Relaxing Requirements of Moved-From Objects

Mar 31, 2021

Document Number:	P2345R0
Date:	2021-04-14
Reply-to:	Sean Parent, sean.parent@stlab.cc
Audience:	LWG & LEWG

Table of Contents

Introduction

The C++ Standard Library requirements are overly restrictive regarding the state of a moved-from object. The strong requirements impose an unnecessary burden on implementers and imposes a performance impact of user-defined operations.

The issue was recognized in Geoffrey Romer's paper, P2027R0. The approach outlined here differs in the following ways:

- The requirements are (slightly) stronger to support `swap(a, a)`

- It avoids introducing a new object state, *partially formed*, (see Future Implications)

This paper details the issue and presents some suggested wording to address it. Depending on the wording chosen, it may be possible to address the issue with a defect report retroactively.

Document Conventions

This is the proposed wording for the standard. There may be more than one proposed variant for the same section.

> This is a quote from an existing document.

This is a comment or work in progress.

This document discusses *requirements* in multiple different contexts. The following terms are used when the meaning is otherwise ambiguous. The *Standard requirements* refer to the current, C++20, documented requirements. *Implementation requirements* refers to the actual requirements necessary to implement the library components. This may be weaker than the stated requirements. Finally there are the *proposed requirements*, this is the proposed wording to bring the *Standard requirements* more inline with the *implementation requirements*.

Motivation and Scope

Given an object, `rv`, which has been moved from, the C++20[1] Standard specifies the required postconditions of a moved-from object:

> `rv` 's state is 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*] — Table 28, p. 488 C++20 Standard.

The Standard requirement applies to both *Cpp17MoveConstructible* and *Cpp17MoveAssignable*. The note is not normative but does clarify that the requirements on a moved-from object are not relaxed.

In general, unless move is specified to make a copy, the Standard requirement is not achievable. For example, the sorting algorithms require `comp(*i, *j)` induce a strict weak ordering. Therefore, a moved-from object must be ordered with respect to every other value in the sequence with an arbitrary user-supplied comparison function. Only a value within the initial sequence could satisfy that requirement.

No implementation of the Standard Library will ever invoke a comparison, user-defined or otherwise, on an object that a library component itself moved-from during the course of the same operation. Such a comparison would not have meaning.

The Standard has a slightly different requirement for the `move_constructible` concept:

> If `T` is not const, `rv`'s resulting state is valid but unspecified ([lib.types.movedfrom]); otherwise, it is unchanged.

And from the requirements for the `assignable_from` concept:

> If `rhs` is a non-const xvalue, the resulting state of the object to which it refers is valid but unspecified ([lib.types.movedfrom]).

The term *valid* in this context does not impose any actual requirement on the moved from object since there is no requirements about what operations must be available on a valid object. The referenced definition for *valid but unspecified* refers to a general guarantee of library types, but is not a library requirement. The only restriction on the moved from object appears to be that it is *unspecified*, which is to say there is no requirement.

The Standard even notes that assignment need not be a total function.

The way the Standard requirements for moved-from objects are frequently taught is that all operations used by Standard components must be *total* when used with a moved-from object. i.e. `rv < a` must be valid and induce a strict weak ordering for *all possible values of* `a`. However, even such a strong guarantee does not solve the issue for arbitrary operations passed to the Standard Library.

Geoffrey Romer's paper, P2027R0 makes the observation that *valid but unspecified* does not compose. The result is that any composite object requires additional code to move the object into a valid state on move.

Attempting to make all operations total with respect to moved-from objects imposes an unnecessary performance penalty and the implementation of such operations is error-prone. Examples and details are provided in an *Annoyance* I wrote for the upcoming *Embracing Modern C++ Safely*. An example directly from the standard library is detailed in the [Weaker Guarantees for Future Components] section.

In the discussion of P2027R0 there is a lot of confusion about the difference between requirements and guarantees in the Standard. In short, the standard library *requirements* impose a set of syntactic and semantic requirements on operations on arguments (both types and values) passed to a standard component. i.e., Give `std::find(f, l, v)`; it is required that `f` and `l` denotes a *valid range*.

A *guarantee* is provided by a standard component, and may be conditional on requirements of the components arguments. i.e., `std::vector<T>::operator==()` is (guaranteed to be) an equivalence relation iff `T` meets the `Cpp17EqualityComparable` requirements. The confusion in part comes from stating a guarantee as *satisfying* a named requirement.

The standard guarantees regarding moved from objects are specified in lib.types.movedfrom, as being *valid but unspecified*. The wording proposed in this paper does not change the guarantees. I included some discussion in the [*Future Implications*][weaker-guarantees-for-future-components] section about why it may be desirable to revisit this

terminology.

Requirements of a Moved-From Object

All known standard library implementations only require the following operations on an object, `mf`, that the library moved from within an operation:

- `mf.~()` (The language also requires this for implicitly moved objects)
- `mf = a`
- `mf = move(a)`
- `mf = move(mf)`

The last implementation requirement comes from `std::swap()` when invoked as `swap(a, a)`. It is note worthy that self-move-assignment is only required in the case where the object has already been moved-from. Self-swap does appear in some older standard library implementations of `std::random_shuffle()`. This underscores the need to support self-swap by the standard requirements.

Supporting self-move-assignment for this narrow case imposes some additional complexity because `a = move(a)` is, in general, a contradiction and is not required by the implementation of any standard component. The implementation required postcondition of `a = move(b)` is that `a` holds the prior value of `b` and the value of `b` is unspecified, but may be guaranteed to be a specific value. For example, if `a` and `b` are both of type `my_unique_ptr<T>` with the guarantee that `a` will hold the prior value of `b`, and `b` will be equal to `nullptr`. Then for the expression `a = move(a)`, the only way both of those guarantees could be satisfied is if `a` is already equal to `nullptr`. The current standard avoids this contradiction by defining the postcondition of move assignment for `std::unique_ptr<T>` as equivalent to `reset(r.release())` which provides a stronger guarantee than any standard component implementation requires while satisfying the Standard requirements.

Non-Requirements

There is not a standard requirement to provide guarantees across operations that result in moved-from objects. For example:

```
1 T a[]{ v0, v1, v1, v2 };
2 (void)remove(begin(a), end(a), v1);
3 sort(begin(a), end(a));
```

After `remove()`, the last two objects at the end of `a` have unspecified values and may have been moved from. There is no requirement that these moved-from objects also satisfy the requirements of `sort()` by being in the domain of the operation `operator<()`, even if `v0`, `v1`, and `v2` are within the domain. The post conditions of `remove()` and the requirements of `sort()` are independent. An invocation of `sort()` for a particular type, `T`, may or may not be valid depending on the guarantees provided by `T`.

Assuming `v0` and `v2` are in the domain of `operator<()` for `sort()` the following is guaranteed:

```
1 T a[]{ v0, v1, v1, v2 };
2 auto p = remove(begin(a), end(a), v1);
3 sort(begin(a), p);
```

As another example:

```
1 for (std::string line; std::getline(std::cin, line);) {
2   v.push_back(std::move(line));
3 }
```

For the call to `std::getline()` to be valid with a moved from string it requires that `std::string()` *guarantees* that `std::erase()` is valid on a moved from string. Changing the requirements of a moved from object does not change the guarantees of the standard components.

Impact on the Standard

All components which are *Movable* in the Standard Library currently satisfy the proposed requirements as stated by both options below. **Both options are non-breaking changes and relax the requirements.** With either option, it may be possible to adopt these options retroactively as part of addressing a defect since neither option is a breaking change.

Concern has been raised that changing the documentation for requirements, especially the named requirements and concepts, would break existing code documentation that referenced the standard. However, taking a strict view of this would mean that the standard documentation could not be changed. For example, one of the libraries I work on has a `task<>` template which is documented as being "Similar to `std::function` except it is not copyable and supports move-only and mutable callable targets…". Of note, this goes on to specify, "`stlab::task` satisfies the requirements of MoveConstructible and MoveAssignable." Weakening the requirements would mean that statement is still true.

However, the concern over changing the documentation is one that should be considered in light of weakening the requirements retroactively as a defect. As that would mean even citing a specific version of the standard would break.

Technical Specifications

We need a general requirement regarding *the domain of an operation*. Borrowing from the text for input iterators:

> Unless otherwise specified, there is a general precondition for all operations that the requirements hold for values within the *domain of the operation*.

> The term *domain of the operation* is used in the ordinary mathematical sense to denote the set of values over

which an operation is (required to be) defined. This set can change over time. Each component may place additional requirements on the domain of an operation. These requirements can be inferred from the uses that a component makes of the operation and is generally constrained to those values accessible through the operation's arguments.

The above wording should appear in the Requirements section of the Library Introduction.

Given the above general requirement, we can then specify what operations must hold for a moved-from object.

Option 1

Option 1 requires that a moved-from object can be used as an rhs argument to move-assignment only in the case that the object has been moved from and it is a self-move-assignment. It introduces a *moved-from-value* to discuss the properties of the moved-from object without specifying a specific value and requires that self-move-assignment for the moved-from object is valid. The wording allows for `swap(a, a)` without allowing `a = move(a)` in general.

Table 28: *Cpp17MoveConstructible* requirements

Expression	Assertion/note pre-/post-condition
`T u = rv;`	*Postconditions:* `u` is equivalent to the value of `rv` before the construction
`T(rv)`	*Postconditions:* `T(rv)` is equivalent to the value of `rv` before the construction

common	*Postconditions:*
	• If `T` meets the *Cpp17Destructible* requirements;
	  ◦ `rv` is in the domain of *Cpp17Destructible*
	• If `T` meets the *Cpp17MoveAssignable* requirements;
	  ◦ `rv` is in the domain of the lhs argument of *Cpp17MoveAssignable* and,
	  ◦ `rv` is a *moved-from-value*, such that following a subsequent operation, `t = (T&&)(rv)`, where `t` and `rv` refer to the same object, `rv` still satisfies the postconditions of *Cpp17MoveConstructible*
	• If `T` meets the *Cpp17CopyAssignable* requirements;
	  ◦ `rv` is in the domain of the lhs argument of *Cpp17CopyAssignable*
	• The value of `rv` is otherwise unspecified

Table 28: *Cpp17MoveAssignable* requirements

Expression	Return type	Return value	Assertion/note pre-/post-condition
`t = rv`	`T&`	`t`	*Preconditions:* `t` and `rv` do not refer to the same object, or the object is a *moved-*

			from-value (see *Cpp17MoveConstructible*)
			
			Postconditions:
			
			• If `t` and `rv` do not refer to the same object, `t` is equivalent to the value of `rv` before the assignment, otherwise the value of `t` is unspecified
			• If `T` is required to meet the *Cpp17Destructible* requirements;
			  ◦ `rv` is in the domain of *Cpp17Destructible*
			• `rv` is in the domain of the lhs argument of *Cpp17MoveAssignable*
			• If `rv` is required to meet the *Cpp17CopyAssignable* requirements;
			  ◦ `rv` is in the domain of the lhs argument of *Cpp17CopyAssignable*
			• The value of rv is otherwise unspecified

Concept `copy_constructible`:

- If `T` is not `const`;
  - If `T` is required to satisfy `destructible`;
    - `rv`'s resulting state is in the domain of `destructible`.
  - If `T` is required to satisfy `assignable_from` as the `LHS` argument;
    - `rv`'s resulting state is in the domain of the `lhs` argument of `assignable_from and,
    - If `T` is also required to satisfy `assignable_from` as the `RHS` argument;
      - `rv`'s resulting state is a *moved-from-value*, such that following a subsequent operation, `t = (T&&)(rv)`, where `t` and `rv` refer to the same object, `rv` still satisfies the requirements for the resulting state of `rv` in `move_constructible`.
  - Otherwise, the value of `rv` is not specified.

The concept `assignable_from` does not place any preconditions on the domain of the operation, therefore there is not a need to reference the *moved-from_value* requirements.

I find the lack of preconditions and postconditions in the concept requirements as confusing, and in this case. The lack of a precondition for `assignable_from` may be a defect.

- ~~If `rhs` is a non-`const` xvalue, the resulting state of the object to which it refers is valid but unspecified ([lib.types.movedfrom]).~~

- If `rhs` is a non-`const` xvalue;
  - If `RHS` is required to satisfy `destructible`;
    - the resulting state of the object to which `rhs` refers is in the domain of `destructible`.
  - If `RHS` is required to satisfy `assignable_from` as the `LHS` argument;
    - the resulting state of the object to which `rhs` is in the domain of the `lhs` argument of

`assignable_from`.

- Otherwise, the value of `rhs` is not specified.

Option 2

Option 2 requires that a moved-from object can be used as an rhs argument to move-assignment always and the result of self-move-assignment is unspecified.

Table 28: *Cpp17MoveConstructible* requirements

Expression	Assertion/note pre-/post-condition
`T u = rv;`	*Postconditions:* `u` is equivalent to the value of `rv` before the construction
`T(rv)`	*Postconditions:* `T(rv)` is equivalent to the value of `rv` before the construction

common	*Postconditions:*
	• If `T` meets the *Cpp17Destructible* requirements;
	  ◦ `rv` is in the domain of *Cpp17Destructible*
	• If `T` meets the *Cpp17MoveAssignable* requirements;
	  ◦ `rv` is in the domain of *Cpp17MoveAssignable*
	• If `T` meets the *Cpp17CopyAssignable* requirements;
	  ◦ `rv` is in the domain of the lhs argument of *Cpp17CopyAssignable*
	• The value of `rv` is otherwise unspecified

Table 28: *Cpp17MoveAssignable* requirements

Expression	Return type	Return value	Assertion/note pre-/post-condition

`t = rv`	`T&`	`t`	*Postconditions:*
			• If `t` and `rv` do not refer to the same object, `t` is equivalent to the value of `rv` before the assignment, otherwise the value of `t` is unspecified
			• `rv` is in the domain of *Cpp17MoveAssignable*
			• If `T` meets the *Cpp17Destructible* requirements;
			  ◦ `rv` is in the domain of *Cpp17Destructible*
			• If `rv` meets the *Cpp17CopyAssignable* requirements;
			  ◦ `rv` is in the domain of the lhs argument of *Cpp17CopyAssignable*
			• The value of `rv` is otherwise unspecified

Concept `copy_constructible`:

- If `T` is not `const`;
  - If `T` is required to satisfy `destructible`;
    - `rv`'s resulting state is in the domain of `destructible`.
  - If `T` is required to satisfy `assignable_from` as the `LHS` argument;
    - `rv`'s resulting state is in the domain of the `lhs` argument of `assignable_from and,
    - If `T` is also required to satisfy `assignable_from` as the `RHS` argument;
      - `rv`'s resulting state is in the domain of the `rhs` argument of `assignable_from`.
  - Otherwise, the value of `rv` is not specified.

The concept `assignable_from`:

- ~~If `rhs` is a non-`const` xvalue, the resulting state of the object to which it refers is valid but unspecified ([lib.types.movedfrom]).~~

- If `rhs` is a non-`const` xvalue;
  - If `RHS` is required to satisfy `destructible`;
    - the resulting state of the object to which `rhs` refers is in the domain of `destructible`.
  - If `RHS` is required to satisfy `assignable_from` as the `LHS` argument;
    - the resulting state of the object to which `rhs` is in the domain of the `lhs` argument of `assignable_from`.
  - Otherwise, the value of `rhs` is not specified.

Future Implications

Weaker Guarantees for Future Components

There are several cases in the existing standard where satisfying the guarantee of *valid but unspecified* has negative performance implications. For example, move operations on node based containers such as `std::list()` are not `noexcept` because some implementations require an allocation for a valid list. To avoid the allocation, the implementation would have to add additional null-checks for nearly all list operations.

Running a release build compiled with Visual C++ 2019[2] and given the following:

```
1  std::vector<std::list<int>> c{{9}, {8}, {7}, {6}, {5}, {4}, {3}, {2},
2  {1}, {0}};
   std::list<int> a;
```

A subsequent call `c.push_back(std::move(a));` generates 21 unnecessary calls to `new` and 20 unnecessary calls to `delete`. The additional call to `new` is to leave `a` in a valid but unspecified state. Sorting the same initial vector with

`std::sort(begin(c), end(c));`. Generates 9 calls to new and delete, all unnecessary.

Unfortunately, weakening the guarantees of an existing component is a breaking change but the committee should consider carefully before compromising new components in a similar fashion.

Note that a new component could still provide stronger guarantees than is required, for example a container may support `clear()`, `reset()`, or other operations that establish a new value.

Class invariants

Although the proposal *Support for contract based programming in C++*, P0542R5, did not include class invariants, it is possible that a future version of the standard will. At which time the likely correct decision would be that invariants are not checked on an object that has just been moved from by default, and more generally allow the declaration of *unsafe* operations where invariants are not checked.

References

Parent, Sean. *Move Annoyance*, Addison-Wesley, 31 Mar. 2021, https://sean-parent.stlab.cc/2021/03/31/move-annoyance.html. ↩

Sutter, Herb. “Move, Simply.” *Sutter’s Mill*, 21 Feb. 2020, https://herbsutter.com/2020/02/17/move-simply/.

Romer, Geoffrey. “Moved-from Objects Need Not Be Valid.” *C++ Standards Committee Papers*, ISO/IEC WG21, 10 Jan. 2020, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2027r0.pdf.

G. Dos Reis et. al. “Support for contract based programming in C++.” *C++ Standards Committee Papers*, ISO/IEC WG21, 08 June 2018, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html

Several links contained within the document. They will be listed here in a future draft along with any existing papers that come to my attention.

Acknowledgements

Thanks to Howard Hinnant, Herb Sutter, Jonathan Wakely, Nicolai Josuttis, Nicholas DeMarco, Eric Neibler, Dave Abrahams, and John Lakos for the many discussions and arguments that resulted in this paper.

1. Similar wording with the same intent appears in every version of the C++ Standard since C++11. ↩
2. Microsoft Visual C++ 2019 00435-60000-00000-AA859 ↩

Sean Parent
Sean Parent

sean.parent@stlab.cc

sean-parent

SeanParent

Too many projects, too little time.