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

```
template <class From, class To> using like_t = clone_cvref_t<From, To>;

struct B {
    template < typename Self>
    auto&& f3(Self&& this self) {
        return std::forward<Self>(*this).i;           // ok if Self and *this are the same type
                                                      // compile other if Self is a derived type
        return std::forward<like_t<Self, B>>(*this).i;// always ok
        return std::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

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;
```

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;
```