Document Number: P2091R0
Date: 2020-02-12
Audience: Library Working Group
Author: Casey Carter
Reply to: casey@carter.net

Issues with range access CPOs

1 Abstract

This paper began as a Proposed Resolution for GB275 "`ranges::begin/end` should not accept arrays of unknown bound," and grew to cover several problems with the range access customization point objects. Each problem is small enough to be addressed by an LWG issue, but as they are closely related (if not intertwined) it seems best to process them as a unit.

2 Discussion

The following largely omits discussion of the constant variations of the range access CPOs `cbegin`, `cend`, `crbegin`, `crend`, and `cdata` for brevity despite that the arguments presented for each CPO apply similarly to its corresponding constant variation. Imagine that occurrences of e.g. "`begin`" below are instead occurrences of "`begin` (and `cbegin`)."

2.1 arrays of unknown bound

GB275 states ([2]):

> `ranges::begin` and `ranges::end` should not accept arrays of unknown bound. The current definitions of `ranges::begin` and `ranges::end` mean that an array of unknown bound is treated as an empty range. The expressions `E + 0` and `E + extent_v<T>` are both well-formed for [an expression `E` with type `T` which is] an array of unknown bound (with `extent_v<T>` equal to zero).
>
> Proposed change:
> Make `ranges::begin(E)` and `ranges::end(E)` ill-formed when `E` is an array of unknown bound.

In private communication before Belfast, the author and Jonathan Wakely discussed arrays of unknown bound and their interactions with the range access CPOs extensively and came to the conclusion that `begin` and `data` should accept them to preserve the reasonable behavior of `std::begin` and `std::data`, but `end` and `size` should not. It seems logical to relax the `range` constraint on `ranges::iterator_t` as well, so it can continue to express "the type that `ranges::begin` returns" even in this non-`range` case.

`empty` is also easily supported despite the lack of extent since arrays cannot have zero extent.

2.2 arrays of elements with incomplete type

The author and Mr. Wakely also discussed how the range access CPOs should handle arrays whose elements have incomplete type. Such array types were never considered during the evolution of Ranges, so any status quo behavior is unintended.

`size` and `empty` make sense for these arrays since we can determine the extent. `begin` and `data` could be made to work, but would nonetheless likely produce problems elsewhere in the program since pointers-to-incomplete-type fail to satisfy `weakly_incrementable` and therefore all of the iterator concepts including notably `contiguous_iterator`. Supporting arrays of incomplete type in the other CPOs seems like a recipe for disaster: pointer arithmetic doesn't work, and given that the element type may be completed, we're skirting the ODR. Factoring in the instability around memoization of concepts (See GB046 [1]), we should avoid attempting to support these types in other CPOs for now.

2.3 `safe_ranges` and `ranges::data`

`ranges::begin` avoids returning dangling iterators with the "lvalue or `safe_range`" test, but `ranges::data` happily returns dangling pointers for rvalue non-`safe_range`s. There's no particular reason for this, other than the fact that P0970R1 "Better, Safer Range Access Customization Points" [7] and P0944 "Contiguous Ranges" [3] were processed at the same time and never reconciled. This seems like an oversight we can easily correct now by requiring "lvalue or `safe_range`" arguments in `ranges::data` just as we already do for `ranges::begin`.

2.4 LWG-3258 and poison pills

LWG-3258 "Range access and `initializer_list`" [4] proposes a change to the `initializer_list` poison pills for `begin` and `end`, and the addition of similar poison pills to `rbegin` and `rend`. This was a great idea, until P1870R1 "*`forwarding-range`*`<T>` is too subtle" [8], changed the opt-in mechanism for *`forwarding-range`*`<T>` (now `safe_range`) from "do rvalues work?" to "is this trait specialized?". We could and should have removed the `initializer_list` poison pills in P1870R1, but did not - so let's do so now.

2.5 P1870 and the advantages of lvalue dispatch

P1870R1 also modified the design of `begin`, `end`, `rbegin`, and `rend`: these CPOs now only perform lookup and dispatch with lvalues bound to their argument expression. For example, `std::ranges::begin(std::span{some_array_argument})` binds a reference we'll call `t` to the result of materializing the prvalue `std::span{some_array_argument}`, determines that `t.begin()` is a valid expression whose decayed type models `input_or_output_iterator`, and then returns `t.begin()`. Always performing lookup and dispatch with lvalues in this fashion makes it easier to reason about, implement, and specify the CPOs. We should specify the others similarly.

While implementing this change, I realized that the forwarding-reference poison pills in the working draft are insufficiently poisonous. `void foo(auto&&)` is less-specialized than either `void foo(auto&)` or `void foo(const auto&)`, so a `void foo(auto&&)` poison pill fails to intercept/ambiguate calls to such overgeneric lvalue functions as intended. We should fix the poison pills by replacing them with two lvalue overloads. (I'm not certain the poison pills serve a useful design purpose anymore, and I'd like to remove them, but it's too late in the cycle for even so small a design change.)

2.6 ADL only for class/enumeration types

Just as LWG-3299 doesn't want users to specify non-pointer iterator behaviors for pointer-to-program-defined-type ([5]), we don't intend for users to specify non-array range behaviors for array-of-program-defined-type. It's similarly not intended that users specify range behaviors for functions. We should forbid such silliness in the range access CPOS just as LWG-3299 does for the iterator machinery by constraining ADL cases to expressions of class or enumeration type.

2.7 Editorial Feedback

During the merge of P1870R1, the Project Editor expressed dissatisfaction with a couple of phrases used widely in the CPO wording:

— "an lvalue `t` that denotes the same object as [subexpression] `E`" doesn't make sense when `E` is a prvalue since prvalues don't "denote an object." We should instead say that `T` denotes the result of applying the temporary materialization conversion to `E` when `E` is a prvalue.

— "ranges::begin(E) is ill-formed" is not something to which [an expression] can be expression-equivalent. This category error is repeated in every CPO specification (sometimes twice after application of P1870R1).

3 Implementation Experience

The proposed changes have been implemented in Microsoft's STL (See https://github.com/microsoft/STL/pull/432).

4 Technical Specifications

The technical specifications that follow take the form of excerpts from the working draft with change markings:

— ~~Text to be struck is in red with strikethough~~, and

— text to be added is “green” with underline.

Note that these specifications supersede the proposed resolution of LWG-3258 and include the proposed resolution of LWG-3368 “Exactly when does `size` return `end - begin`?” [6].

24 Ranges library [ranges]

[...]

24.2 Header `<ranges>` synopsis [ranges.syn]

[...]

```
template<~~range R~~class T>
  using iterator_t = decltype(ranges::begin(declval<~~R~~T&>()));
```

[...]

24.3 Range access [range.access]

[...]

24.3.1 `ranges::begin` [range.access.begin]

1 The name `ranges::begin` denotes a customization point object ([customization.point.object]).

2 Given a subexpression `E` ~~and an lvalue~~with type `T`, let `t` be an lvalue that denotes the same object as `E`~~,~~ if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — ~~if~~If `E` is an rvalue and `enable_safe_range<`~~`remove_cvref_t<decltype((E))>`~~`remove_cv_t<T>>` is `false`, `ranges::begin(E)` is ill-formed. ~~Otherwise, `ranges::begin(E)` is expression-equivalent to:~~

(2.2) — Otherwise, if `T` is an array type ([basic.compound]) and `remove_all_extents_t<T>` is an incomplete type, `ranges::begin(E)` is ill-formed with no diagnostic required.

(2.3) — ~~`t + 0` if `t` is of~~Otherwise, if `T` is an array type~~ ([basic.compound])~~, `ranges::begin(E)` is expression-equivalent to `t + 0`.

(2.4) — Otherwise, if *`decay-copy`*`(t.begin())` ~~if it~~ is a valid expression ~~and its~~whose type ~~`I`~~ models `input_or_output_iterator`, `ranges::begin(E)` is expression-equivalent to *`decay-copy`*`(t.begin())`.

(2.5) — Otherwise, if `T` is a class or enumeration type and *`decay-copy`*`(begin(t))` ~~if it~~ is a valid expression ~~and its~~whose type ~~`I`~~ models `input_or_output_iterator` with overload resolution performed in a context ~~that includes~~in which unqualified lookup for `begin` finds only the declarations~~:~~

```
~~template<class T> void begin(T&&) = delete;~~
~~template<class T> void begin(initializer_list<T>&&) = delete;~~
void begin(auto&) = delete;
void begin(const auto&) = delete;
```

~~and does not include a declaration of `ranges::begin`~~ then `ranges::begin(E)` is expression-equivalent to *`decay-copy`*`(begin(t))` with overload resolution performed in the above context.

(2.6) — Otherwise, `ranges::begin(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::begin(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::begin(E)` is a valid expression, its type models `input_or_output_iterator`. — *end note*]

24.3.2 `ranges::end` [range.access.end]

1 The name `ranges::end` denotes a customization point object ([customization.point.object]).

2 Given a subexpression `E` ~~and an lvalue~~with type `T`, let `t` be an lvalue that denotes the same object as `E`~~,~~ if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — ~~if~~If `E` is an rvalue and `enable_safe_range<`~~`remove_cvref_t<decltype((E))>`~~`remove_cv_t<T>>` is `false`, `ranges::end(E)` is ill-formed. ~~Otherwise, `ranges::end(E)` is expression-equivalent to:~~

(2.2) — Otherwise, if `T` is an array type ([basic.compound]) and `remove_all_extents_t<T>` is an incomplete type, `ranges::end(E)` is ill-formed with no diagnostic required.

(2.3) — Otherwise, if `T` is an array of unknown bound, `ranges::end(E)` is ill-formed.

(2.4) — Otherwise, if `T` is an array, `ranges::end(E)` is expression-equivalent to `t + extent_v<T>` ~~if `E` is of array type ([basic.compound]) `T`~~.

(2.5) — Otherwise, if *`decay-copy`*`(t.end())` ~~if it~~ is a valid expression ~~and its~~whose type ~~`S`~~ models

`sentinel_for<`~~`decltype(ranges::begin(E))`~~`iterator_t<T>>`

then `ranges::end(E)` is expression-equivalent to *`decay-copy`*`(t.end())`.

(2.6) — Otherwise, ~~*`decay-copy`*`(end(t))`~~ if ~~it~~ `T` is a class or enumeration type and *`decay-copy`*`(end(t))` is a valid expression ~~and its~~whose type ~~`S`~~ models

`sentinel_for<`~~`decltype(ranges::begin(E))`~~`iterator_t<T>>`

with overload resolution performed in a context ~~that includes~~in which unqualified lookup for `end` finds only the declarations~~:~~

~~`template<class T> void end(T&&) = delete;`~~
~~`template<class T> void end(initializer_list<T>&&) = delete;`~~
`void end(auto&) = delete;`
`void end(const auto&) = delete;`

~~and does not include a declaration of `ranges::end`~~ then `ranges::end(E)` is expression-equivalent to *`decay-copy`*`(end(t))` with overload resolution performed in the above context.

(2.7) — Otherwise, `ranges::end(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::end(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::end(E)` is a valid expression, the types `S` and `I` of `ranges::end(E)` and `ranges::begin(E)` model `sentinel_for<S, I>`. — *end note*]

[...]

24.3.5 `ranges::rbegin` [range.access.rbegin]

1 The name `ranges::rbegin` denotes a customization point object ([customization.point.object]).

2 Given a subexpression `E` ~~and an lvalue~~with type `T`, let `t` be an lvalue that denotes the same object as `E`~~,~~ if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — ~~if~~If `E` is an rvalue and `enable_safe_range<`~~`remove_cvref_t<decltype((E))>`~~`remove_cv_t<T>>` is `false`, `ranges::rbegin(E)` is ill-formed. ~~Otherwise, `ranges::rbegin(E)` is expression-equivalent to:~~

(2.2) — Otherwise, if `T` is an array type ([basic.compound]) and `remove_all_extents_t<T>` is an incomplete type, `ranges::rbegin(E)` is ill-formed with no diagnostic required.

(2.3) — Otherwise, if *`decay-copy`*`(t.rbegin())` ~~if it~~ is a valid expression ~~and its~~whose type ~~`I`~~ models `input_or_output_iterator`, `ranges::rbegin(E)` is expression-equivalent to *`decay-copy`*`(t.rbegin())`.

(2.4) — Otherwise, ~~*`decay-copy`*`(rbegin(t))`~~ if ~~it~~ `T` is a class or enumeration type and *`decay-copy`*`(rbegin(t))` is a valid expression ~~and its~~whose type ~~`I`~~ models `input_or_output_iterator` with overload resolution performed in a context ~~that includes~~in which unqualified lookup for `rbegin` finds only the declaration~~:~~s

~~`template<class T> void rbegin(T&&) = delete;`~~
`void rbegin(auto&) = delete;`
`void rbegin(const auto&) = delete;`

~~and does not include a declaration of `ranges::rbegin`~~ then `ranges::rbegin(E)` is expression-equivalent to *`decay-copy`*`(rbegin(t))` with overload resolution performed in the above context.

(2.5) — Otherwise, ~~`make_reverse_iterator(ranges::end(t))`~~ if both `ranges::begin(t)` and `ranges::end(t)` are valid expressions of the same type ~~`I`~~ which models `bidirectional_iterator` ([iterator.concept.bidir]), `ranges::rbegin(E)` is expression-equivalent to `make_reverse_iterator(ranges::end(t))`.

(2.6) — Otherwise, `ranges::rbegin(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::rbegin(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::rbegin(E)` is a valid expression, its type models `input_or_output_iterator`. — *end note*]

24.3.6 `ranges::rend` [range.access.rend]

1 The name `ranges::rend` denotes a customization point object ([customization.point.object]).

2 Given a subexpression `E` ~~and an lvalue~~with type `T`, let `t` be an lvalue that denotes the same object as `E`~~,~~ if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — ~~if~~If `E` is an rvalue and `enable_safe_range<`~~`remove_cvref_t<decltype((E))>`~~`remove_cv_t<T>>` is `false`, `ranges::rend(E)` is ill-formed. ~~Otherwise, `ranges::rend(E)` is expression-equivalent to:~~

(2.2) — Otherwise, if `T` is an array type ([basic.compound]) and `remove_all_extents_t<T>` is an incomplete type, `ranges::rend(E)` is ill-formed with no diagnostic required.

(2.3) — Otherwise, if *`decay-copy`*`(t.rend())` ~~if it~~ is a valid expression ~~and its~~whose type ~~`S`~~ models

```
sentinel_for<decltype(ranges::rbegin(E))>
```

then `ranges::rend(E)` is expression-equivalent to *`decay-copy`*`(t.rend())`.

(2.4) — Otherwise, ~~*`decay-copy`*`(rend(t))`~~ if ~~it~~ `T` is a class or enumeration type and *`decay-copy`*`(rend(t))` is a valid expression ~~and its~~whose type ~~`S`~~ models

```
sentinel_for<decltype(ranges::rbegin(E))>
```

with overload resolution performed in a context ~~that includes~~in which unqualified lookup for `rend` finds only the declaration~~:~~s

```
~~template<class T> void rend(T&&) = delete;~~
void rend(auto&) = delete;
void rend(const auto&) = delete;
```

~~and does not include a declaration of `ranges::rend`~~ then `ranges::rbegin(E)` is expression-equivalent to *`decay-copy`*`(rend(t))` with overload resolution performed in the above context.

(2.5) — Otherwise, ~~`make_reverse_iterator(ranges::begin(t))`~~ if both `ranges::begin(t)` and `ranges::end(t)` are valid expressions of the same type ~~`I`~~ which models `bidirectional_iterator` ([iterator.concept.bidir]), then `ranges::rend(E)` is expression-equivalent to `make_reverse_iterator(ranges::begin(t))`.

(2.6) — Otherwise, `ranges::rend(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::rend(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::rend(E)` is a valid expression, the types `S` and `I` of `ranges::rend(E)` and `ranges::rbegin(E)` model `sentinel_for<S, I>`. — *end note*]

[...]

24.3.9 `ranges::size` [range.prim.size]

1 The name `ranges::size` denotes a customization point object ([customization.point.object]).

2 ~~The expression `ranges::size(E)` for some~~Given a subexpression `E` with type `T` ~~is expression-equivalent to:~~, let `t` be an lvalue that denotes the same object as `E` if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — If `T` is an array of unknown bound ([dcl.array]), `ranges::size(E)` is ill-formed.

(2.2) — ~~*`decay-copy`*`(extent_v<T>)`~~ Otherwise, if `T` is an array type ~~([basic.compound])~~, `ranges::size(E)` is expression-equivalent to *`decay-copy`*`(extent_v<T>)`.

(2.3) — ~~Otherwise, if `disable_sized_range<remove_cv_t<T>>` ([range.sized]) is `false`:~~

(2.4) — Otherwise, ~~*`decay-copy`*`(E.size())`~~ if ~~it~~ `disable_sized_range<remove_cv_t<T>>` ([range.sized]) is `false` and *`decay-copy`*`(t.size())` is a valid expression ~~and its~~of integer-like type ~~`I` is integer-like~~ ([iterator.concept.winc]), `ranges::size(E)` is expression-equivalent to *`decay-copy`*`(E.size())`.

(2.5) — Otherwise, ~~*`decay-copy`*`(size(E))`~~ if ~~it~~ `T` is a class or enumeration type, `disable_sized_range<remove_cv_t<T>>` is `false`, and *`decay-copy`*`(size(t))` is a valid expression ~~and its~~of integer-like type ~~`I` is integer-like~~

with overload resolution performed in a context ~~that includes~~in which unqualified lookup for `size` finds only the declaration~~:~~s

```
~~template<class T> void size(T&&) = delete;~~
void size(auto&) = delete;
void size(const auto&) = delete;
```

~~and does not include a declaration of `ranges::size`~~ then `ranges::size(E)` is expression-equivalent to *`decay-copy`*`(size(E))` with overload resolution performed in the above context.

(2.6) — Otherwise, if *`make-unsigned-like`*`(ranges::end(`~~`E`~~`t) - ranges::begin(`~~`E`~~`t))` (~~[range.subrange]~~[range.syn]) ~~if it~~ is a valid expression and the types `I` and `S` of `ranges::begin(`~~`E`~~`t)` and `ranges::end(`~~`E`~~`t)` (respectively) model both `sized_sentinel_for<S, I>` ([iterator.concept.sizedsentinel]) and `forward_iterator<I>`, then `ranges::size(E)` is expression-equivalent to *`make-unsigned-like`*`(ranges::end(t) - ranges::begin(t))`. ~~However, `E` is evaluated only once.~~

(2.7) — Otherwise, `ranges::size(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::size(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::size(E)` is a valid expression, its type is integer-like. — *end note*]

24.3.10 `ranges::empty` [range.prim.empty]

1 The name `ranges::empty` denotes a customization point object ([customization.point.object]).

2 ~~The expression `ranges::empty(E)` for some~~Given a subexpression `E` ~~is expression-equivalent to:~~with type `T`, let `t` be an lvalue that denotes the same object as `E` if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — If `T` is an array of unknown bound ([basic.compound]), `ranges::empty(E)` is ill-formed.

(2.2) — Otherwise, if `bool(`~~`(E)`~~`t.empty())` ~~if it~~ is a valid expression, `ranges::empty(E)` is expression-equivalent to `bool(t.empty())`.

(2.3) — Otherwise, if `(ranges::size(`~~`E`~~`t) == 0)` ~~if it~~ is a valid expression, `ranges::empty(E)` is expression-equivalent to `(ranges::size(t) == 0)`.

(2.4) — Otherwise, ~~`EQ`, where `EQ` is~~if `bool(ranges::begin(`~~`E`~~`t) == ranges::end(`~~`E`~~`t))` ~~except that `E` is only evaluated once, if `EQ`~~ is a valid expression and the type of `ranges::begin(`~~`E`~~`t)` models `forward_iterator`, `ranges::empty(E)` is expression-equivalent to `bool(ranges::begin(t) == ranges::end(t))`.

(2.5) — Otherwise, `ranges::empty(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::empty(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::empty(E)` is a valid expression, it has type `bool`. — *end note*]

24.3.11 `ranges::data` [range.prim.data]

1 The name `ranges::data` denotes a customization point object ([customization.point.object]).

2 ~~The expression `ranges::data(E)` for some~~Given a subexpression `E` ~~is expression-equivalent to:~~with type `T`, let `t` be an lvalue that denotes the same object as `E` if `E` is a glvalue and otherwise denotes the result of applying the temporary materialization conversion ([conv.rval]) to `E`. Then:

(2.1) — If `E` is an rvalue and `enable_safe_range<remove_cv_t<T>>` is `false`, `ranges::data(E)` is ill-formed.

(2.2) — Otherwise, if `T` is an array type ([basic.compound]) and `remove_all_extents_t<T>` is an incomplete type, `ranges::data(E)` is ill-formed with no diagnostic required.

(2.3) — ~~If `E` is an lvalue,~~Otherwise, if *`decay-copy`*`(`~~`E`~~`t.data())` ~~if it~~ is a valid expression of pointer to object type, `ranges::data(E)` is expression-equivalent to *`decay-copy`*`(t.data())`.

(2.4) — Otherwise, if `ranges::begin(`~~`E`~~`t)` is a valid expression whose type models `contiguous_iterator`, `ranges::data(E)` is expresssion-equivalent to `to_address(ranges::begin(E))`.

(2.5) — Otherwise, `ranges::data(E)` is ill-formed.

3 [*Note*: ~~This case can~~Diagnosable ill-formed cases above result in substitution failure when `ranges::data(E)` appears in the immediate context of a template instantiation. — *end note*]

4 [*Note*: Whenever `ranges::data(E)` is a valid expression, it has pointer to object type. — *end note*]

Bibliography

[1] Great Britain. GB046: Allow caching of evaluations of concept specializations. `https://github.com/cplusplus/nbballot/issues/45`. Accessed: 2020-01-12.

[2] Great Britain. GB275: `ranges::begin/end` should not accept arrays of unknown bound. `https://github.com/cplusplus/nbballot/issues/271`. Accessed: 2020-01-10.

[3] Casey Carter. P0944R0: Contiguous ranges, February 2018. `https://wg21.link/p0944r0`.

[4] LWG. Issue 3258: Range access and `initializer_list`. `https://wg21.link/lwg3258`. Accessed: 2020-01-10.

[5] LWG. Issue 3299: Pointers don't need customized iterator behavior. `https://wg21.link/lwg3299`. Accessed: 2020-01-10.

[6] LWG. Issue 3368: Exactly when does `size` return `end - begin`? `https://wg21.link/lwg3368`. Accessed: 2020-01-19.

[7] Eric Niebler. P0970R1: Better, safer range access customization points, May 2018. `https://wg21.link/p0970r1`.

[8] Barry Revzin. P1870R1: *`forwarding-range`*`<t>` is too subtle, November 2019. `https://wg21.link/p1870r1`.