Against implicit conversions for `indirect`

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3902R0

Working Group: Library Evolution

Date: 2025-10-29

Jonathan Coe <jonathanbcoe@gmail.com>

Antony Peacock <ant.peacock@gmail.com>

Sean Parent <sparent@adobe.com>

Abstract

The National Body Comment US 77-140 says:

indirect should convert to T& to simplify the use cases (e.g., returning the object from a function with a return type T&) where indirect appears as a drop-in replacement for T when T may be an incomplete type conditionally. With the proposed change, indirect is closer to reference_wrapper, but carries storage.

The authors of indirect are opposed to this change in the absence of significant implementation experience.

Discussion

Background

The class template `indirect` confers value-like semantics on a dynamically-allocated object. An `indirect` may hold an object of a class `T`. Copying the `indirect` will copy the object `T`. When an `indirect<T>` is accessed through a const access path, constness will propagate to the owned object.

`indirect<T>` can be implemented, like `reference_wrapper`, as a class with a pointer member.

When an instance of `indirect<T>` is used in move construction or move assignment, the moved-from instance becomes valueless: the member pointer is `nullptr`.

Early drafts of `indirect<T>` [1] had preconditions on all member functions, apart from destruction and assignment, that `this` was not in a valueless state. Equality and comparison also had a precondition that neither the left-hand-side or right-hand-side operand was valueless. While the standard requires only that moved-from objects are in a *valid but unspecified state*, there was strong feeling from implementers that adding preconditions so liberally left the undefined behaviour surface of `indirect` too large. In particular, people were concerned

that generic code may copy, move from and compare objects in a potentially valueless state.

In the current working draft of the C++ standard, `indirect<T>` must not be in a valueless state for `operator->` and `operator*` (const and non-const overloads). This is consistent with other types with a potential null state like `unique_ptr` and `optional`.

Requested changes

National Body Comment US 77-140 would require the addition of new member functions to `indirect`:

```
constexpr operator const T&() const & noexcept;
constexpr operator T&() & noexcept;
constexpr operator const T&&() const && noexcept;
constexpr operator T&&() && noexcept;
```

Authors' stance

The authors are opposed to the addition of implicit conversions to reference (and rvalue-reference).

National Body Comment US 77-140 states that "With the proposed change, indirect is closer to reference_wrapper". It is not clear why this is desirable. `reference_wrapper` is non-owning and has no null or valueless state. The current API for `indirect` is most similar to `optional` and `unique_ptr`, which have `operator*` returning `T&` rather than an implicit conversion.

Type	Owning	Null/Valueless state	Member Function	Return Type
`unique_ptr`	Yes	Yes	`operator->`	`T*`
`unique_ptr`	Yes	Yes	`operator*`	`T&`
`optional`	Yes	Yes	`operator->`	`T*`
`optional`	Yes	Yes	`operator*`	`T&`
`reference_wrapper`	No	No	`get()`	`T&`
`reference_wrapper`	No	No	`operator T&`	`T&`
`indirect`	Yes	Yes	`operator->`	`T*`
`indirect`	Yes	Yes	`operator*`	`T&`

The implicit conversions to reference would have the precondition that `this` is not in a valueless state. Having modified the design of indirect to reduce the number of non-valueless preconditions, the authors are reluctant to see opportunities for undefined behaviour introduced at this late stage in the standardization process.

Future direction

With compelling implementation experience, it would be possible to introduce implicit reference conversions for `indirect` in a later version of the C++ Standard.

The current design of `indirect` does not block later introduction of implicit conversions.

Acknowledgements

Many thanks to Neelofer Banglawala and Jonathan Wakely for useful input, review and discussion.

References

[1] p3019r1: *Vocabulary Types for Composite Class Design*,
J. B. Coe, A. Peacock, and S. Parent, 2023
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p3019r1.pdf