Enriching type modification traits

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 focused on type modification. The new type traits we present considerably simplify qualifiers manipulation. We also introduce a new type trait to remove all pointers on a type for the sake of completeness. These type traits have been especially useful in the design of proxy classes, included an updated design for bit manipulation utilities. They also have been used extensively in the implementation of a library dedicated to the creation of custom overload sets that will be proposed for standardization in a separate proposal.

Contents

1 Proposal  
  1.1 Introduction .................................................. 1
  1.2 Background ................................................... 1
  1.3 Impact on the standard ......................................... 1
  1.4 Motivations and design decisions ............................... 1
    1.4.1 Pointers removal ........................................... 1
    1.4.2 Qualifiers manipulation .................................. 2
    1.4.3 Combined removal and qualifiers manipulation .......... 3
  1.5 Technical specification ........................................ 4
  1.6 Discussion and open questions ................................ 4
    1.6.1 Bikeshedding .............................................. 4
  1.7 Acknowledgements ............................................... 4
  1.8 References ..................................................... 4

2 Wording  
  2.1 Metaprogramming and type traits .............................. 5
    2.1.1 Requirements ............................................. 5
    2.1.2 Header <type_traits> synopsis ........................... 5
    2.1.3 Helper classes ............................................ 6
    2.1.4 Unary type traits ......................................... 7
    2.1.5 Type property queries ..................................... 7
    2.1.6 Relationships between types .............................. 7
    2.1.7 Transformations between types ............................ 7
    2.1.8 Logical operator traits .................................. 9
    2.1.9 Endian .................................................. 10

3 Presentation  


1 Proposal

1.1 Introduction

Since their introduction with C++11, the standard library type traits 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 to implement template proxy classes as well as to implement 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 fall in three different categories:

- pointers removal: `remove_all_pointers` inspired from `remove_all_extents`
- qualifiers manipulation: `copy_*` type traits
- combined removal and qualifiers manipulation: `clone_*` type traits

An implementation is available at [https://github.com/vreverdy/type-utilities](https://github.com/vreverdy/type-utilities).

1.2 Background

These type traits have been proposed before in P1016R0. However, back then it was not clear whether this kind of type traits should wait for reflection. In San Diego, SG7 clarified that these type traits are pure library facilities that do not need to be first reviewed by them. They also clarified the fact that type traits in their current form live in a different space than reflection, and that the second one will not make the first one disappear. As a consequence, basic type traits should not wait for reflection. In San Diego, LEWGI recommended the extraction of simple type modification traits from the original P1016R0 proposal, which led to this proposal in its current form.

1.3 Impact on the standard

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.

1.4 Motivations and design decisions

1.4.1 Pointers removal

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. For pointers, only one is currently provided: `remove_pointer` which removes one pointer. However, in some contexts it can be useful to access the “raw” type:

```cpp
§ 1.4.1
```

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 (1.4.2) and (1.4.3)), 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: \texttt{remove\_all\_pointers}.

### 1.4.2 Qualifiers manipulation [proposal.design.copy]

```
// Qualifiers manipulation
template <class From, class To> struct copy_const;
template <class From, class To> struct copy_volatile;
template <class From, class To> struct copy_cv;
template <class From, class To> struct copy_reference;
template <class From, class To> struct copy_signedness;
template <class From, class To> struct copy_extent;
template <class From, class To> struct copy_all_extents;
template <class From, class To> struct copy_pointer;
template <class From, class To> struct copy_all_pointers;
```

In the heavy template metaprogramming involved in the building of template proxy classes and 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 \texttt{const int} into a \texttt{const double}, a \texttt{int[1][2][3]} into \texttt{double[1][2][3]}, or an \texttt{int***} to a \texttt{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 \texttt{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 \texttt{P0847R0}, where \texttt{copy_cvref_t} is called \texttt{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 \texttt{copy\_*} and the cloners \texttt{clone\_*} presented in the next section.

The complete list of proposed \texttt{copy\_*} traits is:

§ 1.4.2
— const-volatile modifications: copy_const, copy_volatile, copy_cv
— reference modifications: copy_reference
— sign modifications: copy_signedness
— array modifications: copy_extent, copy_all_extents
— pointer modifications: copy_pointer copy_all_pointers
— other transformations: copy_cvref

As a note, in the same way remove_pointer deals with cv-qualified pointers, copy_pointer copy_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.

1.4.3 Combined removal and qualifiers manipulation [proposal.design.clone]

// Combined removal and qualifiers manipulation
template <class From, class To> struct clone_const;
template <class From, class To> struct clone_volatile;
template <class From, class To> struct clone_cv;
template <class From, class To> struct clone_reference;
template <class From, class To> struct clone_extent;
template <class From, class To> struct clone_all_extents;
template <class From, class To> struct clone_pointer;
template <class From, class To> struct clone_all_pointers;
template <class From, class To> struct clone_cvref;

// Type aliases
template <class F, class T> using clone_const_t = typename clone_const<F, T>::type;
template <class F, class T> using clone_volatile_t = typename clone_volatile<F, T>::type;
template <class F, class T> using clone_cv_t = typename clone_cv<F, T>::type;
template <class F, class T> using clone_reference_t = typename clone_reference<F, T>::type;
template <class F, class T> using clone_signedness_t = typename clone_signedness<F, T>::type;
template <class F, class T> using clone_extent_t = typename clone_extent<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 clone_pointer_t = typename clone_pointer<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 clone_cvref_t = typename clone_cvref<F, T>::type;

When the second template parameter is also coming from a context where it can be qualified, it can be useful to first remove its qualifiers before copying the new one. The difference between cloners and copiers 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***.

For example:

template <class T> struct foo {
    // Function member
    template <class U> void bar (U& x) {
        std::clone_cvref_t<T, U> something;
        /* function contents */
    }
    /* class contents */
};
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).

The complete list of proposed clone_* traits is:

- const-volatile modifications: clone_const, clone_volatile, clone_cv
- reference modifications: clone_reference
- array modifications: clone_extent, clone_all_extents
- pointer modifications: clone_pointer, clone_all_pointers
- other transformations: clone_cvref

As a note, in the same way remove_pointer deals with cv-qualified pointers, clone_pointer clone_all_pointers clone the cv-qualifiers of pointers. 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.

1.5 Technical specification [proposal.spec]
See the wording (part 2).

1.6 Discussion and open questions [proposal.discussion]
1.6.1 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

1.7 Acknowledgements [proposal.ackwldgmnts]
The authors would like to thank the participants to the related discussion on the future-proposals group. This work has been made possible thanks to the National Science Foundation through the awards CCF-1647432 and SI2-SSE-1642411.

1.8 References [proposal.references]
A few additional type manipulation utilities, Vincent Reverdy, Github (March 2018)
P1016R0, A few additional type manipulation utilities, Vincent Reverdy, ISO/IEC JTC1/SC22/WG21 (May 2018)
P0847R0, Deducing this, Gasper Azman et al., ISO/IEC JTC1/SC22/WG21 (February 2018)
General purpose utilities for template metaprogramming and type manipulation, ISO C++ Standard - Future Proposals, Google Groups (March 2018)
2  Wording

2.1  Metaprogramming and type traits

2.1.1  Requirements

1 No modification.

2.1.2  Header <type_traits> synopsis

Add the following to the synopsis of <type_traits>:

```cpp
namespace std {
    // 2.1.3, helper classes
    // 2.1.4.1, primary type categories
    // 2.1.4.2, composite type categories
    // 2.1.4.3, type properties
    // 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;

} // namespace std
```

§ 2.1.2
using clone_reference_t = typename clone_reference<From, To>::type;

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

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;

// 2.1.8, logical operator traits

// 2.1.9, endian

} }

2.1.3 Helper classes

1 No modification.
2.1.4 Unary type traits [meta.unary]
1 No modification.

2.1.4.1 Primary type categories [meta.unary.cat]
1 No modification.

2.1.4.2 Composite type traits [meta.unary.comp]
1 No modification.

2.1.4.3 Type properties [meta.unary.prop]
1 No modification.

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>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
&\text{template}\langle\text{class } \text{From}, \text{ class } \text{To}\rangle \\
&\text{struct } \text{copy}\_\text{const}; \\
\end{align*}
\] | The member typedef type names the same type as add_const_t<\text{To}> if is_const_v<\text{From}>, and \text{To} otherwise. |
| \[
\begin{align*}
&\text{template}\langle\text{class } \text{From}, \text{ class } \text{To}\rangle \\
&\text{struct } \text{clone}\_\text{const}; \\
\end{align*}
\] | The member typedef type names the same type as copy_const_t<\text{From}, remove_const_t<\text{To}>>. |
| \[
\begin{align*}
&\text{template}\langle\text{class } \text{From}, \text{ class } \text{To}\rangle \\
&\text{struct } \text{copy}\_\text{volatile}; \\
\end{align*}
\] | The member typedef type names the same type as add_volatile_t<\text{To}> if is_volatile_v<\text{From}>, and \text{To} otherwise. |
| \[
\begin{align*}
&\text{template}\langle\text{class } \text{From}, \text{ class } \text{To}\rangle \\
&\text{struct } \text{clone}\_\text{volatile}; \\
\end{align*}
\] | The member typedef type names the same type as copy_volatile_t<\text{From}, remove_volatile_t<\text{To}>>. |
| \[
\begin{align*}
&\text{template}\langle\text{class } \text{From}, \text{ class } \text{To}\rangle \\
&\text{struct } \text{copy}\_\text{cv}; \\
\end{align*}
\] | The member typedef type names the same type as copy_const_t<\text{From}, copy_volatile_t<\text{From}, \text{To}>>. |
| \[
\begin{align*}
&\text{template}\langle\text{class } \text{From}, \text{ class } \text{To}\rangle \\
&\text{struct } \text{clone}\_\text{cv}; \\
\end{align*}
\] | The member typedef type names the same type as copy_cv_t<\text{From}, remove_cv_t<\text{To}>>. |

2.1.7.2 Reference modifications [meta.trans.ref]
1 Add the following to the table “Reference modifications”:

§ 2.1.7.2
Table 2 — Reference modifications (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| `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

Add the following to the table “Sign modifications”:

Table 3 — Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| `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

Add the following to the table “Array modifications”:

Table 4 — Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| `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 4 — Array modifications (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct copy_all_extents;</td>
<td>copy_extent_t&lt;From, copy_all_extents_t&lt;std::remove_extent_t&lt;From&gt;, To&gt;&gt;, if rank_v&lt;From&gt; &gt; 0,</td>
</tr>
<tr>
<td></td>
<td>and To otherwise.</td>
</tr>
<tr>
<td></td>
<td>Requires: From and To shall not be arrays of unknown bounds along their first dimension at</td>
</tr>
<tr>
<td></td>
<td>the same time.</td>
</tr>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct clone_all_extents;</td>
<td>copy_all_extents_t&lt;From, remove_all_extents_t&lt;To&gt;&gt;.</td>
</tr>
</tbody>
</table>

2.1.7.5 Pointer modifications

Add the following to the table “Pointer modifications”:

Table 5 — Pointer modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct remove_all_pointers;</td>
<td>remove_all_pointers_t&lt;remove_pointer_t&lt;T&gt;&gt;, if is_pointer_v&lt;T&gt;, and T otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct copy_pointer;</td>
<td>copy_cv_t&lt;From, add_pointer_t&lt;To&gt;&gt;, if is_pointer_v&lt;From&gt;, and To otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct clone_pointer;</td>
<td>copy_pointer_t&lt;From, remove_pointer_t&lt;To&gt;&gt;.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct copy_all_pointers;</td>
<td>copy_all_pointers_t&lt;From, remove_all_pointers_t&lt;To&gt;&gt;.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct clone_all_pointers;</td>
<td>copy_all_pointers_t&lt;From, remove_all_pointers_t&lt;To&gt;&gt;.</td>
</tr>
</tbody>
</table>

2.1.7.6 Other transformations

Add the following to the table “Other transformations”:

Table 6 — Other transformations

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct copy_cvref;</td>
<td>copy_reference_t&lt;From, copy_reference_t&lt;To&gt;, copy_cv_t&lt;remove_reference_t&lt;From&gt;,</td>
</tr>
<tr>
<td></td>
<td>remove_reference_t&lt;To&gt;&gt;, remove_reference_t&lt;To&gt;&gt;.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class From, class To&gt;</td>
<td>The member typedef type names the same type as</td>
</tr>
<tr>
<td>struct clone_cvref;</td>
<td>copy_cvref_t&lt;From, remove_cvref&lt;T&gt;&gt;, remove_cvref&lt;T&gt;&gt;.</td>
</tr>
</tbody>
</table>

2.1.8 Logical operator traits

No modification.
2.1.9 Endian

1 No modification.
Enriching type modification traits

Vincent Reverdy
From Metaprogramming Tricks to Elegance: Custom Overload Sets and Inline SFINAE for Truly Generic Interfaces

- **is_valid**
  - if this code compiles
  - then do this

- **overload_sequence**
  - a list of functions
  - call the first function compatible with arguments

- **overload_set**
  - a list of functions
  - a list of arguments
  - call the best function for the provided arguments

- **type_selector**
  - a list of types
  - a list of expressions
  - the first type that produces valid code when replaced in the expressions
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
3 domains: pointers removal, qualifiers manipulation, combined removal and qualifiers manipulation.
**Pointers removal**

**Current pointer and extent transformation traits**

```cpp
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**

```cpp
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**

```cpp
// 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 clone_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 clone_signedness_t = typename clone_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

```c++
// 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

<table>
<thead>
<tr>
<th></th>
<th>cv</th>
<th>reference</th>
<th>sign</th>
<th>array</th>
<th>pointer</th>
<th>cvref</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>remove_</strong>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>remove_cvref</td>
</tr>
<tr>
<td>remove_const</td>
<td>remove_volatile</td>
<td>remove_cv</td>
<td>remove_reference</td>
<td>remove_extent</td>
<td>remove_pointer</td>
<td></td>
</tr>
<tr>
<td><strong>add_</strong>*</td>
<td>add_const</td>
<td>add_volatile</td>
<td>add_lvalue_reference</td>
<td>add_rvalue_reference</td>
<td>add_pointer</td>
<td></td>
</tr>
<tr>
<td>add_cv</td>
<td>add_cv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>make_</strong>*</td>
<td>make_signed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>make_unsigned</td>
</tr>
<tr>
<td><strong>copy_</strong>*</td>
<td>copy_const</td>
<td>copy_volatile</td>
<td>copy_reference</td>
<td>copy_signedness</td>
<td>copy_extent</td>
<td>copy_pointer</td>
</tr>
<tr>
<td>copy_cv</td>
<td>copy_cv</td>
<td></td>
<td></td>
<td></td>
<td>copy_all_extents</td>
<td>copy_all_pointers</td>
</tr>
<tr>
<td><strong>clone_</strong>*</td>
<td>clone_const</td>
<td>clone_volatile</td>
<td>clone_reference</td>
<td>clone_signedness</td>
<td>clone_extent</td>
<td>clone_pointer</td>
</tr>
<tr>
<td>clone_cv</td>
<td>clone_cv</td>
<td></td>
<td></td>
<td></td>
<td>clone_all_extents</td>
<td>clone_all_pointers</td>
</tr>
</tbody>
</table>

- remove_: remove const, volatile, cv
- add_: add const, volatile, cv
- make_: make signed, unsigned
- copy_: copy const, volatile, cv
- clone_: clone const, volatile, cv

The table provides an overview of how different manipulations affect the cv, reference, sign, array, pointer, and cvref qualifiers.
Qualifiers manipulation: examples

**Use case: manipulation of universal refs**

```cpp
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**

```cpp
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**

```cpp
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**

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

**Use case: P0847R0: Deducing this**

```cpp
// 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
        return forward<like_t<Self, B>>(*this).i; // always ok
        return forward_like<Self>(*this).i; // always ok
    }
};
```
Qualifiers manipulation: examples

Use case: wrapper with a wrapped data member

```cpp
template <class T>
class wrapper {
    /* ... */

    // Data members
    private:
    other_wrapper<copy_cvref_t<T, foo>> _wrapped;
};
```

Use case: class template argument deduction and universal references

```cpp
template <class T>
class something {
    template <class U, class = std::enable_if_t</* ... */>>
    explicit constexpr something(U&& x);
};
```

```cpp
template <class U, class = std::enable_if_t</* ... */>>
something(U&&) -> something<copy_cvref_t<U&&, foo>>;
```
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 \textit{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

```cpp
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
```
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:

```cpp
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
Conclusion: overview of design decisions

Bikeshedding

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

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
Conclusion: overview of functionalities

<table>
<thead>
<tr>
<th>Pointers removal</th>
<th>Qualifiers copy</th>
<th>Qualifiers cloning</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove_all_pointers</td>
<td>copy_const</td>
<td>clone_const</td>
</tr>
<tr>
<td>copy_<em>/clone_</em></td>
<td>copy_volatile</td>
<td>clone_volatile</td>
</tr>
<tr>
<td>copy_signedness</td>
<td>copy_cv</td>
<td>clone_cv</td>
</tr>
<tr>
<td>copy_reference</td>
<td>copy_signedness</td>
<td>clone_reference</td>
</tr>
<tr>
<td>copy_extent</td>
<td>copy_all_extents</td>
<td>clone_extent</td>
</tr>
<tr>
<td>copy_all_pointers</td>
<td>copy_pointer</td>
<td>clone_all_extents</td>
</tr>
<tr>
<td>copy_cvref</td>
<td></td>
<td>clone_pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clone_all_pointers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clone_cref</td>
</tr>
</tbody>
</table>
Thank you for your attention