Against implicit conversions for `indirect`

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3902R2

Working Group: Library Evolution

Date: 2025-11-6

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 usage and 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 standard library algorithms.

In the current working draft of the C++ standard, the precondition that `indirect<T>` must not be in a valueless state is present only for `operator->` and `operator*` (including const-qualified and reference-qualified overloads). This is consistent with other standard library types with a null state such as `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 usage and 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

History

Changes in R2

Add Appendix discussing recursive variants.

Changes in R1

- Add clarification that `operator->` and `operator*` (including const-qualified and reference-qualified overloads) are the only functions with preconditions for which the associated `indirect` instance(s) must not be in a valueless state.
- Add clarification that the authors would like to see implementation and **usage** experience to motivate the introduction of a `reference_wrapper`-like API.

Appendix: Recursive variants and `OneOf`

Taken from the National Body Comment author's variant library (OneOf) there is a type `indirection`, similar to `indirect` and reproduced below.

`indirection` has a reference-wrapper-like API but has much more scope for undefined behaviour than `indirect` as accepted into the C++26 working draft.

Copy construction, move construction, assignment and move assignment of `indirection` all require that `other` has not been moved-from. `operator T&` and `get` (const and non-const qualified) all require that `this` has not been moved from. As designed, `indirection` gives a user no way to check that an instance of `indirection` has not been moved from.

https://github.com/lichray/oneof/blob/master/include/stdex/oneof.h

```
template <typename T, typename U>
using disable_capturing =
    std::enable_if_t<!std::is_base_of<T, std::remove_reference_t<U>>::value,
                     int>;

template <typename T>
struct indirection {
 public:
  indirection() : p_(new T{}) {}

  indirection(indirection&& other) noexcept : p_(other.p_) {
    other.p_ = nullptr;
  }

  indirection& operator=(indirection&& other) noexcept {
    this->~indirection();
    return *::new ((void*)this) indirection{std::move(other)};
  }

  indirection(indirection const& other) : indirection(other.get()) {}

  indirection& operator=(indirection const& other) {
    if (p_) {
      get() = other.get();
    } else {
      indirection tmp{other};
      swap(*this, tmp);
    }

    return *this;
  }

  template <typename A, typename... As, disable_capturing<indirection, A> = 0>
  explicit indirection(A&& a, As&&... as)
      : p_(new T(std::forward<A>(a), std::forward<As>(as)...)) {}

  ~indirection() { delete p_; }

  friend void swap(indirection& x, indirection& y) noexcept {
    std::swap(x.p_, y.p_);
  }
```

```
  operator T&() { return this->get(); }

  operator T const&() const { return this->get(); }

  T& get() { return *p_; }

  T const& get() const { return *p_; }

 private:
  T* p_;
};
```

The variant in OneOf uses `indirection` to implement recursive variants.

A recursive variant can be implemented with `indirect` but `indirect`'s absence of an implicit conversion to a reference type means that a pointer-like interface is exposed to users.

Improvements to the usability of recursive variants could be made by introducing another type to automatically dereference indirect storage. We illustrate the use of a helper type below.

Any design to better support recursive variants would be best addressed with a concrete proposal and the authors of indirect invite the author of the National Body Comment to put forward a paper for the C++29 standard.

```
template <typename T>
struct deref {
  T t_;

  using U = decltype(*std::declval<T>());

  operator U&() { return *t_; }

  operator const U&() const { return *t_; }
};

struct ASTNode;
using ASTNodeRecursiveStorage = deref<xyz::indirect<ASTNode>>;
using ASTNodeData = std::variant<int, std::string, ASTNodeRecursiveStorage>;

struct ASTNode {
  ASTNodeData data_;
};

/// Access tests.

template <class... Ts>
```

```
struct overload : Ts... {
  using Ts::operator()...;

  overload(Ts&&... ts) : Ts(std::forward<Ts>(ts))... {}
};

TEST(RecursiveVariant, ExplicitAccess) {
  ASTNode node;

  // This is a pain to write and exposes implementation details.
  int result =
      std::visit(overload([](const int&) { return 0; },          //
                          [](const std::string&) { return 1; },  //
                          [](const ASTNodeRecursiveStorage&) { return 2; }),
                 node.data_);

  EXPECT_EQ(result, 0);
}

TEST(RecursiveVariant, DerefAccess) {
  ASTNode node;

  // This is nicer to write.
  int result = std::visit(overload([](const int&) { return 0; },          //
                                   [](const std::string&) { return 1; },  //
                                   [](const ASTNode&) { return 2; }),
                          node.data_);

  EXPECT_EQ(result, 0);
}

TEST(RecursiveVariant, LazyAccess) {
  ASTNode node;

  // This is lazy.
  int result = std::visit(overload([](const int&) { return 0; },          //
                                   [](const std::string&) { return 1; },  //
                                   [](const auto&) { return 2; }),
                          node.data_);

  EXPECT_EQ(result, 0);
}
```