Automatically Generate operator->

Document #: P3039R1
Date: 2026-01-12
Project: Programming Language C++
Audience: Evolution Working Group
Library Evolution Working Group
Reply-to: David Stone
<davidfromonline@gmail.com>
Andre Kostur
<andre@kostur.net>

Contents

1 Summary

This proposal follows the lead of `operator<=>` (see Consistent Comparison [P0515R3]) by generating rewrite rules if a particular operator does not exist in current code. This is a follow-up to Automatically Generate More Operators [P1046R2], but includes only `operator->` and `operator->*`.

— Rewrite `lhs->rhs` as `(*lhs).rhs`
— Rewrite `lhs->*rhs` as `(*lhs).*rhs`

2 Revision History

2.1 R1 January 2026

— Paper adopted by Andre Kostur andre@kostur.net
— Added a second motivation around applying `operator->` to an rvalue
— Added proposed wording
— Rephrasing from "generating" an `operator->` to describing a rewriting behaviour

2.2 R0 November 2023

— Initial publication, written by David Stone davidfromonline@gmail.com

3 Design Motivations

The first motivation of this paper is that users should have little or no reason to write their own version of `operator->` when they have already provided an appropriate `operator*`. It would be a strong indictment if there are many examples of well-written code for which this paper provides no simplification, so a fair amount of time in this paper will be spent looking into the edge cases and making sure that we behave correctly. At the very least, types that would not have the correct behaviour generated should not generate an operator at all. In other words, it should be uncommon for users to define their own versions (the default should be good enough most of the time), and it should be extremely rare that users want to prevent `operator->` from being available (`= delete` should almost never appear).

This paper was split off from [P1046R2]. That paper proposed generating many operators, this paper is just the arrow overloads. This is the part of that paper with the most motivation (as users cannot write the equivalent), the feature that gained the most support from the previous EWG meeting, and one of the least complicated changes.

This area has been well-studied for library solutions. This paper, however, traffics in rewrite rules (following the lead of `operator<=>`), not in terms of function calls. Because of this, we have one more option that the library-only solutions lack: we define `lhs->rhs` as being equivalent to `(*lhs).rhs`. This neatly sidesteps all of the issues of library-only solutions (how do we get the address of the object? how do we handle temporaries?). It even plays nicely with existing rules around lifetime extension of temporaries. This solves many long-standing issues around proxy iterators that return by value.

The second motivation is that when one applies `operator->` to an rvalue, at some point during the evaluation invoking `operator->` will return a raw pointer. Since one cannot have a pointer-to-reference, the rvalueness is lost: which means that one would not be able to invoke an rvalue-qualified member function on that final type. Which includes a potentially deleted rvalue-qualified member function. If the expression is rewritten to use `operator*`, that operator can be reference-qualified to retain the rvalueness through to the end of the expression.

4 Comparison with comparison rewrites

This change tries to follow the lead of the existing rewrite rules we have. However, unlike the comparison operators this change is significantly simpler and less likely to have all the corner cases we've seen with `operator==` and `operator<=>`. Much of the complexity around the comparison operators has come from reversed candidates and dealing with potentially multiple arguments. The specification of these operators should be significantly simpler.

4.1 operator->

`operator->` is the simplest. It cannot be defined outside of the class, so there is exactly one place to look for whether it exists today. It is also a unary operator (it does not depend on the right-hand side). This makes overload resolution and name lookup very simple.

4.2 operator->*

`operator->*` is slightly more complicated. It can be defined as a free operator, a member function, and a hidden friend. However, we've already solved the problems associated with that for comparison operators. It is a binary operator, but the order is meaningful and thus there are no complexities around reversed candidates. One of the base operators for this, `operator.*`, also cannot be overloaded. This means that the only overloaded operator that is relevant to `operator->*` is the same as for `operator->`: `operator*`, which is a unary operator. That fact also eliminates much of the complexity of `operator==` vs. `operator!=`.

5 Design

If any of this conflicts with how `operator!=` is defined in terms of `operator==` and is not explicitly called out as a difference, that difference is unintentional.

All of these examples assume there is a variable lhs of type LHS.

5.1 operator->

If the expression `lhs->rhs` is encountered:

— If overload resolution for `operator->` would succeed, then that operator is called. This can also select a deleted overload.
— If overload resolution does not succeed, but the expression `*lhs` is well-formed, then the expression is rewritten to `(*lhs).rhs`
— Otherwise, the expression is ill-formed

5.2 operator->*

If the expression `lhs->*rhs` is encountered:

— If overload resolution for `operator->*` would succeed, then that operator is called. This can also select a deleted overload.
— If overload resolution does not succeed, but the expression `*lhs` is well-formed, then the expression is rewritten to `(*lhs).*rhs`
— Otherwise, the expression is ill-formed

6 Library impact

Simply removing the `operator->` definition from classes could possibly cause a different member function to be invoked. Specifically if the `operator*` that is chosen returns an rvalue reference and we are invoking a member function that has been both lvalue and rvalue reference qualified: using `operator->` would cause the lvalue qualified form to be invoked but using `operator*` would cause the rvalue qualified form to be invoked. This may cause subtle behaviour changes. Because of this consideration, this paper will not be removing any existing declarations of `operator->` in the standard library types. We will leave that discussion to happen in a follow-up paper once this facility is made available in the language.

Even if we do not change the specification of the existing standard library types, there will still be new possibilities that are exposed. I have surveyed the standard library to get an overview of what would change in response to this, and to ensure that the changes would work properly. This covers every type that was in the standard library as of early 2020.

6.1 Types that will effectively gain `operator->` and this is a good thing

- `move_iterator` currently has a deprecated `operator->`
- `counted_iterator`
- `istreambuf_iterator`
- `istreambuf_iterator::proxy` (exposition only type)
- `iota_view::iterator`
- `transform_view::iterator`
- `split_view::outer_iterator`
- `split_view::inner_iterator`
- `basic_istream_view::iterator`
- `elements_view::iterator`

Most of these are iterators that return either by value or by `decltype(auto)` from some user-defined function. It is not possible to safely and consistently define `operator->` for these types, so we do not always do so, but under this proposal they would all do the right thing.

6.2 Types that will technically gain operator-> but it is not observable

- `insert_iterator`
- `back_insert_iterator`
- `front_insert_iterator`
- `ostream_iterator`

The insert iterators and `ostream_iterator` technically gain an `operator->`, but `operator*` returns a reference to *this and the only members of those types are types, constructors, and operators, none of which are accessible through `operator->` using the syntaxes that are supported to access the standard library.

6.3 Types that will gain `operator->` and it's weird either way

- `ostreambuf_iterator`

`ostreambuf_iterator` is the one example for which we might possibly want to explicitly delete `operator->`. It has an `operator*` that returns *this, and it has a member function `failed()`, so it would allow calling `it->failed()` with the same meaning as `it.failed()`.

6.4 Types that have `operator->` now and will need to be examined

All types in this section have an `operator->` that is identical to what the rewrite would accomplish, if we do not wish to support users calling with the syntax `thing.operator->()`.

- `optional`
- `unique_ptr` (single object)
- `shared_ptr`
- `weak_ptr`
- `basic_string::iterator`
- `basic_string_view::iterator`
- `array::iterator`
- `deque::iterator`
- `forward_list::iterator`
- `list::iterator`
- `vector::iterator`
- `map::iterator`
- `multimap::iterator`
- `set::iterator`
- `multiset::iterator`
- `unordered_map::iterator`

- `unordered_set::iterator`
- `unordered_multimap::iterator`
- `unordered_multiset::iterator`
- `span::iterator`
- `istream_iterator`
- `valarray::iterator`
- `tzdb_list::const_iterator`
- `filesystem::path::iterator`
- `directory_iterator`
- `recursive_directory_iterator`
- `match_results::iterator`
- `regex_iterator`
- `regex_token_iterator`
- `reverse_iterator`
- `common_iterator`
- `filter_view::iterator`
- `join_view::iterator`

All of these types that are adapter types define their `operator->` as deferring to the base iterator's `operator->`. However, the `Cpp17InputIterator` requirements specify that `a->m` is equivalent to `(*a).m`, so anything a user passes to `reverse_iterator` must already meet this. `common_iterator`, `filter_view::iterator`, and `join_view::iterator` were added in C++20 and require `input_or_output_iterator` of their parameter, which says nothing about `->`. Its `operator->` is defined as the first in a series that compiles:

1) Try calling member `operator->` on the base iterator
2) Try taking the address of the value returned from `operator*`
3) Create a proxy object that stores by-value returns and returns the address of that

If this paper were accepted, we have two options.

1) Get rid of the manual definition of `operator->` from those new types, which is a breaking change for iterator types with an `operator->` that does something meaningfully different from what their `operator*` does, or
2) Manually define it only when the wrapped type has a member `operator->`. This would keep step 1, but eliminate steps 2 and 3.

6.5 iterator_traits

`std::iterator_traits<I>::pointer` is essentially defined as `typename I::pointer` if such a type exists, otherwise `decltype(std::declval<I &>().operator->())` if that expression is well-formed, otherwise `void`. The type appears to be unspecified for iterators into any standard container, depending on how you read the requirements. The only relevant requirement on standard container iterators (anything that meets `Cpp17InputIterator`) are that `a->m` is equivalent to `(*a).m`. We never specify that any other form is supported, nor do we specify that any of them contain the member type pointer. There are three options here:

1) Change nothing. This would make pointer defined as void for types that have a rewritten `operator->`
2) Specify a further fallback of `decltype(std::addressof(*a))` to maintain current behavior and allow users to delete their own `operator->` without changing the results of `iterator_traits`
3) Deprecate or remove the `pointer` typedef, as it is not used anywhere in the standard except to define other `pointer` typedefs and it seems to have very little usefulness outside the standard.

My recommendation is 2, 3, or both.

6.6 `to_address` and `pointer_traits`

20.2.4 [pointer.conversion] specifies to_address in terms of calling `p.operator->()`, so some thought will need to be put in there on what to do.

The following standard types can be used to instantiate `pointer_traits`:

- `T *`
- `unique_ptr`
- `shared_ptr`
- `weak_ptr`
- `span`

However, none of them are specified to have member `to_address`.

Note that span does not have `operator->` and is thus not relevant to the below discussion at all. `unique_ptr`, `shared_ptr`, and `weak_ptr` are not iterators, and are thus minimally relevant to the below discussion.

`std::to_address` is specified as calling `pointer_traits<Ptr>::to_address(p)` if that is well formed, otherwise calling `operator->` with member function syntax. This leaves us with several options:

1) Leave this function as-is and specify that all of the types that currently have `operator->` have a specialization of `pointer_traits` that defines `pointer_traits<T>::to_address`
2) Specify that all types that currently have `operator->`work with `std::to_address`
3) Define a second fallback if `p.operator->()` is not valid that would be defined as `std::addressof(*p)`. This is similar to the question for `std::iterator_traits<I>::pointer`.

1 and 2 feel like the wrong approach – they would mean that authors of iterator types still need to define their own `operator->`, or they must specialize some class template (if we agree that the current semantics with regard to iterators are correct), or they must overload `to_address` and we make that a customization point found by ADL.

7 Proposed Wording

Add a second paragraph to 12.4.6 [over.ref]:

> If a class member access operator function is not selected by overload resolution, proceed as described in 7.6.1.5 [expr.ref]/2. That is: For an expression of the form
>
> *postfix-expression* `->` `template`$_{opt}$ *id-expression*
>
> the unary operator * function is selected by overload resolution (12.2.2.3 [over.match.oper]), and the expression is interpreted as
>
> `(` *postfix-expression* `. operator * () ) . template`$_{opt}$ *id-expression*
>
> Analogously, for an expression of the form
>
> *postfix-expression* `->` *splice-expression*
>
> the operator function is selected by overload resolution, and the expression is interpreted as
>
> `(` *postfix-expression* `. operator * () ) .` *splice-expression*

8 Feature Test Macro

```
#define __cpp_rewrite_arrow xxxxxxL
```

9 Acknowledgements

R0 was written by David Stone davidfromonline@gmail.com

10 References

[P0515R3] Herb Sutter, Jens Maurer, Walter E. Brown. 2017-11-10. Consistent comparison.
https://wg21.link/p0515r3

[P1046R2] David Stone. 2020-01-11. Automatically Generate More Operators.
https://wg21.link/p1046r2