

The type of a lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type, called the **closure type**, whose properties are described below. [...]

Existing definition of a function object type [function.objects]

A **function object type** is an object type that can be the type of the postfix-expression in a function call. A function object is an object of a function object type.

Existing definition of a callable type [func.def]

The following definitions apply to this Clause:

- A call signature is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types
- A **callable type** is a function object type or a pointer to member
- A callable object is an object of a callable type
- A call wrapper type is a type that holds a callable object and supports a call operation that forwards to that object
- A call wrapper is an object of a call wrapper type
- A target object is the callable object held by a call wrapper

Callables categorization: function objects and functors

Definition of function objects and functors [Wikipedia]

In computer programming, a **function object** is a construct allowing an object to be invoked or called as if it were an ordinary function, usually with the same syntax (a function parameter that can also be a function). Function objects are often called **functors**.

In C++ pointers to functions are of function object types

"A **function object type** is an object type that can be the type of the postfix-expression in a function call." Since that for a type `P` such that `is_pointer_v<P>` evaluates to `true`, `is_object_v<P>` also evaluates to `true`, and since pointers to functions can be of the type of the postfix-expression in a function call, pointers to functions are considered to be of function object types.

Misleading for non-expert users

As observed on StackOverflow, this is very misleading for non-expert users: they would expect function object types to be class types with an overloaded `operator()`.

Introducing a functor trait

A type trait `is_functor` is introduced on the top of `is_function_object`:

- `is_functor`: a (possibly union) class type with an overloaded `operator()`
- `is_function_object`: an object type that can be the type of the postfix-expression in a function call [function.objects]

Callables categorization: discussion and open questions

Remarks on `is_functor`

Should only public overloads of `operator()` be considered, or also protected and private ones?

Remarks on `is_callable`

Should references to callables be also considered as callables?

Miscellaneous helpers: summary

Current helpers related to the new ones

```
// Integral constant and integer sequence
template<class T, T v> struct integral_constant;
template<bool B> using bool_constant = integral_constant<bool, B>;
using true_type = bool_constant<true>;
using false_type = bool_constant<false>;

// Integer sequence
template<class T, T...> struct integer_sequence;
template<size_t... I> using index_sequence = integer_sequence<size_t, I...>;
template<class T, T N> using make_integer_sequence = integer_sequence<T, see below >;
template<size_t N> using make_index_sequence = make_integer_sequence<size_t, N>;
template<class... T> using index_sequence_for = make_index_sequence<sizeof...(T)>;

// SFINAE
template<class...> using void_t = void;
```

Synopsis of the proposed additions

```
template <size_t I> using index_constant = integral_constant<size_t, I>;
template <class T, class...> using type_t = T;
template <class...> inline constexpr bool false_v = false;
template <class...> inline constexpr bool true_v = true;
```

Motivations for `index_constant`

- Symmetry of `integral_constant` with `integer_sequence` and `index_sequence`
- As useful as `index_sequence` since specifying a type corresponding to a constant size is a common need

Miscellaneous helpers: false_v and true_v

Motivations for false_v and true_v

- Avoid hard errors in template dependent contexts
- Example of use with static_assert

Use case for false_v

```cpp
// Example provided by Arthur O'Dwyer
template<class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor {
    public:
    using oalloc_t = OuterAlloc;
    using ialloc_t = scoped_allocator_adaptor<InnerAllocs...>;

    private:
    oalloc_t _outer;
    ialloc_t _inner;
    using _otraits = allocator_traits<OuterAlloc>;

    public:
    template<class... Args>
    void construct(value_type* p, Args&&... args) {
        if constexpr (!uses_allocator_v<value_type, ialloc_t>) {
            _otraits::construct(_outer, p, forward<Args>(args)...);
        } else if constexpr (is_constructible_v<value_type, alloc_arg_t, ialloc_t, Args&&...>) {
            _otraits::construct(_outer, p, allocator_arg, _inner, forward<Args>(args)...);
        } else if constexpr (is_constructible_v<value_type, Args&&..., ialloc_t&>) {
            _otraits::construct(_outer, p, forward<Args>(args)..., _inner);
        } else {
            // static_assert(false, "value_type is not constructible from args"); // not working
            static_assert(false_v<Args&&...>, "value_type is not constructible from args");
        }
    }
};
```

Miscellaneous helpers: type_t

Complements to void_t: the logic behind it

```
// Link between type_t, void_t, false_v and true_v
template <class T, class...> using type_t = T;
template <class... Ts> using void_t = type_t<void, Ts...>;
template <class... Ts> inline constexpr bool false_v = type_t<false_type, Ts...>::value;
template <class... Ts> inline constexpr bool true_v = type_t<true_type, Ts...>::value;
```

Motivations for type_t

- Simple generalization of `void_t`
- Useful to combine SFINAE with the return of a type
- Powerful when combined with the custom overload set proposal

Basic use case of type_t

```
template <class T, class U> struct foo;
template <class T, class U> struct foo<T, type_t<U, decltype(declval<T>() + declval<U>())>> {
    static constexpr auto value = "declval<T>() + declval<U>()";
};
template <class T, class U> struct foo<T, type_t<U, decltype(declval<T>().size())>> {
    static constexpr auto value = "declval<T>().size()";
};
int main() {
    cout << foo<double, int>::value << endl; // declval<T>() + declval<U>()
    cout << foo<string, int>::value << endl; // declval<T>().size()
}
```

Bikeshedding

- Alternative names for `type_t`?
- Alternative names for `false_v` and `true_v`?

Conclusion: overview of design decisions

Bikeshedding

- Alternative names for `copy_*`?
- Alternative names for `clone_*`?
- Alternative names for `copy_signedness`?
- Alternative names for `blank`?
- Alternative names for `is_inheritable`?
- Alternative names for `inherit_if`?
- Alternative names for `type_t`?
- Alternative names for `false_v` and `true_v`?

Main open questions and remarks

- `copy/clone_pointer` and `copy/clone_all_pointer`: copy cv-qualification of pointers
- `copy_signedness` does not add a sign keyword (contrarily to the others `copy_*`) but uses `make_signed` and `make_unsigned` instead
- `clone_signedness` is not introduced because `remove_sign` does not exist
- `blank` vs `monostate` vs `empty_base`: need for an empty base class
- `is_functor`: should only public overloads of `operator()` be considered, or also protected and private ones?
- `is_callable`: should references to callables be also considered as callables?

Conclusion: overview of functionalities

	SF	F	N	A	SA
remove_all_pointers					
copy_*/clone_*					
copy_signedness					
blank					
is_inheritable					
inherit_if					
is_closure					
is_functor					
is_function_object					
is_callable					
index_constant					
type_t					
false_v/true_v					

Pointers removal

remove_all_pointers

Qualifiers copy

copy_const
copy_volatile
copy_cv
copy_reference
copy_signedness
copy_extent
copy_all_extents
copy_pointer
copy_all_pointers
copy_cvref

Qualifiers cloning

clone_const
clone_volatile
clone_cv
clone_reference
clone_extent
clone_all_extents
clone_pointer
clone_all_pointers
clone_cvref

Conditional inheritance

blank
is_inheritable
inherit_if

Callable categorization

is_closure
is_functor
is_function_object
is_callable

Helpers

index_constant
type_t
false_v
true_v

Thank you for your attention