# Relocation Has A Library Interface

Exploiting trivial relocatability in the Standard Library

|  |  |
|---|---|
| Document #: | P2967R0 |
| Date: | 2023-10-15 |
| Project: | Programming Language C++ |
| Audience: | Library Evolution Working Group |
| Reply-to: | Alisdair Meredith <br> <ameredith1@bloomberg.net> |

## Contents

## 1 Abstract

Paper [P2786R3] proposes the notion of trivial relocatability for the C++ Standard, and provides the minimal library interface necessary to detect and exploit that property; it deliberately defers analysis of how to apply trivial relocation throughout the standard library to this paper, which considers relocation more broadly, both trivial and non-trivial.

## 2 Revision history

### R0: October 2023 (pre-Kona mailing)

— Initial draft of this paper.

## 3 Introduction

The goal of this paper is to serve as an adjunct to [P2786R3], extending the Standard Library to provide better support for optimizing the use of trivially relocatable types. Paper [P2959R0] handles their interaction in the semantics and optimization of block-based containers. This paper provides the remaining library support for users to apply relocation to their own code.

## 4 Proposed Additions

Our primary proposal ([P2786R3]) is essentially a Core language facility, with the minimal library interface, one type trait and one function template with a special meaning to the translator. However, in practice we are likely to want to build user facing library facilities on top of this minimal feature set, to deliver our goal of using relocation in `std::vector`.

Here, we look at library extensions that would put relocation on the same level as move and copy.

### 4.1 `relocate`

We propose a new "convenience" function template:

```
template <class T>
   requires (is_trivially_relocatable_v<T> || is_nothrow_move_constructible_v)
         && !is_const_v<T>
constexpr
T* relocate(T* begin, T* end, T* new_location);
```

Which is equivalent to:

```
if constexpr (is_trivially_relocatable_v<T>) {
    trivially_relocate(begin, end, new_location);
}
else if (ranges-do-not-overlap) {
    std::uninitialized_move(begin, end, new_location);
    std::destroy(begin, end);
}
else {
    // move-and-destroy each member in the appropriate order
}
```

Note that this function supports overlapping ranges, just like `memmove`.

This function is similar to `uninitialized_relocate` in [P1144R8], except that our proposal requires pointers rather than input iterators for the source, and mandates we always trivially relocate types that support trivial relocation. The always-trivially-relocate-where-possible requires the input range be contiguous, but in principle we could relax this to using iterators that model the `contiguous_iterator` concept.

This function is also constrained to require no-throw move constructible types, as that better reflects its use case as an efficient relocation with minimal overhead. If an exception were thrown, the user would lack the information to put the program back into a good state, and the following `uninitialized_move_and_destroy` function is intended to support such use cases.

We do not have `uninitialized` in the name, as relocation already implies that we target range will be overwritten — but note that we *do* support overlapping ranges where some of the relocating objects are already initialized (and being overwritten) in the target range, which would therefore *not* be fully uninitialized.

### 4.2 `uninitialized_move_and_destroy`

We further propose a second "convenience" function template, that takes iterator ranges, supports potentially-throwing move constructors, but does *not* support overlapping input and output ranges:

```
template<class ForwardIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move_and_destroy(ForwardIterator first,
                                                        ForwardIterator last,
                                                        NoThrowForwardIterator result);
```

**Design note: the input sequence should probably require no-throw iterators, in order to guarantee the postcondition that all elements are destroyed, even when an exception is thrown.**

This function is directly inspired by `uninitialized_relocate` in [P1144R8]. However, as per its name in this proposal, it is mandated to *always* perform a move-construct followed by destruction and is not given permission to switch to a trivially relocating implementation for certain types. Implementations may still find ways to as-if such an implementation if they stare carefully at all the requirements, but we do not explicitly ban such implementations, we do not anticipate that as a worthwhile optimization.

Note that this function does not accept types that are not move constructible, even if they are trivially relocatable.

We do not support overlapping ranges in this function as, in general, it is undefined behavior to compare iterators into different sequences when trying to determine if there is an overlap, never mind the cost for non-random access iterators, and unsupportability of input iterators. Pointers are a special case as arbitrary (valid) pointers can be compared using `std::less<>`.

We provide a single iterator range signature to introduce this facility, but can imagine LEWG wanting to consider sentinels, std ranges support, and bounded output ranges rather than a single iterator to range-check the output.

Finally, we note that this function is not marked as `constexpr`, even though we know of no reason it could not be so marked, with the `constexpr`ness being implicitly dependent on the supplied iterators being `constexpr` iterators.

## 5 Proposed Wording

The following wording based on the latest working draft at the time this paper is written, [N4958].

**Editors' note: Uncompleted wording tasks:**

— complete specification for `uninitialized_move_and_destroy`

### REVIEW NOTE FOR LEWG

THIS WORDING IS EVOLVING AND INCOMPLETE, BUT WILL GIVE EWG A FLAVOR OF THE TEXT WE INTEND TO FINALIZE BEFORE SENDING TO LWG. IT WILL BE INTEGRATED INTO A SINGLE WORDING SECTION, RATHER THAN PER TOPIC, ONCE LEWG DETERMINES HOW MUCH OF THIS PAPER, IF ANY, SHOULD PROCEED.

### 5.1 Library extensions for trivial relocation

The following library extensions assume the adoption of some revision of [P2786R3].

#### 5.1.1 `relocate`

Add to the `<memory>` header synopsis in 20.2.2 [memory.syn]p3:

```
// 20.2.6, explicit lifetime management template<class T>
  T* start_lifetime_as(void* p) noexcept;                                 // freestanding
template<class T>
  const T* start_lifetime_as(const void* p) noexcept;                     // freestanding
template<class T>
  volatile T* start_lifetime_as(volatile void* p) noexcept;               // freestanding
template<class T>
  const volatile T* start_lifetime_as(const volatile void* p) noexcept;   // freestanding
template<class T>
```

```
  T* start_lifetime_as_array(void* p, size_t n) noexcept;                        // freestanding
template<class T>
  const T* start_lifetime_as_array(const void* p, size_t n) noexcept;            // freestanding
template<class T>
  volatile T* start_lifetime_as_array(volatile void* p, size_t n) noexcept;      // freestanding
template<class T>
  const volatile T* start_lifetime_as_array(const volatile void* p,
                                            size_t n) noexcept;                   // freestanding
```

```
template <class T>
   requires (is_trivially_relocatable_v<T> || is_nothrow_move_constructible_v)
         && !is_const_v<T>
constexpr
T* relocate(T* begin, T* end, T* new_location);                                  // freestanding
```

Append to Append to 20.2.6 [obj.lifetime]:

```
template <class T>
   requires (is_trivially_relocatable_v<T> || is_nothrow_move_constructible_v)
         && !is_const_v<T>
constexpr
T* relocate(T* begin, T* end, T* new_location);
```

x **Effects**: Equivalent to:

```
if constexpr (is_trivially_relocatable_v<T>) {
   return std::trivially_relocate(begin, end, new_location);
}
else if (less{}(end, new_location) || less{}(new_location + begin - end, begin)) {
   // No overlap
   uninitialized_move(begin, end, new_location);
   destroy(begin, end);
   return new_location;
}
else if (less{}(begin, new_location) {   // move-and-destroy each member, back to front
   while (T* dest = new_location + begin - end; dest != new_location) {
      ::new (--dest) T(std::move(*--end));
      destroy_at(end);
   }
   return dest;
}
else {                                   // move-and-destroy each member, front to back
   while (begin != end) {
      ::new (new_location++) T(std::move(*begin++));
      destroy_at(begin);
   }
   return new_location;
}
```

y *Throws*: Nothing.

### 5.2 `uninitialized_move_and_destroy` as a non-optimizing algorithm

`uninitialized_move_and_destroy` is directly inspired by `uninitialized_relocate` in [P1144R8] and the `uninitialized_move` family of algorithms in the standard. This functionality is entirely separable into a pure library proposal, as it does not rely on any of our language extension features. It is proposed purely for feature parity with [P1144R8].

The function name, `uninitialized_move_and_destroy`, is chosen to provide a clear hint at what the operation does. We might have preferred `uninitialized_move_construct_and_destroy` as more descriptive, but following the naming of `uninitialized_move`, we take `uninitialized` as sufficient information that the output range will be populated by move constructors, as there cannot be a live object as an alternative to assign to.

Compared to the proposed functions with `relocate` in their name, these overloads explicitly call the move constructor, and then the destructor of the source, so element types must support those operations, just as in [P1144R8]. They are specified as library algorithms using iterators rather than simple pointers. They also have a precondition excluding overlapping ranges.

We note that blanket wording for clause 27.11.1 [specialized.algorithms.general] guarantees all constructed objects will be destroyed if an exception is thrown, but we add a *Remarks* element to ensure that the source range honors its guarantee to destroy all elements in such cases as well.

The `uninitialized_move` family inspires the full range of overloads, for parallel algorithms, `std::ranges`, and [iterator, length) sequences.

Contrasting [P1144R8] and `uninitialized_move`, our design requires forward iterators for the input sequence, as we are modifying the source elements by moving out, and then destroying them.

### 5.2.1 `uninitialized_move_and_destroy` [uninitialized.move.and.destroy]

Add to the `<memory>` header synopsis in 20.2.2 [memory.syn]p3:

```
// 27.11, specialized algorithms
// 27.11.2, special memory concepts

// ...

template<class InputIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move(InputIterator first,                  // freestanding
                                          InputIterator last,
                                          NoThrowForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move(ExecutionPolicy&& exec,               // see 27.3.5
                                          ForwardIterator first, ForwardIterator last,
                                          NoThrowForwardIterator result);
template<class InputIterator, class Size, class NoThrowForwardIterator>
  pair<InputIterator, NoThrowForwardIterator>
    uninitialized_move_n(InputIterator first, Size n,                           // freestanding
                         NoThrowForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator, class Size,
         class NoThrowForwardIterator>
  pair<ForwardIterator, NoThrowForwardIterator>
    uninitialized_move_n(ExecutionPolicy&& exec,                                // see 27.3.5
                         ForwardIterator first, Size n, NoThrowForwardIterator result);
namespace ranges {
  template<class I, class O>
    using uninitialized_move_result = in_out_result<I, O>;                      // freestanding
  template<forward_iterator I, sentinel_for<I> S1,
           nothrow-forward-iterator O, nothrow-sentinel-for <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);             // freestanding
  template<forward_range IR, nothrow-forward-range OR>
```

```
    requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
      uninitialized_move_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
        uninitialized_move(IR&& in_range, OR&& out_range);                      // freestanding

  template<class I, class O>
    using uninitialized_move_n_result = in_out_result<I, O>;                    // freestanding
  template<forward_iterator I,
           nothrow-forward-iterator O, nothrow-sentinel-for <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);                                // freestanding
}
```

```
template<class ForwardIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move_and_destroy(ForwardIterator first,    // freestanding
                                                      ForwardIterator last,
                                                      NoThrowForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move_and_destroy(ExecutionPolicy&& exec,   // see 27.3.5
                                                      ForwardIterator first, ForwardIterator last,
                                                      NoThrowForwardIterator result);
template<class ForwardIterator, class Size, class NoThrowForwardIterator>
  pair<ForwardIterator, NoThrowForwardIterator>
    uninitialized_move_and_destroy_n(ForwardIterator first, Size n,             // freestanding
                                     NoThrowForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator, class Size,
         class NoThrowForwardIterator>
  pair<ForwardIterator, NoThrowForwardIterator>
    uninitialized_move_and_destroy_n(ExecutionPolicy&& exec,                    // see 27.3.5
                                     ForwardIterator first, Size n, NoThrowForwardIterator result);

namespace ranges {
  template<class I, class O>
    using uninitialized_move_and_destroy_result = in_out_result<I, O>;          // freestanding
  template<forward_iterator I, sentinel_for<I> S1,
           nothrow-forward-iterator O, nothrow-sentinel-for <O> S2>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
      uninitialized_move_and_destroy_result<I, O>
        uninitialized_move_and_destroy(I ifirst, S1 ilast, O ofirst, S2 olast); // freestanding
  template<forward_range IR, nothrow-forward-range OR>
    requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
      uninitialized_move_and_destroy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
        uninitialized_move_and_destroy(IR&& in_range, OR&& out_range);          // freestanding

  template<class I, class O>
    using uninitialized_move_and_destroy_n_result = in_out_result<I, O>;        // freestanding
  template<input_iterator I,
           nothrow-forward-iterator O, nothrow-sentinel-for <O> S>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
      uninitialized_move_and_destroy_n_result<I, O>
        uninitialized_move_and_destroy_n(I ifirst, iter_difference_t<I> n,
                                         O ofirst, S olast);                    // freestanding
}
```

Add a new subclause between 27.11.6 [uninitialized.move] and 27.11.7 [uninitialized.fill]:

**`uninitialized_move_and_destroy` [uninitialized.move.and.destroy]**

```
template<class ForwardIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move_and_destroy(ForwardIterator first,    // freestanding
                                                      ForwardIterator last,
                                                      NoThrowForwardIterator result);
```

1 *Preconditions*: `destination` is not in the range [`first`, `last`).

*Effects*: Equivalent to:

```
for (; first != last; ++destination, (void)++first) {
   ::new (voidify(*destination)) iter_value_t<NoThrowForwardIterator>(*first);
   destroy_at(addressof(*first));
}
return destination;
```

2 *Throws*: Nothing, unless an exception is thrown by a move constructor.

3 *Remarks*: If an exception is thrown, all objects in both the source and destination ranges are destroyed.

```
namespace ranges {
  template<forward_iterator I, sentinel_for<I> S1,
           nothrow-forward-iterator O, nothrow-sentinel-for <O> S2>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_and_destroy<I, O>
      uninitialized_move_and_destroy(I ifirst, S1 ilast, O ofirst, S2 olast);   // freestanding
}
```

```
namespace ranges {
  template<forward_iterator IR, nothrow-forward-range OR>
    requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
    uninitialized_move_and_destroy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
      uninitialized_move_and_destroy(IR&& in_range, OR&& out_range);            // freestanding
}
```

```
template<class ForwardIterator, class Size, class NoThrowForwardIterator>
  pair<ForwardIterator, NoThrowForwardIterator>
    uninitialized_move_and_destroy_n(ForwardIterator first, Size n,             // freestanding
                                     NoThrowForwardIterator result);
```

```
namespace ranges {
  template<forward_iterator I,
           nothrow-forward-iterator O, nothrow-sentinel-for <O> S>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_and_destroy_n_result<I, O>
      uninitialized_move_and_destroy_n(I ifirst, iter_difference_t<I> n,
                                       O ofirst, S olast);                      // freestanding
}
```

**Editors' note: Preconditions and throws clause implicit from Effects:, but stated for clarity while in Evolutionary groups.**

## 6 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Thanks to Arthur O'Dwyer for the inspiration for an interface that goes beyond just a `relocate` function.

## 7 References

[N4958] Thomas Köppe. 2023-08-14. Working Draft, Programming Languages — C++.
https://wg21.link/n4958

[P1144R8] Arthur O'Dwyer. 2023-05-14. std::is_trivially_relocatable.
https://wg21.link/p1144r8

[P2786R3] Mungo Gill, Alisdair Meredith. 2023-10-15. Trivial Relocatability For C++26.
https://wg21.link/p2786r3

[P2959R0] Alisdair Meredith. 2023-10-15. Relocation Within Containers.
https://wg21.link/p2959r0