Relocation Has A Library Interface

Exploiting trivial relocatability in the Standard Library

Document #:	P2967R1
Date:	2024-05-22
Project:	Programming Language C++
Audience:	Library Evolution Working Group

Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>
	Mungo Gill <mgill83@bloomberg.net>

Contents

1 Abstract

Paper [P2786] proposes the notion of trivial relocatability for the C++ Standard, and provides the minimal library interface necessary to detect and exploit that property. This paper is an adjunct to [P2786], extending the Standard Library to provide better support for relocating objects, both trivially and non-trivially.

2 Revision history

R1: May 2024 (pre-St Louis mailing)

— Simplified the Abstract
— Renamed library function to `uninitized_relocate`
— Using iterator concepts to overload signatures with different capabilities
— Apply `noexcept` to overloads that cannot fail
— Removed support for overlapping ranges when iterators are not contiguous
— Added a prototype implementation in Appendix A
— Delegated optimizing `std::swap` to [P3239]

R0: October 2023 (pre-Kona mailing)

— Initial draft of this paper.

3 Introduction

The goal of this paper is to serve is to introduce relocation as a first class facility in C++. We build on [P2786] that introduces the notion of *trivial relocatability*, and extend the notion of relation to types that provide a publicly accessible API for relocation to be used when trivial relocation is not supported. For the first draft of this paper, that public API is providing a publicly accessible move constructor, and a publicly accessible destructor. Future proposals, including language proposals, may extend the support for non-trivial relocation further.

4 Motivating Use Cases

4.1 A Simpler Interface for Non-Library Writers

[P2786] Proposed a low-level *magic function* `trivially_relocate` which was guaranteed to use copying of object representations for trivially relocatable types but which could not be used otherwise. This is exactly what is required for writers of low-level generic algorithms who need to know exactly what functions they call will do.

On the other hand, a user who simply wants to relocate a range of objects, without caring too much about the details, wants a higher level interface:

```
template<typename T>
MyOwnNonAggregateArray::MyOwnNonAggregateArray(MyOwnNonAggregateArray &&other) {
    size = other.size;
    uninitialized_relocate(other.begin(), other.end(), data);
    other.clearWithoutCallingDestructors();
}
```

5 Proposed Library Changes

5.1 A new function `std::uninitialized_relocate`

We are proposing a new "convenience" function, `std::uninitialized_relocate` that has a number of overloads taking variously specified input ranges and a destination.

```
namespace std {
template <typename InIter, typename OutIter>
    auto uninitialized_relocate(InIter first, InIter last, OutIter to)
    /*noexcept - see below*/ -> OutIter;
}
```

This function was inspired by and modelled upon the existing `uninitialized_move` and `uninitialized_copy` functions in the standard library, which was the reason we opted for the name `uninitialized_relocate` rather than a simple `relocate`.

This function will be constrained to only permit lvalue reference iterators, disallowing proxy iterators.

It is declared noexcept when the relocation is guaranteed to not fail, ie when the type `T = iter_value_t<InIter>` (which is required to match `iter_value_t<OutIter>`) is either trivially relocatable or nothrow move constructible.

This function is guaranteed to perform trivial relocation where it is permitted to do so (ie if `T` is trivially relocatable), otherwise using either move or copy construction as appropriate. It is guaranteed that this will *not* use the assignment operator for such relocation.

Overlapping source and destination ranges are supported only if both `InIter` and `OutIter` are contiguous iterators and the types `iter_value_t<I>` and `iter_value_t<O>` are the same and either nothrow move constructible or trivially relocatable. Attempting to use this function with overlapping source and destination

ranges when not supported will result in unspecified behavior (for consistency with `uninitialized_move` and `uninitialized_copy`)

For types that are copy constructible, but neither trivially relocatable not nothrow move constructible, we support the strong exception safety guarantee. If an exception is thrown during the initialization, objects in the source range are unmodified and any objects already constructed in the destination range are destroyed in an unspecified order.

For types that are move constructible, but neither copy constructible, trivially relocatable, nor nothrow move constructible, we support the basic exception safety guarantee. If an exception is thrown during the initialization, objects in the source range are left in a valid but unspecified state, and the objects already constructed in the destination range are destroyed in an unspecified order.

5.2 A new function `ranges::uninitialized_relocate`

We are also proposing overloads of a new "convenience" function, `ranges::uninitialized_relocate` that has a number of overloads taking variously specified input ranges and a destination.

```
namespace ranges {
template <typename InIter, typename OutIter>
    auto uninitialized_relocate(InIter ifirst, InIter ilast, OutIter ito, OutIter ilast)
    /*noexcept - details to follow*/
        -> OutIter;

template<typename InRange, typename OutRange>
    auto uninitialized_relocate(InRange&& in_range, OutRange&& out_range)
    /*noexcept - details to follow*/
        -> uninitialized_relocate_result< borrowed_iterator_t<InRange>
                                        , borrowed_iterator_t<OutRange> >;
}
```

This function was inspired by and modelled upon the existing `ranges::uninitialized_move` and `ranges::uninitialized_copy` functions in the standard library, which was the reason we opted for the name `uninitialized_relocate` rather than a simple `relocate`.

We fully intend that this will provide similar type support and safety guarantees to `std::uninitialized_relocate` above, but the full wording and sample implementations for these will not be available until R2 of this paper.

6 FAQ

6.1 Is this paper a full proposal

No, although it is close. There are multiple choices of design and semantics made throughout this paper, and it is very much a first draft of how to build a general purpose relocation facility on top of [P2786]. The API presented is implemented and used in a `vector` implementation in Corentin's branch of Clang.

6.2 What *is* relocation?

Moving an object into a different memory location, by any valid means at our disposal. A relocated object should be a drop-in replacement for the original object, there should be no semantically meaningful difference — this is a stronger guarantee than simply moving a *value*.

For clarification and for the avoidance of any doubt, our definition of relocation does *not* permit the use of the assignment operator, and the assignment operator is never considered to be a valid means.

6.3 Why do we insist that all iterator reference types are lvalue references?

Relocate is a primitive operation on lvalues. We cannot guarantee to extact an lvalue from a proxy iterator, and we can support input sequences when the element-wise relocate is a no-fail operation. Hence, we use C++20 iterator concepts and further constrain to require iterators to return only lvalues.

6.4 Should library definition of relocation support *all* trivially relocatable types?

We might restrict the library relocate functionality to only types that support the public API. In such cases, trivial relocation would be an optimization within the more limited domain, and users would need to call the sharp tool of trivial relocation directly if they wanted to support such types.

We prefer to support both type categories, as the proposed `uninitializes_relocate` function template supports iterators that are not limited to pointers.

6.5 What happens if relocation fails?

Relocation cannot be a *partial* operation, we must provide the basic guarantee. Following the design of `vector` we choose the strong exception safety guarantee, but this may be deemed to expensive and we would prefer users to implement strong exception safety themselves if needed — the cost of copying rather than moving is expected to be high.

No-fail relocation can be identified and signalled through `noexcept` on appropriate overloads, by conforming to the neo-Lakos rule that permits `noexcept` where it is expected to be queries, even if a function has a narrow contract.

6.6 How can we support potentially-throwing contiguous iterators?

If the iterators are contiguous, we can take the address of the lead element of both sequences and work with plain pointers. Plain pointers do not throw, and we are not guaranteeing to call the iterator operations directly — side effects of iterator operations are not guaranteed.

6.7 How can we use this API if we cannot detect no-throw iterators

There is a general concern any time a concept has semantic guarantees, in particular when we look to use it to implement other library functionality that lacks those guarantees. In this case, to preserve the correct no-fail behavior of the proposed relocate facility, we must rely on no-throw iterators. These are not a first-class library concept, being specified for documentation only. We may have to step up those guarantees, in a manner similar disambiguating iterators for `vector::insert`, to say that it is unspecified how far a library will look to determine

that an iterator operation never throws, but we do guarantee no standard library iterator throws, and standard library iterator adapters are non-throwing if they adapt a no-throw iterator.

6.8 With many arbitrary choices of semantics, should we support multiple

functions?

We are deliberately rolling a primitive `relocate` facility with a single name. While nothing precludes adding similar functionality with different constraints and guarantees, we prefer to keep a single name. Note that this does mean that the API promises different capabilities depending on the kinds of iterators that are passed.

6.9 Should we support the ranges library?

Yes. All the other `uninitialized_*` algorithms have ranges overloads, and this should be no exception. However, the specification in this document is primitive and should be refined with LEWG feedback.

6.10 Why not just `relocate`

For this initial proposal we prefer to retain the name *`uninitialized`*`_relocate` so that we do not occupy a more fundamental name that may be wanted for future language extensions.

6.11 Should we provide support for single-object relocation?

See alternative designs, but again, we deem such usage to be intruding into an area of active interest for lower level language support.

6.12 What is the deal with transparent replacement?

This is something we need to spell out carefully. Thinking of an example with an iterator adaptor that exposes a reference to a data member of a complete type, and how that would interact with this design.

6.13 Why does the proposed API use concepts rather than constraints

It is much simpler to specify the proposed API with the different semantics of each overload. Moving all of the specification into a single function with constraints and extensive *Remarks:* to cover the `noexcept` specification, overlapping ranges guarantees, exception handling, etc. would lead to a more awkward text.

That said, we are very happy to redraft in that direction if that is what the L(E)WG would prefer. For our initial presentation of ideas though, we are deliberately choosing the clearer (non-conventional) form.

6.14 Should we support sentinels?

Going with the concept form of function template declaration, we could generalize the specification to accept sentinels to terminate the input range. We chose not to do so simply as the lead author does not have much practice with sentinel APIs to guarantee a correct prototype within the time constraints on this paper. In addition, it would be the first use of sentinels outside the `std::rangaes` namespace, so we were happy to step back from an initial specification that supported them.

7 Alternative Designs

7.1 R0 of this paper

The original design for this API came in two parts. A `relocate` function that accepted pointer arguments for a no-fail function (stronger than `noexcept`) that would relocate objects in memory using either trivial relocation, or a non-throwing move constructor followed by destruction. This function was no-fail so that it would not be burdened with recovery code for exceptions; it would *mandate* support only for types with non-throwing move-construct and destroy operations, or that were trivially relocatable.

A second API would provide algorithms consistent with the `uninitialized_move` function templates in the Standard Library. These functions would accept a much wider variety of ranges and potentially-throwing failure modes, providing appropriate guarantees to clean up resources in event of an exception — consistent with the `uninitialized_move` family of functions itself.

These `uninitialized_move_and_destroy` functions would explicitly call the move constructor followed by the destructor, and hence constrained that those operations be eligible. Implementations would then have the QoI ability to render each move-and-destroy operation as a call to `trivially_relocate` for trivially relocatable types, under the traditional as-if rule.

We opted against this design as having two names and two semantics for what is essentially the same operation — relocate — seemed more confusing than helpful.

7.2 Relocate a single object

A function to relocate a single object, rather than a whole range, seems a tempting addition to our library — especially as it can easily be specified with a single *Effects:* as if … library specification element.

However, relocating single objects is often a cause for concern. If it is too easy to relocate a local variable, for example, we will be encouraging code that hits undefined behavior at the end of block scope by destroying an object that no longer exists. Now there are reasonable idioms where relocating an object and then replacing it before leaving scope (even by a propagating exception) can be relied upon, but we are beyond the domain where the compiler can help us easily catch and fix our bugs. If someone is prepared to write the more awkward relocate-an-array-of-one-object call, that frustration will also often trigger their memory to check for bad control paths that do not replace the object.

In practice, we believe the idea of destructive move of local variables, such as passing as arguments into a function call, is going to be a useful feature of a future C++. However, we believe such support should be a language feature and not a library feature, so that the compiler can properly reason about object lifetimes and act accordingly.

For example, see [P2839R0] for one possible direction.

7.3 Add further support to optimize `std::swap`

The API in *this* paper is competing with [P1144] that also provides support for optimizing `std::swap` for trivially relocatable types. We delegate the extensive technical discussion on such a design to [P3239]. We are happy to merge future revisions of these papers if that is the view expressed by LEWG.

8 Appendix A: Implementation Prototype

Note that all names are fully qualified as `std::` and `std::ranges::`, per 16.4.6.4 [global.functions]p4.

```
#include <algorithm>
#include <cassert>
#include <iterator>
#include <memory>
#include <type_traits>

// Forward looking for asserting preconditions --- tune feature macro as needed
#ifndef __cpp_contracts
# define contract_assert(...) assert(__VA_ARGS__)
#endif

namespace {
// Implementation details --- using an anonymous namespace within this single TU prototype

template <class I>
concept nothrow_input_iterator = std::input_iterator<I>;        // exposition only

template <class I>
concept nothrow_forward_iterator = std::forward_iterator<I>;    // exposition only

// We need a concept to distinguish the "not" case at the end --- subsumption
// does not kick in, so could maybe be just a constexpr boolean variable
// template.

template <class I, class O>
concept relocatable_iterators                                   // exposition only
      = std::same_as<std::iter_value_t<I>, std::iter_value_t<O>>
    and std::is_lvalue_reference_v<std::iter_reference_t<I>>
    and std::is_lvalue_reference_v<std::iter_reference_t<O>>;


template <class I, class O>
concept no_fail_relocatable_iterators                           // exposition only
      = relocatable_iterators<I, O>
    and ( std::is_nothrow_move_constructible_v<std::iter_value_t<I>>
     or std::is_trivially_relocatable_v<std::iter_value_t<I>>
      );
} // unnamed namespace

namespace std {

template <std::contiguous_iterator I, std::contiguous_iterator O>
  requires no_fail_relocatable_iterators<I, O>
auto unintialized_relocate(I first, I last, O to) noexcept
  -> O {
  // Note that by design, none of the operations in this implementation are
  // potentially-throwing --- hence there are no concerns about exception
  // safety

  // One implementation bug deferred from proof of concept is that we assume
  // that the object pointer type can be implicitly converted to the
```

```
// contiguous output iterator when creating the return value. I have yet to
// confirm that this is mandated by the relevant concepts
// Using pointers to relocate whole range at once, and remove
// no-throw-iterator concerns from support for contiguous iterators. There
// remains corner-case concerns to clean up for throwing `operator*` and
// dereferencing past-the-end iterators for empty ranges.
auto * begin = std::addressof(*first);
auto * end = begin + (last - first);
auto * new_location = std::addressof(*to);

// This check is redundant if we follow all the branches below, but makes
// the design intent clear.
if (begin == new_location) {
  return to;
}

// For trivially relocatable type, just delegate to the core language primitive
if constexpr (std::is_trivially_relocatable_v<std::iter_value_t<I>>) {
  (void) std::trivially_relocate(begin, end, new_location);
  return to + (last - first);
}

// For non-trivial relocation, we must detect and support overlapping ranges
// ourselves. At this point we no longer need to worry about trivial
// relocatable types but are not move-constructible.

if (less{}(end, new_location) || less{}(new_location + (end - begin), begin)) {
  // No overlap
  O result = std::uninitialized_move(first, last, to);
  destroy(begin, end);
  return result;
}
else if (less{}(begin, new_location)) {
  // move-and-destroy each member, back to front

  // Implementation note: we could just modify `new_location` rather than
  // create the local `dest`, but want to clearly show the algorithm
  // without micro-optimizing.
  auto * dest = new_location + (end - begin);
  while (dest != new_location) {
    dest = std::construct_at(--dest, std::move(*--end));
    std::destroy_at(end);
  }
  return to + (last - first);

}
else if (begin == new_location) {
  // Note that this is the same check redundantly hoisted out of the
  // if/elif/else flow, but is NOT redundant here
  return to;
}
else {
  // move-and-destroy each member, front to back
```

```
    while (first != last) {
      (void) std::construct_at(std::addressof(*to), std::move(*first));
      destroy_at(std::addressof(*first));
      ++to;
      ++first;
      }
    return to;
  }

  std::unreachable();
}

template <nothrow_input_iterator I, nothrow_input_iterator O>
  requires no_fail_relocatable_iterators<I, O>
auto unintialized_relocate(I first, I last, O to) noexcept
  -> O {
  // There is no support for overlapping ranges unless both iterator types are
  // contiguous iterators, which would be subsumed by the contiguous itetator
  // overload.

  // Note that there are trivially relocatable types that are not no-throw
  // move constructible and vice-versa, so we need to handle both cases. In
  // doing so, we should pick a preferred implementation for types that are
  // both trivially relocatable and no-throw move constructible.
  if (first == last) {
    return to;
  }

  // Think carefully about iterator invalidation when destroying elements.
  // The post-increment on `first` seems important, in case element
  // destruction invalidates an iterator, which would be the case for a
  // pointer.
  while (first != last) {
    auto * begin = std::addressof(*first++);
    auto * new_location = std::addressof(*to++);

    if constexpr(std::is_trivially_relocatable_v<std::iter_value_t<I>>) {
      (void) std::trivially_relocate(begin, begin + 1, new_location);
    }
    else {
      (void) std::construct_at(new_location, std::move(*begin));
      std::destroy_at(begin);
    }
  }

  return to;
}


template <std::forward_iterator I, nothrow_input_iterator O>
  requires relocatable_iterators<I, O>
       and (!no_fail_relocatable_iterators<I, O>)
       and std::is_move_constructible_v<std::iter_value_t<I>>
```

```
auto unintialized_relocate(I first, I last, O to) // NOT declared noexcept
  -> O {
  // The move-constructor can throw, so relocation is not guaranteed to
  // succeed. The opt for the vector-like strong exception safety guarantee
  // for this operation: if move can throw, but the type is
  // copy-constructible, then *copy* the range, so that we can safely unwind
  // if an exception is thrown without leaving the initially relocated
  // elements in an unknown state. Otherwise, we will *move* all of the
  // elements, and leave those elements in an unspecified valid state. Only
  // if the copy/move operation succeeds are the original elements destroyed.
  if (first == last) {
    return to;
  }

  if constexpr(std::contiguous_iterator<I> and std::contiguous_iterator<O>) {
    // Note that once we complete overload resolution, all viable iterators
    // referring to trivially relocatable types should have directed to an
    // overload above. We might still have contiguous iterators to
    // move-constructible types that may throw, and are not trivially
    // relocatable, which means we may be able to assert preconditions that
    // overlapping ranges are not supported.
    // check for overlapping range with a contract_assert
#if defined(__clang__)
    contract_assert(!__is_pointer_in_range(std::addressof(*first), std::addressof(*last),
                                           std::addressof(*to)));
#else
    const size_t count = last - first;
    auto * begin = std::addressof(*first);
    auto * end  = begin + count; // `last` is often not a derefencerable iterator
    auto * new_location = std::addressof(*to);
    auto less = std::less<>{};
    // Note that end == new_location is fine
    contract_assert(less(end, new_location) || less(new_location + count, begin));
#endif
  }

  // Remember original `to` may be invalidated by this operation, so we need
  // to capture the return value.
  if constexpr(std::is_copy_constructible_v<std::iter_value_t<I>>) {
    to = std::uninitialized_copy(first, last, to); // cleans up new range if any copy throws
  }
  else {
    to = std::uninitialized_move(first, last, to); // cleans up new range if any move throws
  }

  std::destroy(first, last); // "Library UB" if destructor throws
  return to;
}

} // std

namespace std::ranges {
```

```
template <class S, class I>
concept nothrow_sentinel_for = std::sentinel_for<S, I>;

template<class I, class O>
using uninitialized_relocate_result = std::ranges::in_out_result<I, O>;

template<nothrow_forward_iterator I, nothrow_sentinel_for<I> S1,
         nothrow_forward_iterator O, nothrow_sentinel_for<O> S2>
  requires std::constructible_from<std::iter_value_t<O>, std::iter_reference_t<I>>
auto uninitialized_relocate(I ifirst, S1 ilast, O ofirst, S2 olast)
  -> uninitialized_relocate_result<I, O> {
  auto result = std::ranges::uninitialized_move(ifirst, ilast, ofirst, olast);
  auto laundered = std::ranges::destroy(ifirst, result.in);
  return { laundered, result.out };
}

template <class R>
concept nothrow_forward_range = std::ranges::forward_range<R>;

template<nothrow_forward_range IR, nothrow_forward_range OR>
  requires std::constructible_from< std::ranges::range_value_t<OR>
                                  , std::ranges::range_reference_t<IR>>
auto uninitialized_relocate(IR&& in_range, OR&& out_range)
  -> uninitialized_relocate_result< std::ranges::borrowed_iterator_t<IR>
                                  , std::ranges::borrowed_iterator_t<OR>
                                  > {

  auto result = std::ranges::uninitialized_move(in_range, out_range);
  auto laundered = std::ranges::destroy( std::ranges::borrowed_iterator_t<IR>(begin(in_range))
                                       , result.in );

  return { laundered, result.out };
}

} // std::ranges


int main() {
  // Test driver goes here
  int source[5] {};
  int dest[5];

  std::uninitialized_relocate(std::begin(source), std::end(source), std::begin(dest));

  std::ranges::uninitialized_relocate(dest, source);
}
```

9 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.

Thanks to Marshall Clow for catching many bugs in the original prototype, all remaining bugs are purely of my own making.

10 References

[P1144] Arthur O'Dwyer. std::is_trivially_relocatable.
https://wg21.link/p1144

[P2786] Mungo Gill, Alisdair Meredith. Trivial Relocatability For C++26.
https://wg21.link/p2786

[P2839R0] Brian Bi, Joshua Berne. 2023-05-15. Nontrivial relocation via a new "owning reference" type.
https://wg21.link/p2839r0

[P3239] Alisdair Meredith. A Relocating Swap.
https://wg21.link/d3239r0