Document number: P2027
Date: 2020-01-10
Reply To: Geoff Romer (gromer@google.com)
Audience: LEWG, LWG

Moved-from objects need not be valid

Abstract

The library specification of `movable` and *Cpp17MoveConstructible* (among others) currently requires that move operations must leave the source object in a "valid but unspecified state". However, this requirement is inconsistent with the widespread expectation that special member functions should usually rely on the default implementation, because defaulted move operations very often do not produce a "valid" state, and (unlike defaulted destructors and copy operations) this problem typically cannot be corrected by a better choice of member types.

This paper proposes to resolve the inconsistency by relaxing the library requirement, and instead require only that the source object is left in a state that can be destroyed or assigned to.

Problem

According to the standard, this seemingly-reasonable program has undefined behavior:

```
class IndirectInt {
  std::shared_ptr<const int> i_;
 public:
  IndirectInt(int i) : i_(std::make_shared<const int>(i)) {}

  friend auto operator<=>(const IndirectInt& lhs, const IndirectInt& rhs) {
    return *lhs.i_ <=> *rhs.i_;
  }
};

int main() {
  std::vector<IndirectInt> v = {1, 2, 3};
  std::ranges::sort(v);
}
```

The elements of a range passed to `sort` must model `move_constructible`, but according to [concept.moveconstructible], a type models `move_constructible` only if, among other things, the source of the move is left in a "valid but unspecified" state. [defns.valid] defines "valid but unspecified state" as a "value of an object that is not specified except that the object's invariants are met and operations on the object behave as specified for its type". Comparison operations on an `IndirectInt` that has been moved do not "behave as specified", because `operator<=>` unconditionally dereferences `i_`, which will be null after a move.

This problem is not confined to the C++20 range algorithms; the *Cpp17MoveConstructible* named requirement states that the source of the move "must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether rv has been moved from or not." This is admittedly in a non-normative note, so we could debate whether a violation of this requirement technically has undefined behavior, but the committee's intent seems clear: move operations must leave the source object in a valid state, i.e. a state that satisfies the type's invariants, and supports any operation that doesn't have a precondition on the object's value.

It's possible to fix this example by adding a null check to `operator<=>`, but that requires abandoning the intended class invariant that `*i_` is always a live `int`. If we want to retain that invariant, we must define the move operations explicitly:

```
class IndirectInt {
  std::shared_ptr<const int> i_;
 public:
  IndirectInt(int i) : i_(std::make_shared<const int>(i)) {}

  IndirectInt(IndirectInt&& rhs) : i_(rhs.i_) { i_.swap(rhs.i_); }
  IndirectInt& operator=(IndirectInt&& rhs) { i_.swap(rhs.i_); }
  IndirectInt(const IndirectInt&) = default;
  IndirectInt& operator=(const IndirectInt&) = default;

  friend auto operator<=>(const IndirectInt& lhs, const IndirectInt& rhs) {
    return *lhs.i_ <=> *rhs.i_;
  }
};
```

This contravenes the class design principle (sometimes called the "rule of zero") that classes should be designed so that the default special member functions are correct (see e.g. C++ Core Guideline C.20). Of course, that's a best practice rather than an absolute law, but it is striking that such a pedestrian class is unable to follow it.

The fix also contradicts some formulations of the "rule of five" (see e.g. C++ Core Guideline C.21), because whereas we must explicitly define the move operations, the defaulted destructor and copy operations are completely unproblematic. Why are the move operations different?

"Valid but unspecified" is hostile to composition

The problem with defaulted move operations is that the "valid but unspecified" postcondition does not compose: if all data members are in a valid but unspecified state, that does not imply that the object as a whole is in a valid but unspecified state. In fact, it's almost the reverse: if all members are in an unspecified state, then the object's state will *not* be valid (except by coincidence) if its type has any state invariants at all. The only partial saving grace is that some types go beyond the "valid but unspecified" postcondition, and specify their moved-from state precisely. For example, `IndirectInt` can be fixed by adding a null check to `operator<=>`, but that relies on the fact that `shared_ptr`'s move constructor is guaranteed to leave the source object holding null, rather than any other valid state.

To state the point in general terms: a defaulted move operation for a class C *will necessarily be incorrect* unless every data member involved in C's invariants has a type that provides a stronger move postcondition than "valid but unspecified", and those stronger postconditions happen to cumulatively imply that C's invariants are satisfied.

Of course, it's possible for defaulted destructors and copy operations to be incorrect too, but it's almost always possible to make those operations correct through a better choice of member types, unless the class is doing something quite unusual (e.g. having an invariant involving the address of one of its members). Defaulted move operations are unique in that they can be unavoidably incorrect even for well-designed classes that are doing nothing out of the ordinary.

"Valid but unspecified" is overbroad

I am aware of no library implementation that has any trouble with the example above; the undefined behavior appears to be entirely theoretical. In fact, as far as I know no implementation of *any* standard library algorithm misbehaves when given `IndirectInt` values. Furthermore, it seems *a priori* unlikely that any such algorithms exist. Not only would such algorithms choke on a great many reasonable types, it also just seems like an absurd thing for an algorithm to do in the first place: while a moved-from object may be technically valid, it's also useless (unless there a stronger constraint than "valid" is in effect), so it's very difficult to see how efficient correct code could usefully invoke an operation like comparison on one. Indeed, it appears to be a common guideline to avoid almost all operations on moved-from objects, and there are even static analysis tools to help us avoid doing so by accident.

Thus, we find ourselves in the position of asserting that certain natural and commonplace coding patterns are incorrect, despite the fact that they will never break in practice. I submit that when that's the case, the problem is not in the code, but rather in our definition of correctness. The "valid but unspecified" requirement appears to be mostly reserving the standard library's right to do things that it shouldn't and doesn't do. A much less restrictive rule would probably suffice.

Solution

We typically don't invoke arbitrary operations on moved-from objects (especially in generic code), but there are two operations that we commonly do invoke and expect to work: destroying the object, and assigning a new value to it. Notice, for example, that both the CERT guideline and the static analysis cited above exclude precisely those two operations.

I will use the term "partially formed" to describe object states that can be destroyed or assigned to, but might not otherwise satisfy the type's invariants. That term is drawn from *The Elements of Programming* by Stepanov and McJones, who identified this as a useful and natural category even though their subset of C++ lacked move semantics. Note that, unlike them, I am not proposing "partially formed" as a postcondition for default constructors — while that model has much to recommend it in theory, it goes against the longstanding C++ convention that constructors should always fully initialize the object.

Returning to our motivating example, a moved-from `IndirectInt` is not "valid", because it does not satisfy the invariant that `*i_` is a live `int`, and cannot safely be used in operations like comparison. However, it is partially formed, because it is safe to assign to or destroy, which is why no library implementation has a problem with it. This isn't a fluke: defaulted move operations naturally leave the source in a partially formed state, because "partially formed" is inherently composable: if it's safe and correct to destroy every data member, that should be sufficient to safely and correctly destroy the object — if that were not the case, the destructor could not be defaulted. Similarly, if it's safe and correct to assign a new value to every data member, that should be sufficient to safely and correctly assign a new value to the object — if that were not the case, the destructor and/or the copy operations could not be defaulted.

I therefore propose that the standard library should only require move operations to leave the source object in a partially formed state, not necessarily a valid (i.e. well formed) state. This will eliminate a subtle but widespread gotcha in post-C++11 class design, by ensuring that whenever the defaulted destructor and copy operations are correct, the defaulted move operations are almost certainly correct as well.

Possible objections

Does this mean I couldn't call `clear()` on a moved-from `vector`?

I am not proposing any change to the move semantics of existing standard library types; they will continue to satisfy the stronger "valid but unspecified" postcondition. This proposal is purely a relaxation of the requirements on user-defined types.

Doesn't this mean allowing live objects to violate their class invariants?

For user-defined types, this proposal would drop the requirement that after a move, "the object’s invariants are met", so this creates the possibility that standard-conforming C++ programs might contain moved-from objects that don't satisfy their own invariants. However, that prospect is not so dire as it may appear.

First of all, notice how bizarre it is to have a normative requirement that depends on the invariants of a user-defined type. Does it cover invariants that are documented outside the source code? Does it cover invariants that exist in the mind of the programmer, but aren't documented at all? To the extent that this passage applies to user-defined types, it can only be understood as best-practice guidance, and best-practice guidance doesn't belong in the standard to begin with.

Even as best-practice guidance, I'm no longer convinced this principle is sound. We already know and accept that an object's invariants might not hold during construction or destruction, but that's OK because we adopt practices (like avoiding method calls in constructors and destructors) that make it easy to tell whether an object's invariants hold in a given context. In a world where moved-from objects are allowed to violate their invariants, it's still easy to tell whether an object's invariants hold, so long as we follow the existing conventional wisdom that moved-from states are "radioactive" (i.e. should only be assigned to or destroyed).

You may not agree with me about this; it's certainly simpler in some ways to insist that invariants must hold throughout an object's lifetime. However, nothing about this proposal prevents you from holding that view. If it's adopted, we can agree that the C++ standard defines the behavior of my `IndirectInt` example, while still disagreeing about whether it's advisable to implement `IndirectInt` that way, and whether I should use the term "invariant" to refer to the property that `*i_` is a live `int`.

"Partially formed" isn't sufficient for `std::swap`

Consider the canonical implementation of `std::swap`:

```
template <typename T>
void swap(T& lhs, T& rhs) {
  T tmp = std::move(lhs);
  lhs = std::move(rhs);
  rhs = std::move(tmp);
}
```

If `lhs` and `rhs` refer to the same object, this code will move that object twice without an intervening assignment. The EoP definition of "partially formed" does not require a partially formed state to be usable as the *source* of an assignment, so the "partially formed" postcondition is formally insufficient to make this code correct for the self-swap case. We could avoid that by adding a branch on `&lhs == &rhs`, but that would make `swap` less efficient, which we should strenuously avoid in such a low-level algorithmic primitive.

However, this argument presupposes that `std::swap` must support cases where `rhs` and `lhs` refer to the same object, but that doesn't appear to be the case. The semantics of `swap` are specified in [utility.swap]/p3 as "*Effects:* Exchanges values stored in two locations", so when there is only one location, the behavior seems to be at best unspecified, if not undefined. Furthermore, it seems unlikely that generic code ever has a valid reason to perform such a swap. Note, for example, that all three major implementations of `std::shuffle` have an explicit branch to avoid swapping an item with itself.

In any event, this concern seems academic: it is very difficult to imagine a plausible type T that actually breaks this implementation of `swap` in practice. By the same token, we can easily resolve this problem by requiring that repeatedly moving from `T` is safe, either as an additional requirement on `swap`, or as part of the definition of "partially formed". I have avoided that in my proposal primarily to keep the wording simple, and to avoid diverging from the EoP definition of "partially formed".

Are you sure the standard library never operates on moved-from values?

I have not surveyed library implementations to determine whether any algorithm actually does need the stronger "valid but unspecified" postcondition, and I'm not sure whether an exhaustive survey is even possible. However, I claim such a survey is unnecessary, because this change is essentially a wording fix, aligning the formal specification with our users' pre-existing expectation that defaulted move operations will generally work. This change would not invalidate any user code (but see the caveat below), and if it invalidates any standard library code, I would argue that code was already broken with respect to user expectations, if not with respect to the formal standard.

Furthermore, this proposal does not preclude individual algorithms from requiring "valid but unspecified" (in the unlikely event that they need it enough to warrant violating user expectations in that way); it merely means that they must state that requirement explicitly, rather than leaving it implicit in their use of `move_constructible` et al.

This is a special case of a much bigger problem

Move semantics are not the only case where the standard library's requirements on types are seemingly too strict. Consider the following code:

```
std::vector<double> v = {2.0, 1.0, 3.0};
std::ranges::sort(v);
```

`sort` requires its comparator (in this case `std::less<double>`) to model `strict_weak_order`, which means that `std::less<double>` must impose a strict weak order on its arguments. On platforms where `double` has IEEE floating point semantics, that's not the case due to the semantics of NaN.

The standard addresses this problem in [structure.requirements]/p8:

> "Required operations of any concept defined in this document need not be total functions; that is, some arguments to a required operation may result in the required semantics failing to be met. [*Example:* The required `<` operator of the `totally_ordered` concept (18.5.4) does not meet the semantic requirements of that concept when operating on NaNs. — *end example*] This does not affect whether a type models the concept."

However, that amounts to little more than handwaving; it gives no guidance about which arguments are and are not bound by a concept's requirements. Indeed, arguably this handwave is capacious enough to make the original `IndirectInt` example valid.

From that point of view, this paper is not making a normative change, but only clarifying that handwaving with respect to moved-from objects. It's possible that a more general solution to this problem could entirely eliminate the handwaving, so some people have advocated seeking that general solution, rather than focusing on the special case of moved-from objects.

However, this problem is for the most part quite theoretical; in practice, there is no debate, or even any real uncertainty, about whether it's safe to sort a range of `double` values that doesn't contain NaN; we're just not sure how to make the words in the standard rigorously express what we all know. By contrast, there is genuine uncertainty about whether the `IndirectInt` example is safe, and genuine debate about whether we should teach people to avoid writing classes like `IndirectInt`.

The fix proposed in this paper does not in any way preclude a future solution to the more general problem. That being the case, it would be a mistake to postpone solving a practical problem that has a concrete solution in the hope that it will be addressed by an as-yet-unknown solution to a much broader class of theoretical problems.

We can't change concepts after shipping them

There is an emerging conventional wisdom that any change to a library concept is a breaking change, and so can't be made after the concept has shipped in an IS. That poses an obvious problem for this paper, which proposes to change the definition of `movable` and related library concepts. However, the principle that all concept changes are breaking changes applies with much less force to changes that, like this one, have no effect other than to remove semantic requirements that are rarely if ever relied upon.

A change to the semantic requirements of a concept (as opposed to the syntactic requirements) should be understood as a documentation change, not a code change. It cannot change whether any program is well formed, and it cannot change the observable behavior of any program; realistically, it cannot even change the diagnostics produced for an ill formed program. The only risk is that by making such a change, we are implicitly modifying the documentation of APIs constrained by that concept (including APIs outside the standard). In other words, changing the semantic requirements of a C++20 concept has exactly the same risks as a corresponding change to a pre-C++20 named requirement, and for exactly the same reasons: C++ programmers can and do use standard named requirements in their documentation, and that hasn't stopped us from making major changes to them. For example, C++11 dropped the requirement that *Swappable* types be copyable, a much more drastic change than what I'm proposing.

Of course, API documentation is important, and we should not change its meaning on a whim. But neither should we treat every documentation change with the same seriousness as a build break or ABI break, because documentation is inherently less rigid than code. Instead, we should ask whether the change might cause misunderstandings, and how bad those misunderstandings will be.

When a change adds new requirements to a concept (so that some types no longer model it), that could license generic libraries to change their implementations in a way that breaks client code, but that risk doesn't apply to changes that only drop requirements. The only potential risk is that generic code constrained by these concepts might be relying on the dropped requirements, and so might break when it's used with types that don't satisfy them. Even then, it will be exactly the same kind of breakage that can occur anyway if the caller simply overlooks the semantic requirement. In particular, the breakage will only affect newly-written code; it can't be triggered by upgrading your language or standard library version.

So the risk of dropping a semantic requirement is minimal if APIs rarely rely on that requirement (and therefore are rarely affected by dropping it), and if callers often overlook the requirement (and therefore are already dealing with whatever problems would be caused by dropping it). The requirement that moved-from states be "valid" has both of those properties.

Proposed Wording

Edits are relative to N4830.

Edit [definitions] as follows:

16.3.?   [defns.well.formed]

well formed state
value of an object such that the object’s invariants are met and operations on the object behave as specified for its type

[*Example:* If an object x of type `std::vector<int>` is in a well formed state, `x.empty()` can be called unconditionally, and `x.front()` can be called only if `x.empty()` returns `false`. — *end example*]

16.3.29 [defns.valid]

valid but unspecified state
value of an object that is not specified except that it is well formed ~~the object's invariants are met and operations on the object behave as specified for its type~~
~~[*Example:* If an object x of type `std::vector<int>` is in a valid but unspecified state, `x.empty()` can be called unconditionally, and `x.front()` can be called only if `x.empty()` returns `false`. — *end example*]~~

16.3.? [defns.partially.formed]

partially formed state
value of an object that is not specified except that if the object is destroyed or assigned to, those operations behave as specified.
[*Note:* A state that is well formed is also partially formed. — *end note*]

Edit [structure.requirements] as follows:

16.4.1.3 Requirements [structure.requirements]

Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

- Template arguments
- Derived classes
- Containers, iterators, and algorithms that meet an interface convention or model a concept

The string and iostream components use an explicit representation of operations required of template arguments. They use a class template `char_traits` to define these constraints.

Interface convention requirements are stated as generally as possible. Instead of stating "class X has to define a member function `operator++()`", the interface requires "for any object x of class X, ++x is defined". That is, whether the operator is a member is unspecified.

Requirements are stated in terms of well-defined expressions that define valid terms of the types that meet the requirements. For every set of well-defined expression requirements there is either a named concept or a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 25) that uses the well-defined expression requirements is described in terms of the valid expressions for its template type parameters.

The library specification uses a typographical convention for naming requirements. Names in *italic* type that begin with the prefix *Cpp17* refer to sets of well-defined expression requirements typically presented in tabular form, possibly with additional prose semantic requirements. For example, *Cpp17Destructible* (Table 30) is such a named requirement. Names in `constant width` type refer to library concepts which are presented as a concept definition (Clause 13), possibly with additional prose semantic requirements. For example, `destructible` (18.4.10) is such a named requirement.

Template argument requirements are sometimes referenced by name. See 16.4.2.2.

In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.

Required operations of any concept defined in this document need not be total functions; that is, some arguments to a required operation may result in the required semantics failing to be met. [*Example:* The required `<` operator of the `totally_ordered` concept (18.5.4) does not meet the semantic requirements of that concept when operating on NaNs. — *end example*] This does not affect whether a type models the concept. In particular, a required operation need not meet the required semantics when any operand is not in a well formed state, unless otherwise specified.

A declaration may explicitly impose requirements through its associated constraints (13.4.2). When the associated constraints refer to a concept (13.6.8), the semantic constraints specified for that concept are additionally imposed on the use of the declaration.

Edit [tab:cpp17.moveconstructible] as follows:

Table 26: *Cpp17MoveConstructible* requirements [tab:cpp17.moveconstructible]

Expression	**Post-condition**
`T u = rv;`	If `rv` was in a well formed state before the construction, then `u` is equivalent to that state~~the value of rv before the construction~~.
`T(rv)`	If `rv` was in a well formed state before the construction, then `T(rv)` is equivalent to that state~~the value of rv before the construction~~.
`rv`'s state is partially formed~~unspecified [*Note:* rv must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether rv has been moved from or not. — *end note*]~~	

Edit [tab:cpp17.moveassignable] as follows:

Table 28: *Cpp17MoveAssignable* requirements [tab:cpp17.moveassignable]

Expression	Return type	Return value	Post-condition
`t = rv`	`T&`	`t`	If `t` and `rv` do not refer to the same object and rv was in a well formed state before the assignment, `t` is equivalent to that state~~the value of `rv` before the assignment~~. This requirement applies even if `t` is not in a well formed state.
`rv`'s state is partially formed~~unspecified~~. ~~[Note: `rv` must still meet the requirements of the library component that is using it, whether or not `t` and `rv` refer to the same object. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — end note]~~			

Edit [tab:cpp17.destructible] as follows:

Table 30: *Cpp17Destructible* requirements [tab:cpp17.destructible]

Expression	Post-condition
`u.~T()`	All resources owned by u are reclaimed, no exception is propagated. This requirement applies even if u is not in a well formed state.
[*Note*: Array types and non-object types are not *Cpp17Destructible*. — *end note*]	

Edit [concepts.equality] as follows:

18.2 Equality preservation [concepts.equality]

An expression is *equality-preserving* if, given equal well formed inputs, the expression results in equal outputs. The inputs to an expression are the set of the expression's operands. The output of an expression is the expression's result and all operands modified by the expression.

A given expression need not be well defined for all well formed input values~~Not all input values need be valid for a given expression~~; e.g., for integers a and b, the expression a / b is not well-defined when b is 0. This does not preclude the expression a / b being equality-preserving. The *domain* of an expression is the set of input values for which the expression is required to be well-defined.

Expressions required by this document to be equality-preserving are further required to be stable: two evaluations of such an expression with the same well formed input objects are required to have equal outputs absent any explicit intervening modification of those input objects. [*Note:* This requirement allows generic code to reason about the current values of objects based on knowledge of the prior values as observed via equality-preserving expressions. It effectively forbids spontaneous changes to an object, changes to an object from another thread of execution, changes to an object as side effects of non-modifying expressions, and changes to an object as side effects of modifying a distinct object if those changes could be observable to a library function via an equality-preserving expression that is required to be valid for that object. — *end note*]

Edit [concept.convertible] as follows:

18.4.4 Concept `convertible_to` [concept.convertible]

The `convertible_to` concept requires an expression of a particular type and value category to be both implicitly and explicitly convertible to some other type. The implicit and explicit conversions are required to produce equal results.

```
template<class From, class To>
  concept convertible_to =
    is_convertible_v<From, To> &&
    requires(From (&f)()) {
      static_cast<To>(f());
    };
```

Let `test` be the invented function:

```
To test(From (&f)()) {
  return f();
}
```

for some types `From` and `To`, and let `f` be a function with no arguments and return type `From` such that `f()` is equality-preserving. `From` and `To` model `convertible_to<From, To>` only if:

- `To` is not an object or reference-to-object type, or `static_cast<To>(f())` is equal to `test(f)`.
- `From` is not a reference-to-object type, or
- If `From` is an rvalue reference to a non const-qualified type, the resulting state of the object referenced by `f()` after either above expression is partially formed ([defns.partially.formed])~~valid but unspecified (16.5.5.15)~~.
- Otherwise, the object referred to by `f()` is not modified by either above expression.

Edit [concept.assignable] as follows:

18.4.8 Concept `assignable_from` [concept.assignable]

```
template<class LHS, class RHS>
  concept assignable_from =
    is_lvalue_reference_v<LHS> &&
    common_reference_with<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&> &&
    requires(LHS lhs, RHS&& rhs) {
      { lhs = std::forward<RHS>(rhs) } -> same_as<LHS>;
    };
```

Let:

- `lhs` be an lvalue that refers to an object `lcopy` such that `decltype((lhs))` is `LHS`,
- `rhs` be an expression such that `decltype((rhs))` is `RHS`, and
- `rcopy` be a distinct object that is equal to `rhs` if `rhs` is well formed, and not well formed otherwise.

`LHS` and `RHS` model `assignable_from<LHS, RHS>` only if

- `addressof(lhs = rhs) == addressof(lcopy)`.
- After evaluating `lhs = rhs`:
  - `lhs` is equal to `rcopy`, unless `rhs` is a non-const xvalue that refers to `lcopy` or `rcopy` is not well formed.
  - If `rhs` is a non-`const` xvalue, the resulting state of the object to which it refers is ~~valid but unspecified (16.5.5.15)~~partially formed.
  - Otherwise, if `rhs` is a glvalue, the object to which it refers is not modified.

[*Note:* Assignment need not be a total function (16.4.1.3); in particular, if assignment to an object `x` can result in a modification of some other object `y`, then `x = y` is likely not in the domain of `=`. — *end note*]

Edit [concept.destructible] as follows:

18.4.10 Concept `destructible` [concept.destructible]

The `destructible` concept specifies properties of all types, instances of which can be destroyed at the end of their lifetime, or reference types.

```
    template<class T>
      concept destructible = is_nothrow_destructible_v<T>;
```

If `T` is an object type, then let `v` be a partially formed lvalue of type (possibly `const`) `T`. `T` models `destructible` only if the expression `v.~T()` has well defined behavior.

[*Note:* Unlike the *Cpp17Destructible* requirements (Table 30), this concept forbids destructors that are potentially throwing, even if a particular invocation of the destructor does not actually throw. — *end note*]

Edit [concept.moveconstructible] as follows:

18.4.13 Concept `move_constructible` [concept.moveconstructible]

```
template<class T>
  concept move_constructible = constructible_from<T, T> && convertible_to<T,
T>;
```

If `T` is an object type, then let `rv` be a~~n~~ partially formed rvalue of type `T`, and `u2` a distinct object of type `T` that is equal to `rv` if rv is well formed, and not well formed otherwise. `T` models `move_constructible` only if

— After the definition `T u = rv;`, `u` is equal to `u2` if u2 is well formed.

— `T(rv)` is equal to `u2` if u2 is well formed.

— If `T` is not `const`, `rv`'s resulting state is partially formed ([defns.partially.formed])~~valid but unspecified (16.5.5.15)~~; otherwise, it is unchanged.

Edit [uninitialized.move] as follows:

20.10.11.6 uninitialized_move [uninitialized.move]

```
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

Expects: `[result, (last - first))` shall not overlap with `[first, last)`.

Effects: Equivalent to:

```
    for (; first != last; (void)++result, ++first)
      ::new (voidify(*result))
        typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
    return result;
```

```
namespace ranges {
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_result<I, O>
      uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<input_range IR, no-throw-forward-range OR>
      requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
    uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
      uninitialized_move(IR&& in_range, OR&& out_range);
}
```

Expects: `[ofirst, olast)` shall not overlap with `[ifirst, ilast)`.

Effects: Equivalent to:

```
    for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
      ::new (voidify(*ofirst))
        remove_reference_t<iter_reference_t<O>>(ranges::iter_move(ifirst));
    }
    return {ifirst, ofirst};
```

[*Note:* If an exception is thrown, some objects in the range `[first, last)` are left in a ~~valid, but unspecified~~partially formed state. — *end note*]

```
template<class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator>
    uninitialized_move_n(InputIterator first, Size n, ForwardIterator result);
```

Expects: `[result, n)` shall not overlap with `[first, n)`.
Effects: Equivalent to:

```
    for (; n > 0; ++result, (void) ++first, --n)
      ::new (voidify(*result))
        typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
    return {first,result};
```

```
namespace ranges {
  template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
      requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_n_result<I, O>
      uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}
```

Expects: `[ofirst, olast)` shall not overlap with `[ifirst, n)`.
Effects: Equivalent to:

```
    auto t = uninitialized_move(counted_iterator(ifirst, n),
                                default_sentinel, ofirst, olast);
    return {t.in.base(), t.out};
```

[*Note:* If an exception is thrown, some objects in the range `[first, n)` are left in a ~~valid but unspecified~~partially formed state. — *end note*]

Edit [iterator.concept.writable] as follows:

23.3.4.3 Concept `writable` [iterator.concept.writable]

The writable concept specifies the requirements for writing a value into an iterator's referenced object.

```
    template<class Out, class T>
      concept writable =
        requires(Out&& o, T&& t) {
          *o = std::forward<T>(t); // not required to be equality-preserving
          *std::forward<Out>(o) = std::forward<T>(t); // not required to be equality-preserving
          const_cast<const iter_reference_t<Out>&&>(*o) =
            std::forward<T>(t); // not required to be equality-preserving
          const_cast<const iter_reference_t<Out>&&>(*std::forward<Out>(o)) =
            std::forward<T>(t); // not required to be equality-preserving
    };
```

Let `E` be an an expression such that `decltype((E))` is `T`, and let `o` be a dereferenceable object of type `Out`. `Out` and `T` model `writable<Out, T>` only if

- If `Out` and `T` model `readable<Out> && same_as<iter_value_t<Out>, decay_t<T>>`, then `*o` after any above assignment is equal to the value of `E` before the assignment.

After evaluating any above assignment expression, `o` is not required to be dereferenceable.

If `E` is an xvalue (7.2.1), the resulting state of the object it denotes is~~ valid but unspecified (16.5.5.15)~~partially formed ([defns.partially.formed]).

[*Note:* The only valid use of an `operator*` is on the left side of the assignment statement. Assignment through the same value of the writable type happens only once. — *end note*]

[*Note:* `writable` has the awkward `const_cast` expressions to reject iterators with prvalue non-proxy reference types that permit rvalue assignment but do not also permit `const` rvalue assignment. Consequently, an iterator type `I` that returns `std::string` by value does not model `writable<I, std::string>`. — *end note*]

Edit [alg.req.ind.move] as follows:

23.3.7.2 Concept `indirectly_movable` [alg.req.ind.move]

The indirectly_movable concept specifies the relationship between a `readable` type and a `writable` type between which values may be moved.

```
    template<class In, class Out>
      concept indirectly_movable =
        readable<In> &&
        writable<Out, iter_rvalue_reference_t<In>>;
```

The `indirectly_movable_storable` concept augments `indirectly_movable` with additional requirements enabling the transfer to be performed through an intermediate object of the `readable` type's value type.

```
    template<class In, class Out>
    concept indirectly_movable_storable =
      indirectly_movable<In, Out> &&
      writable<Out, iter_value_t<In>> &&
      movable<iter_value_t<In>> &&
      constructible_from<iter_value_t<In>, iter_rvalue_reference_t<In>> &&
      assignable_from<iter_value_t<In>&, iter_rvalue_reference_t<In>>;
```

Let `i` be a dereferenceable value of type `In`. `In` and `Out` model `indirectly_movable_storable<In, Out>` only if after the initialization of the object `obj` in

```
iter_value_t<In> obj(ranges::iter_move(i));
```

`obj` is equal to the value previously denoted by `*i`. If `iter_rvalue_reference_t<In>` is an rvalue reference type, the resulting state of the value denoted by `*i` is ~~valid but unspecified (16.5.5.15)~~partially formed ([defns.partially.formed]).

Edit [alg.req.ind.copy] as follows:

23.3.7.3 Concept `indirectly_copyable` [alg.req.ind.copy]

The `indirectly_copyable` concept specifies the relationship between a `readable` type and a `writable` type between which values may be copied.

```
template<class In, class Out>
  concept indirectly_copyable =
    readable<In> &&
    writable<Out, iter_reference_t<In>>;
```

The `indirectly_copyable_storable` concept augments `indirectly_copyable` with additional requirements enabling the transfer to be performed through an intermediate object of the `readable` type's value type. It also requires the capability to make copies of values.

```
template<class In, class Out>
  concept indirectly_copyable_storable =
    indirectly_copyable<In, Out> &&
    writable<Out, const iter_value_t<In>&> &&
    copyable<iter_value_t<In>> &&
    constructible_from<iter_value_t<In>, iter_reference_t<In>> &&
    assignable_from<iter_value_t<In>&, iter_reference_t<In>>;
```

Let `i` be a dereferenceable value of type `In`. `In` and `Out` model `indirectly_copyable_storable<In, Out>` only if after the initialization of the object `obj` in `iter_value_t<In> obj(*i);`
`obj` is equal to the value previously denoted by `*i`. If `iter_reference_t<In>` is an rvalue reference type, the resulting state of the value denoted by `*i` is ~~valid but unspecified (16.5.5.15)~~partially formed ([defns.partially.formed]).