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. Implementation available at `https://github.com/vreverdy/type-utilities`.

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`

Inheritance

`empty_base`
`is_inheritable`
`is_instantiable`
`inherit_if`

Callables categorization

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

Miscellaneous 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

```cpp
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

```
template <class... T> struct empty_base;
template <class T> struct is_inheritable;
template <class T> struct is_instantiable;
template <bool b, class T> struct inherit_if;

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

empty_base: a general purpose templated empty base class

```
template <class... T> struct empty_base {};
```

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>> {};
```

is_instantiable: to check if it is possible to instantiate an object of type T

```
template <class T> struct is_instantiable: bool_constant</* TBD */> {};
```

inherit_if: to enable the conditional inheritance pattern

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

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 `empty_base`?
- Alternative names for `is_inheritable`?
- Alternative names for `is_instantiable`?
- Alternative names for `inherit_if`?

Remarks on `empty_base`: 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<T>>`

Remarks on `is_instantiable`

What should the exact definition be?

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`

```
// 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 empty_base?
- Alternative names for is_inheritable?
- Alternative names for is_instantiable?
- 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
- 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 class
- is_instantiable: what should the exact definition be?
- is_functor: should only public overloads of operator() be considered?
- 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					
empty_base					
is_inheritable					
is_instantiable					
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

Inheritance

empty_base
is_inheritable
is_instantiable
inherit_if

Callables categorization

is_closure
is_functor
is_function_object
is_callable

Miscellaneous helpers

index_constant
type_t
false_v
true_v

Thank you for your attention