Further relaxing the range-for loop customization point finding rules

PXXXXR0

Document #:	PXXXXR0	**Date:**	2026-08-10
Project:	Programming Language C++	**Audience:**	EWG
Author: Connor Song

1. Abstract

> Customization protocols without an explicit opt-in mechanism necessarily rely, to some extent, on heuristics for programmer intent. A language rule must draw a boundary between "the programmer probably intended to provide this customization" and "an operation is actually applicable in this context."
> For range-based `for`, the current language draws that boundary at the mere presence of member declarations named `begin` and `end`: if class-scope searches for both names find declarations, the language commits to the member interpretation, regardless of whether those declarations can participate in the required calls.
> This paper argues that this heuristic is too strong. We propose moving the boundary from **declaration presence** to **applicability**, represented by candidate viability when overload resolution applies.

The change addresses two practical problems:

1. the discrepancy, identified by LWG 4389, between range-based `for` and the Ranges customization point objects when member declarations exist but the corresponding member calls are not applicable; and
2. the inability to non-intrusively adapt a type to the range protocol when it already contains unrelated members named `begin` and `end`. [LWG4389]

The proposal preserves the existing rule that `begin` and `end` are selected together from the same customization mechanism. It does not propose independently selecting a member `begin` and an ADL-found `end`, or vice versa.

2. Motivation

2.1 Non-intrusive adaptation of legacy types

Consider a container from a legacy library, perhaps designed before range-based `for` existed:

```
namespace legacy {

template<class T>
struct node;

template<class T>
struct linked_list {
    node<T>* begin;   // first node
    node<T>* end;     // sentinel / past-the-end node

    // legacy interface...
};

template<class T>
struct iterator;

// Added later as a non-intrusive range adaptation.
template<class T>
iterator<T> begin(linked_list<T>&);

template<class T>
iterator<T> end(linked_list<T>&);
```

```
}
```

There is no reason to interpret the data members `linked_list<T>::begin` and `linked_list<T>::end` as an attempted member-based range customization. Their names simply describe the representation of the legacy container.

Nevertheless:

```
legacy::linked_list<int> list;

for (auto&& x : list) {
    // ...
}
```

is ill-formed under the current language rules.

The searches in the scope of `linked_list<int>` find declarations named both `begin` and `end`. `[stmt.ranged]` therefore commits to the member interpretation and forms:

```
list.begin();
list.end();
```

The two data members are not callable, but the ADL customization is never considered. The current normative rule explicitly requires only that both class-scope searches find at least one declaration.

There is no malformed member range customization to diagnose here. The member declarations are unrelated to the protocol, while the free functions are the intentional non-intrusive adaptation.

P2279R0 identifies non-intrusive opt-in as an important property of customization mechanisms and specifically observes that ADL-based free-function customization provides it. It also observes that ADL does not provide a genuine explicit marker of the interface into which a declaration intends to opt. [P2279R0]

2.2 Divergence from the Ranges customization points

The same rule creates an observable discrepancy with the Ranges customization point objects.

Consider:

```
#include <algorithm>
#include <ranges>

namespace N {

struct R {
    int data[1] = {42};

    int* begin(int);
    int* end(int);

    friend int* begin(R& r)
    {
        return r.data;
    }

    friend int* end(R& r)
    {
        return r.data + 1;
    }
};

}

static_assert(std::ranges::range<N::R>);

void f(N::R& r)
{
    std::ranges::for_each(r, [](int) { }); // OK

    for (int x : r) { }                    // ill-formed
```

```
}
```

The Ranges customization point objects can proceed to ADL when the member form is not valid, so `N::R` satisfies `std::ranges::range` and can be consumed by `std::ranges::for_each`. Range-based `for`, however, finds both member names and commits to `r.begin()` and `r.end()` before discovering that neither call has a viable candidate. LWG 4389 identifies exactly this discrepancy. [LWG4389]

LWG 4389 also identifies another difference: `ranges::begin` and `ranges::end` are selected independently, so one may use a member customization while the other uses ADL. That is a separate question.

This paper does not address the "together versus separate" difference. Range-based `for` continues to select `begin` and `end` through the same mechanism.

3. Proposal

We propose that range-based `for` commit to the member interpretation only when both member call expressions are applicable.

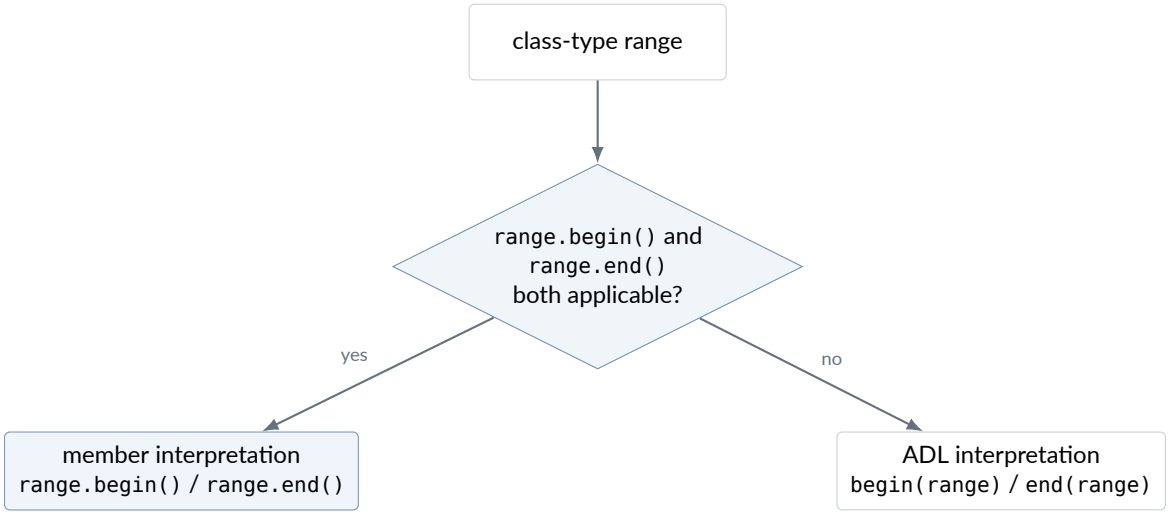

Figure 1: Proposed member-vs-ADL selection rule for a class-type range.

For the purposes of this proposal, applicability is determined as follows:

- where overload resolution applies to the member call, overload resolution must find at least one viable candidate;
- where overload resolution does not apply, the call itself must be a valid expression.

The proposal deliberately does **not** require overload resolution to succeed. The existence of a viable candidate is sufficient to commit to the member protocol. Thus, an ambiguous member overload set remains a member customization and is diagnosed rather than causing fallback to ADL.

Likewise, a viable function that is subsequently rejected because it is deleted or inaccessible continues to lock in the member interpretation. The proposal changes the point at which a member mechanism is considered applicable; it does not turn later invocation errors into fallback.

The proposal also preserves the existing atomic selection of the two bounds: if the member pair does not meet the criterion, the existing ADL pair is considered.

4. Relation to P0962R1

P0962R1 previously relaxed exactly the same range-based `for` customization rule.

Before P0962R1, finding either `begin` or `end` was sufficient to commit to the member interpretation. P0962R1 changed the rule to require both names, allowing ADL customization of types that merely happened to contain one unrelated member name. It explicitly analyzed the old behavior in terms of an assumption about programmer intent:

the rule assumed that the programmer had intended to provide member-based range traversal and had implemented it incorrectly. P0962R1 argued that such an assumption should not prevent otherwise useful customization techniques. [P0962R1]

The evolution of the rule can therefore be summarized as:

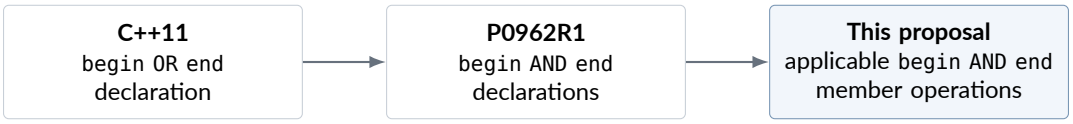

Figure 2: Successive relaxation of the range-for member lock-in heuristic.

P0962R1 established that the presence of a relevant-looking member name is not, by itself, sufficient evidence that the type author intended to advertise the member range protocol.

This paper asks the natural follow-up question:

> *Why should two declarations that cannot participate in the required calls constitute sufficient evidence?*

P0962R1 was adopted as a Defect Report and applied to the working paper by unanimous consent.

5. Design considerations

5.1 Intent, opt-in, and diagnostics

The current declaration-based lock-in has an intentional benefit.

Consider:

```
namespace N {

struct R {
    iterator begin();       // perhaps accidentally missing const
    sentinel end();
};

iterator begin(R const&);
sentinel end(R const&);

}

void f(N::R const& r)
{
    for (auto&& e : r) {
        // ...
    }
}
```

Under the current rule, the existence of both member names commits the range-based `for` to the member interpretation. The calls on `const R` then fail, potentially revealing that the member functions were accidentally declared without `const`.

Under this proposal, those member functions are not viable for a `const R` object. The non-member ADL customization can therefore be selected instead.

That is a real trade-off. CWG has explicitly described the current member-name lock-in as intentional, including the desire to diagnose cases such as an accidentally missing `const` rather than silently falling back. [CWG3123]

The language, however, cannot determine from the declarations alone whether the missing `const` was accidental. The non-member overloads in the example may themselves be the intentionally supplied customization.

This is ultimately a question about inferred intent.

P2279R0 separates several desirable properties of a customization mechanism, notably **explicit opt-in**, **diagnosing**

incorrect opt-in, **atomic grouping**, and **non-intrusive customization**. For pure ADL customization, P2279 identifies non-intrusiveness as a strength, while noting that opt-in is implicit rather than explicitly associated with an interface and that incorrect opt-in cannot in general be reliably diagnosed. [P2279R0]

The current range-based `for` rule attempts to recover some ability to diagnose an incorrect opt-in by treating the mere presence of two member names as evidence that a member customization was intended. But this is only a heuristic: the legacy `linked_list` example contains exactly those two names without expressing any such intent.

C++ also routinely distinguishes the existence of a declaration from its applicability. Substitution failure can make deduction fail rather than immediately making the program ill-formed, and an invalid substitution in the immediate context of an atomic constraint causes that constraint not to be satisfied. Concepts and requires-based programming build extensively on the ability to ask whether an operation applies in a particular context.

These mechanisms are not direct precedents for range-based `for`. They do, however, demonstrate that non-applicability is not generally treated by C++ as proof that the programmer made an error that must be diagnosed immediately.

The proposal therefore does not claim that fallback can never hide a mistake. It changes where the language draws the trade-off:

> from **“two declarations with these names exist”**
> to **“two applicable member operations exist.”**

5.2 Why this wording?

There are two separate reasons for the proposed wording.

5.2.1 Why are there two cases?

It would be tempting to specify the rule solely in terms of overload resolution finding a viable candidate. That would accidentally reject existing valid member call forms for which overload resolution is not performed.

For example:

```
using iter_fn = iterator (*)();
using sent_fn = sentinel (*)();

struct R {
    iter_fn begin;
    sent_fn end;
};
```

Assuming the returned iterator and sentinel types satisfy the remaining range requirements,

```
R r;
r.begin();
r.end();
```

are valid calls through function-pointer data members.

No overloaded member function is being selected. `[over.match.call.general]` applies overload resolution when the postfix-expression names functions or function templates, or when it denotes an object of class type; a direct call through a function pointer does not fall into those overload-resolution cases.

The wording must therefore distinguish:

```
overload resolution applies        → ask whether it finds a viable candidate
overload resolution does not apply → ask whether the expression is valid
```

This preserves callable non-function members while still allowing unrelated non-callable data members such as the legacy `node<T>* begin` and `node<T>* end` example to fall through to ADL.

5.2.2 Why viable candidates rather than successful overload resolution?

Candidate viability is the point at which overload processing has established that a candidate can actually participate in the call. Only after the viable set is formed is the best viable function selected; overload resolution succeeds only when a unique best viable function exists.

Consider:

```
namespace N {

struct R {
    iterator begin(int = 0);
    iterator begin(double = 0);

    sentinel end();

    friend iterator begin(R&);
    friend sentinel end(R&);
};

}
```

`r.begin()` has viable member candidates, but selecting between them is ambiguous.

This proposal intentionally continues to commit to the member interpretation. The ambiguity is then diagnosed exactly as an error in the chosen member protocol. The existence of an ADL customization does not rescue the program.

Requiring **successful overload resolution** instead would be a substantially more aggressive fallback rule: ambiguity would cause ADL to be tried. Requiring the complete member expression to be valid in all cases would go further still and could also turn other post-viability errors into fallback.

Viability therefore marks a useful middle boundary:

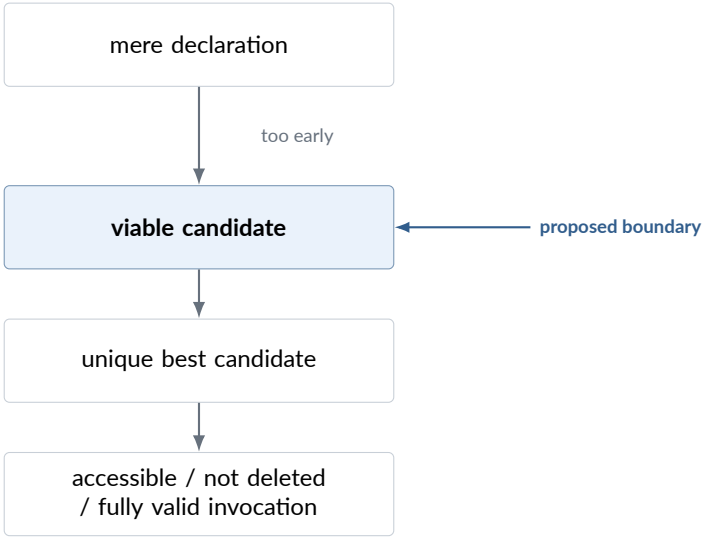

Figure 3: The proposed boundary sits after candidate viability, but before best-candidate selection and later invocation checks.

There is a closely related recent precedent in `[stmt.expand]`. Following CWG 3123 and EWG consideration, the non-member expansion-iterable rule now requires ADL for `begin(E)` and `end(E)` to each find at least one **viable candidate**, rather than merely finding functions or function templates. The official Core issue records that EWG approved the viable-candidate approach after considering alternatives and its potential instantiation costs. [CWG3123]

That decision concerned the ADL branch of expansion statements, not member lock-in, so it is not a direct precedent for this proposal. It does demonstrate that **candidate viability is already an accepted applicability boundary in closely related range machinery**.

6. Impact on existing code

At the level of range-protocol selection, the intended effect of this proposal is a relaxation.

An existing well-formed range-based `for` that uses ordinary member `begin()` and `end()` functions necessarily has applicable member calls and continues to select the member interpretation. Existing callable non-function members are preserved by the second branch of the proposed criterion.

The intended behavioral change occurs where the current rule finds both names but at least one corresponding member operation is not applicable. Such programs are currently ill-formed after committing to the member interpretation; this proposal allows the existing ADL interpretation to be considered instead.

Table 1: Range-protocol selection: current vs. proposed rule.

Case	Current rule	Proposed rule
viable member `begin` and `end`	member	member
ambiguous but viable member overloads	diagnostic	diagnostic
selected deleted/inaccessible member	diagnostic	diagnostic
unrelated non-callable data members + ADL	diagnostic	**ADL**
wrong-arity members + ADL	diagnostic	**ADL**
non-`const` members on a `const` range + ADL	diagnostic	**ADL**
callable function-pointer data members	member	member

The proposal is therefore not intended to change the meaning of existing well-formed range-based `for` statements. Its principal semantic effect is to make some currently ill-formed programs well-formed by permitting an intentional ADL customization to participate.

There is nevertheless a compile-time compatibility consideration. Determining candidate viability can require template deduction, constraint checking, conversion analysis, and potentially additional template instantiation. CWG 3123 identified the same concern when viability was introduced into `[stmt.expand]`, including the possibility of failures outside an immediate context and non-local effects observable through reflection; EWG nevertheless selected the viable-candidate approach there. [CWG3123]

This issue deserves implementation experience, but it is distinct from the source-level semantic compatibility of the range protocol itself.

6.1 Application as a Defect Report

P0962R1 modified the same range-based `for` customization rule and was adopted as a Defect Report. Its principal motivation was that accidental member-name lock-in prevented otherwise valid ADL customization. [P0962R1]

This proposal addresses a remaining instance of the same class of problem: two unrelated or non-applicable declarations can still suppress an intentional ADL customization.

The author therefore recommends that, if EWG accepts the proposed design, the change be considered for application as a Defect Report to C++11 and later.

7. Proposed wording

The wording below is preliminary and is expected to receive Core review.

The wording is relative to the current working draft. The current `[stmt.ranged]` member-selection rule requires class-scope searches for `begin` and `end` to each find at least one declaration.

Change `[stmt.ranged]` paragraph 1.3.2 as follows:

if the type of `range` is a reference to a class type `C` and, ~~searches in the scope of `C` ([class.member.lookup]) for the names `begin` and `end` each find at least one declaration~~ **for each expression `E` in the set comprising `range.begin()` and `range.end()`, either**

- **overload resolution ([over.match]) is applied to `E` and finds at least one viable candidate ([over.match.viable]), or**
- **overload resolution is not applied to `E` and `E` is a valid expression,**

`begin-expr` and `end-expr` are `range.begin()` and `range.end()`, respectively;

The existing ADL alternative in the following bullet is unchanged.

8. References

[P0962R1] Ville Voutilainen, *Relaxing the range-for loop customization point finding rules*, 2018. The paper changed the member lock-in condition from either name being found to both names being found and proposed the change as a Defect Report.
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0962r1.html

[P2279R0] Barry Revzin, *We need a language mechanism for customization points*, 2021. The paper discusses explicit opt-in, diagnosis of incorrect opt-in, atomic grouping, and non-intrusive customization, and identifies ADL-based customization as non-intrusive but only implicitly opted into.
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2279r0.html

[LWG4389] *ranges::for_each possibly behaves differently from range-based for*. The issue identifies both the independent-selection difference and the invalid-member-call fallback difference between the Ranges CPOs and core-language range-based `for`.
https://cplusplus.github.io/LWG/issue4389

[CWG3123] *Global lookup for begin and end for expansion statements*. The accepted resolution uses the "at least one viable candidate" criterion for the ADL case of expansion statements; EWG approved that approach in March 2026.
https://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html

[Working Draft] C++ Working Draft — `[stmt.ranged]`, `[stmt.expand]`, `[over.match]`, `[over.match.call]`. The current draft retains declaration-based member lock-in for range-based `for`, while the ADL case of expansion statements uses viable candidates.