P3516R1: Uninitialized algorithms for relocation

Louis Dionne
Giuseppe D'Angelo

LEWG — WG21 Hagenberg, Feb 2025

P3516 in a nutshell

- We propose a new family of “specialized memory algorithms”: `std::uninitialized_relocate` (+ `_n`, `_backwards` variants)

- These algorithms *relocate objects* from a source range into uninitialized storage

- Goal is to be used by containers to move elements around in memory
  - Unify the code paths for relocation via move+destroy and/or via trivial relocation

P3516 in a nutshell

```
template <typename FwdIt1, typename FwdIt2>
constexpr FwdIt2
std::uninitialized_relocate(FwdIt1 first, FwdIt1 last, FwdIt2 result)
```

- Pre:
  a. `[first, last)` is a valid range with live objects;
  b. `result` is the beginning of a range over uninitialized storage

- Post:
  a. objects in `[first, last)` have been destroyed;
  b. `[result, result + (last-first))` contains live objects
     Each one has been *relocated* from the corresponding object in the source

Use case 1: vector insertion

```
/* Status quo */
vector<T>::emplace(iterator position,
                   Args&&... args) {
if (size() == capacity()) { /* ... */ }
if (position == end()) { /* ... */ }
} else {
  T tmp(std::forward<Args>(args)...);

  std::construct_at(std::to_address(end()),
                    std::move(back()));
  ++end_;

  std::move_backward(
      position, end() - 2, end() - 1);

  *position = std::move(tmp);
}
return position;
}
```

```
/* This paper */
vector<T>::emplace(iterator position,
                   Args&&... args) {
if (size() == capacity()) { /* ... */ }
if (position == end()) { /* ... */ }
} else {
  T tmp(std::forward<Args>(args)...);

  std::uninitialized_relocate_backward(
                position, end(), end() + 1);

  std::construct_at(std::to_address(position),
                    std::move(tmp));

  ++end_;
}
return position;
}
```

Use case 2: vector erasure

```
/* Status quo */
constexpr iterator
vector<T>::erase(iterator first,
                 iterator last)
{
  if (first == last)
    return last;

  auto new_end = std::move(last, end(),
                           first);
  std::destroy(new_end, end());

  end_ -= (last - first);
  return first;
}
```

```
/* This paper */
constexpr iterator
vector<T>::erase(iterator first,
                 iterator last)
{
  if (first == last)
    return last;

  std::destroy(first, last);
  std::uninitialized_relocate(last, end(),
                              first);

  end_ -= (last - first);
  return first;
}
```

Use case 3: vector reallocation

```
/* Status quo */

template <class ...Args>
constexpr reference
vector<T>::emplace_back(Args&& ...args) {
  if (size() < capacity()) { ... }

  vector<T> tmp;
  tmp.reserve((size() + 1) * 2);
  std::construct_at(tmp.begin_ + size(), std::forward<Args>(args)...);
  // ... guard destruction of the new element ...

  for (auto& element : *this)
    tmp.emplace_back(std::move_if_noexcept(element));


  // ... disengage guard ...
  ++tmp.end_;

  swap(tmp);
  return back();
}
```

Use case 3: vector reallocation

```
/* Status quo */

// same as before,
// reserve, construct element, ...




for (auto& element : *this)
  tmp.emplace_back(
    std::move_if_noexcept(element)
  );


// ... disengage guard ...
++tmp.end_;
swap(tmp);
return back();
```

```
/* This paper */

// same as before,
// reserve, construct element, ...

if constexpr (is_nothrow_relocatable_v<T>) {
  tmp.end_ = std::uninitialized_relocate(begin(),
                                         end(), tmp.begin_);
  end_ = begin_;
} else {
  for (auto& element : *this)
    tmp.emplace_back(
      std::move_if_noexcept(element)
    );
}

// ... disengage guard ...
++tmp.end_;
swap(tmp);
return back();
```

Details: how does relocation happen?

- All the algorithms are implemented as loops around the exposition-only *relocate-at* function template:

```
template<class T>
constexpr T* relocate-at(T* dest, T* source)
```

- This function relocates 1 element from `source` to `dest`
  - Via trivial relocation (P2786) if available
  - Otherwise via move+destroy

relocate-at does a lot of heavy lifting

- Tackles 2 axis
  - TR vs. move+destroy
  - constant evaluation vs. runtime
- Allows writing streamlined code that relocates objects
- Necessary to implement basic functionality like `vector<T>::emplace`

Life without *relocate-at*

```
template<class T>
constexpr T* relocate-at(T* dest, T* source) {
  if constexpr (is_trivially_relocatable_v<T> && is_move_constructible_v<T>) {
    if consteval {
      return relocate-via-move-and-destroy(dest, source);
    } else {
      return trivially_relocate(source, source + 1, dest);
    }
  } else if constexpr (is_trivially_relocatable_v<T>) {
    return trivially_relocate(source, source + 1, dest);
  } else {
    return relocate-via-move-and-destroy(dest, source);
  }
}
```

Exception handling

- If an exception is thrown during a call to std::uninitialized_relocate:
  - all elements of *both* the source and destination ranges are destroyed;
  - the exception is thrown again.
- Destroying all the objects constructed is common with the other uninitialized_* algorithms
- Here we also need to fully destroy the source range, otherwise the program state is unrecoverable
  - Identical conclusions as other relocation papers (cf. P1144)
- Still fulfills the contract of the intended use cases:
  - Vector reallocation won't use relocation *anyhow*, if possibly throwing
  - Vector erase/insert in the middle only has the basic guarantee, losing the tail is OK

FAQ: Why taking iterators/ranges and not pointers?

- Because it's more useful: implementation of container functions that call the relocation algorithms is usually based on iterators
  - Even if the container is contiguous, like std::vector
  - Louis has done work in libc++

- Lowering iterators to pointers is possible for contiguous containers, but one loses information, and generally wants to avoid doing so
  - Debug/hardened iterators are a thing

- Consistency with the existing specialized memory algorithms, specified in terms of iterators/ranges

FAQ: What about allocator support?

- Pre-existing: none of the specialized memory algorithms has allocator support
- Some implementations (e.g. libstdc++) have private allocator-aware versions
- The proposed algorithms are still perfectly usable usable by
  - Allocator-unaware containers (`inplace_vector`; containers in Qt / 3rd parties)
  - Allocator-aware containers if the allocator is known not to specialize construct/destroy (e.g. `std::allocator`) or does not keep track of the constructed items (e.g. `polymorphic_allocator`)
- `std::trivially_relocate()` has the same “issue” (no support for allocators)
  - See also P3585R0 (`allocator_traits::is_internally_relocatable`)
  - Such a trait can be used to enable the uninitialized_relocate algorithms! And not just TR

FAQ: Why forward and backward variants?

- Relocation in overlapping ranges requires careful ordering of the operations
- Relocating “to the right” (destination range starts from inside the source range) requires relocating from the tails, backward
- Not a novelty:
  - `std::copy, std::copy_backward`
  - `std::move, std::move_backward`
- Therefore we also propose `std::uninitialized_relocate_backward`
- Useful in std::vector::insert (in order to “create a window” where to insert)

FAQ: Why supporting a throwing relocation?

- A *language* notion of “relocating constructor” is likely going to be nothrow
  - At least, all papers in the area so far seem to agree on this
- However, as a *library* notion, relocation needs to be backwards-compatible in order to be useful

FAQ: Why supporting a throwing relocation? (Cont.)

- In order to relocate an object:
  - If possible, use trivial relocation (C++26)
  - (Otherwise, if possible, use language relocation (C++-next))
  - Otherwise, use move+destroy (C++11)
    - Which may actually mean copy+destroy (C++98)
- Some of these may throw! And there's nothing wrong with that. Code that cares about a possible throwing relocate already has provisions in place.
  - E.g. std::vector reallocation always copies in that case
- This is similar to throwing moves: containers and algorithms in the library must support them, even if "we don't like them"
- P2786 indeed added std::is_nothrow_relocatable

FAQ: Does uninitialized_relocate really need to relocate one element at a time?

- No, that's just the specification!

- Any implementation worth its salt is going to special-case the performance sensitive paths anyhow
  - Just like in the existing uninitialized_* algorithms, or in std::copy, std::move, std::rotate, etc.

- Within the algorithms, we can detect if source/destination ranges are contiguous, and then:
  - if the value_type is trivial (e.g. int) => use "bulk" memmove()
  - If the value_type is TR (e.g. unique_ptr<T>) => use "bulk" std::trivially_relocate()

FAQ: Why specifying so many algorithms, when only a few seem to have direct use cases?

- Consistency with the existing memory algorithms
  - Which already have many variations: default construct, value construct, copy, move, fill
  - Which already come in parallel and range versions
    - That’s why we provided these as well
- Library consistency ought to trump the “perceived” usefulness

Suggested Polls

1. Do we want “higher level” relocation facilities that work with types other than trivially relocatable types for C++26?

2. Do we want such relocation facilities to work with iterators/ranges or pointers?

3. Do we want such relocation facilities to allow potentially-throwing moves?

4. Should we expose relocate-at as a std::relocate_at public function?