A Relocating Swap

A memory-based swap function that does not end object lifetimes

Document #:	D3239R0
Date:	2024-05-01
Project:	Programming Language C++
Audience:	EWG, LEWG
Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

This paper proposes a new Standard Library function, `swap_representations`, built upon the foundation of trivial relocatability proposed by [P2786], that can be used to enable the optimizations of `std::swap` similar to those described in [P1144] by swapping the in-memory representations of two objects without ending their lifetimes.

2 Revision History

R0 May 2024 (pre-St Louis mailing)

Initial draft of this paper.

3 Introduction

We can define a *swap semantic* for C++ in a variety of ways. For example, most of the Standard Library algorithms are specified using the notion of swapping object *values* using the public interface of an object; two lvalues can be swapped if their move constructor, move-assignment operator, and destructor are all publicly accessible with the final state being that of move-assigning each original object to the other.

Another notion for the swap semantic might be to exchange the object representations of two objects, as defined by the core language. Such a swap semantic, much like the object relocation facility described by [P2786], could be implemented efficiently using `memcpy`-like intrinsics and need not rely on a publicly accessible interface. Note that for certain kinds of types, swapping their object representations could produce different results to swapping their values using the assignment operators, e.g., `std::tuple<int&>`, or a container using allocators that do not propagate on swap.

In [P1144], Arthur O'Dwyer demonstrates that significant runtime benefits can be achieved in the Standard Library algorithms for types where swapping their object representations produces the same behavior as swapping their values. However, swapping object representations using a facility like trivial relocation runs afoul of C++'s object lifetime rules for nontrivial types.

This paper provides the missing library primitive that can safely swap objects' value representations without ending the lifetime of either object or any potentially overlapping subobjects, which can serve as the foundation for [P1144]-style optimizations in the Standard Library.

It further provides, as a pure library extension, the necessary trait to distinguish types where swapping their value representation is *not* the same semantic as swapping their values — enabling the key optimization of `std::swap`.

4 Open Design Issues and FAQ

A variety of concerns are not addressed by this paper but are described here to simplify the design review, in case reviewers feel strongly that this paper would be incomplete until some of or all these concerns are addressed. FAQ are also answered here to provide further clarity.

4.1 Why would I want to swap object representations?

Swapping object representations is a low level operation that is useful when writing data structures that need the full efficiency provided by the hardware a program is running on. It is a primitive semantic at the level of the abstract machine, that is expected to be highly optimizable when it can be reduced to operations on raw memory, although efficiency and semantics are separate properties.

4.2 Should we permit swapping representation with xvalues?

We are following the design of `std::swap` for now, but this is a lower-level facility.

4.3 Which Standard Library types are safe for the `swap_representations` function?

The Standard Library specification says nothing — or as little as possible — about the implementation of the types that it specifies, so no guarantees are made regarding which types in the Standard Library users can safely pass to `trivially_swap`.

The current [P2786] proposal similarly says nothing about which types in the Standard Library can be trivially relocated, but even if we were to explicitly specify a set of library types that must be trivially relocatable, that would be insufficient to guarantee that such types do not have laundry and thus may safely swap representations. Similarly, no guarantees are made that swapping representations is consistent with swapping values.

We expect that something must be said, even if only to explicitly make all such concerns a choice deferred to implementers' quality of implementation (QoI). We would expect a more significant follow-up paper to review the whole library and to make specific guarantees on a subset of types that do — or do not — offer such guarantees.

[P3262R0] is an initial draft of such a paper.

4.3.1 For what types can I swap representations but not values?

We can swap representations, but not values, for types without public destructors, move constructors, or assignment operators, and for immovable types.

4.3.2 For what types can I swap values but not representations?

We can swap values, but not representations, for types in which move-assignment produces a different result to move-construction.

4.3.3 For what types can I swap neither representations nor values?

We can swap neither representations nor values for types with data members that cannot be replaced, including const-qualified nonstatic data members or nonstatic data members that are references. Note that a `swap` for values may still be defined that does not change these members, such as `swap` for `tuple<T&>` that swaps the referenced elements, but such semantics strictly go beyond swapping values.

4.4 Can I mark a type as trivially relocatable but unable to swap representations?

No. We have no use cases for such an annotation, and the cost of providing a further annotation to the Core language to achieve such a minor effect outweighs any desire to do so.

Nothing would prevent a follow-up paper adding such functionality if it were deemed essential, but absent a strong motivating use case, this proposal's authors will spend no further time on this question.

4.5 Can I mark a type as able to swap representations but not trivially relocatable?

No. We have no use cases for such an annotation, and the cost of providing a further annotation to the Core language to achieve such a minor effect outweighs any desire to do so.

Nothing would prevent a follow-up paper adding such functionality if it were deemed essential, but absent a strong motivating use case, this proposal's authors will spend no further time on this question.

4.6 What is *pointer end-zap*?

Pointer end-zap is the term used within WG21 to describe the effect of ending an object's lifetime on pointers and references to that object. In particular, all pointers to the original object at the address denoted by those pointers become *invalid*, meaning that doing anything but ending their lifetime or assigning them a new valid pointer value is undefined behavior (UB). In particular, even if a new object is constructed at the given address, deferencing those old pointers or even comparing or copying them — such as passing by-value into a function call — remains UB.

Mitigating some of the concerns arising from the end-zap behavior in C++26 has been attempted, notably in paper [P2414R2]; however, replacing nontrivial object representations by overwriting their memory remains a problem unless we create a new feature in the Standard to enable such behavior, such as a Library function with "magic" wording for the compiler.

5 Basic Design Principles

5.1 Create a feature for users, not just the Standard Library

Swapping object representations is often needed for data structures that manage their elements in contiguous storage or in some other location that is not a node at the end of a pointer.

However, it is also a sharp tool that is not expected to see much use other than optimizing the internal management of data structures.

5.2 A formal specification is built around object lifetimes

C++ has a well-specified object model that is important to optimizers and analysis tools alike. Such tools must reason about object lifetimes and, importantly, minimize the doubt created for developers regarding that reasoning leading to false positives or false negatives when seeking to optimize or alert users.

5.3 Make the minimal necessary change

Small features tend to be more composable, with fewer constraints restricting their use and applicability. In particular, we do not anticipate any need to add new vocabulary to the Core language to deliver this feature.

5.4 Behavior must be reliable

No freedom for QoI in semantics is an important quality that builds on the well-specified object model.

5.5 Guard against accidental undefined behavior

A *Mandates:* clause will restrict the new Library API to support only types that would produce well-defined behavior.

5.6 Be consistent with existing similar facilities

The Standard Library specification historically uses type traits and "magic" functions that are understood by the compiler for this kind of feature. Similarly, we should follow existing naming conventions where they make sense.

5.7 Do not review all Standard Library types to support `swap_representations`

The Standard Library is *huge*, and specifying which types must support trivial relocation — and additionally support `swap_representations` — is a task left to a follow-up paper, [P3262R0], assuming the Standard Library groups feel such specification is needed.

Leaving such choices as a QoI concern for implementers should suffice for the initial release, and we will need to canvas vendors to determine which types can support explicit guarantees in the future without invalidating their implementations. Note that a large overlap occurs between which library types support trivial relocation and which support optimizing `swap` by calling `swap_representations`, but the mapping is not one to one, e.g., `std::tuple<int &>`.

Note that while our proposal should just *do the right thing* for `std::tuple` without any work from Standard Library implementers, we would still need to update the specification to say that.

6 Proposed Solution and Alternative Designs

We offer our proposed solution, and for reference, we include some alternative designs for the library interface that were considered but rejected as we developed our proposed solution.

6.1 Proposed solution

We propose a new function for the Standard Library, `swap_representations`, that exchanges — passed by reference just like `std::swap`— the value representations of two objects of the same type. This is a "magic" library function that achieves the postcondition without ending the lifetime of either object. Note that by swapping the value representation rather than the object representation, we intend to support potentially overlapping empty subobjects since empty types have no value representation, i.e., a value representation of zero bytes, regardless of the size of their object representation. Also note that no *empty types* are mentioned in the Core language, only objects of zero bytes.

To maintain well-defined behavior in the C++ object model, we mandate that the type of the swapped objects must be trivially relocatable, as proposed by [P2786].

As we are passing references, it would be too easy to accidentally swap two derived objects of different type that share a polymorphic base class. This operation is not a problem when swapping representations of complete objects, but will lead to undefined behavior in the case of swapping base classes with different vtables. Hence, we will require a compile-time *force* parameter to allow use with polymorphic types when the caller knows they are dealing with complete types, such as elements in a container, but that defaults to `false` producing a diagnosable error otherwise.

Naturally, such a low-level facility that is close to the compiler should be supported in a freestanding implementation. Our proposed solution has no dependencies on dynamic memory, no exceptions, and no other concerns that a freestanding implementation might raise.

In addition we provide a new user-customizable type trait, `swap_uses_value_representations_v`, that indicates when `std::swap` can safely use the `swap_representations` function as an optimized implementation without changing the observable behavior of the program. This trait is a pure library extension, and does not rely on any further compiler work, or "magic", beyond an implementation of both [P2786] and [P2996R2].

6.2 Alternative design: `trivial_swap`

Since operations at the level of moving bytes around memory are often defined only for types with trivial properties, we considered including the term `trivial` in the proposed function name. After consideration, we decided that such a name too literally addressed the notion of matching trivial to memory operations and opted for a simple, clear name.

Among the names we considered were (listed alphabetically)

— `trivial_swap`
— `trivially_relocate_swap`
— `trivially_swap`
— `trivially_swap_as_relocate`
— `trivially_swap_by_relocate`
— `trivially_swap_representations`
— `trivially_swap_reps`
— `trivially_swap_using_relocate`
— `trivially_swap_with_relocate`

6.3 Alternative design: Pointer parameters

Since we are performing low-level operations that are essentially compiler primitives, we considered that passing arguments as pointers might be more conventional.

```
template <class T>
  void swap_representations(T* a, T* b);
```

In the end, we decided that the expected user interface should look more like `std::swap`.

7 Implementation to Recursively Test Members for Support

It is relatively straightforward to implement a trait that queries all nonstatic data members and base classes to verify that all support the trait and that allows users to opt out by specializing the trait for their types that do not follow the expected library semantic that `swap` and `swap_representations` produce identical results.

Assumptions that feed our algorithm include the following.

— Types that are not trivially relocatable are not supported due to the *Mandates* specification for the `swap_representations` function.
— Types with nonstatic data members of reference type need explicit handling because types with such members that also support assignment typically assign through reference members rather than rebinding.
— Types with const-qualified nonstatic data members need special handling because types with such members that also support assignment typically ignore their `const` members rather than trying to change their value.
— Polymorphic types need special handling because swapping representations is going to swap their vtables, which is not the behavior expected from `std::swap`, but that is a concern only for the type of the complete object; data members are always of a most-derived type.
— Types that comprise types for which this trait is `false`, either implicitly or because it has been deliberately specialized, need special handling.

We do not call out types with virtual bases since they can never be trivially relocatable according [P2786] and thus can never pass the *Mandates* for the `swap_representations` function.

Implementing this trait without any compiler magic is possible using just the reflection facilities proposed by [P2996R2]. We present a sample implementation, and a primitive test driver for this trait implementation can be found on Godbolt Compiler Explorer (https://godbolt.org/z/hP34YdxG9):

```
template <class T>
constexpr bool swap_uses_value_representations_v = !std::is_polymorphic_v<T> and [] {
   if (!is_trivially_relocatable_v<T> or std::is_const_v<T>) {
      // References are not trivially relocatable unless they are nonstatic
      // data members, so they are covered by the is_trivially_relocatable_v test.
      return false;
   }

   if (std::is_class_v<T>) {
      for (std::meta::info mem : subobjects_of(^T)) {
         std::meta::info ty = type_of(mem);
         if (!test_type(^swap_uses_value_representations_v, ty)) {
            // self-referential to support customization by specialized this
            // variable template
            return false;
         }
      }
   }
   return true;
}();
```

Note that we have chosen to not include `is_swappable_v<T>` in any part of this algorithm since the intent is to determine that the result of swapping two objects is the same as swapping their value representation; if it is not possible to `swap` the objects, then the question does not arise. However, an alternative formulation might note that reference types and const-qualified types are not swappable and that users have no ability to customize that situation. Hence, we might consider a more abstract approach and simply verify that each nonstatic data member and base class is swappable.

Note that with this specification, a well-written class type should rarely need to specialize this trait. Let's consider the examples we have given from the Standard Library. `std::tuple<int &>` should be both trivially relocatable and unable to optimize `std::swap` for existing implementations without further work, while `std::tuple<int>` should be both trivially relocatable and `swap` optimized without even marking the class template as `trivially_relocatable`. Similarly, a natural implementation of `std::pmr::polymorphic_allocator` will internally store either a reference to a `std::pmr::memory_resource` or a `const` pointer (not a pointer-to-const), implicitly deleting the assignment operators without the need to explicitly delete them. In doing so, the implementation will disable the `swap` optimization while naturally being trivially relocatable, so we have no need to use the mark-up from P2786, and the primary template of all the `pmr` containers will produce the correct behavior too since the trait tests the members including the allocator data member. Finally, `std::list` is trivially relocatable only if the implementation allocates the sentinel node rather than storing it in a data member. As the swap optimization trait tests for trivial relocatability, it should pick up the correct default for both implementation strategies without a need to explicitly specialize the trait. Note that for any container implementation to be trivially relocatable, it will need to use the `trivially_relocatable` markup from P2786. Finally, note that that all three types, apart from the nontrivially relocatable `std::list` implementation, support the `swap_representations` function directly.

Design note: We have chosen to use specializing the trait itself — or at least a variable template denoting the trait — as the preferred form of user customization. An alternative might be to use some other method that is similar to how we customize detecting allocator support by asking users to supply an identifier in their class if they wish to support — or in this case, to disable support for — a particular library semantic.

8 Analysis

Here we analyse the problems that the C++ abstract machine raises that mean we cannot simply swap the object representations of any two objects that bind to references of the same type.

8.1 Swapping values rather than representations

`std::swap` exchanges "values" that are essentially defined by the move constructor and move-assignment operator. Using operations to directly swap the bytes swaps the object representations, which are not always the same thing.

8.2 Object lifetime and pointer end-zap

An object's lifetime ends when its storage is released or is used to store another object. When its lifetime ends, all pointers and references to that object are invalidated, which is colloquially known as *pointer end-zap*.

Special rules dictate when an object can be replaced directly without ending its lifetime and triggering pointer end-zap. Notably, objects can be *transparently replaced* by another object of the same type as long as the object being replaced is not a const-qualified *complete* object or a *potentially overlapping* subobject. Similarly, a member subobject can be replaced by constructing a new object of exactly the same type in its place, with problems arising only if that object genuinely does overlap with another member subobject.

8.3 Object representation vs value representation

The *object representation* of an object is the set of bytes the compiler uses to store that object. To allow for alignment, this set of bytes is often larger than simply adding up all the sizes of that object's member subobjects and introduces padding bytes. The *value representation* of an object is just those nonpadding bytes that store information. For example, an *empty* class has an object representation of at least one byte, which is important for several reasons, one of which is to correctly represent arrays of empty types. On the other hand, the value representation of an empty type is zero bytes, i.e., there is no value representation.

8.4 Toward a well-defined function

If we specify a new library function to swap the value representations (rather than the object representations) of two objects of the same type, we expect to preserve the properties that matter to the abstract machine that defines C++. In particular, since empty types have no value representation, we can safely exchange their value representations even when they are actually overlapping subobjects because this new swap function would be a no-op. Not touching the padding bytes is "magic" needed by the implementation to preserve all the outstanding guarantees.

8.5 Nested subobjects

A complete object can store a variety of nested subobjects. The obvious case is all of its member subobjects, but nested subobjects can be created in other ways too. For example, if a class has a nonstatic data member that is an array of `std::byte`, a nested subobject with dynamic storage duration can be created in that storage.

Any new function that swaps value representations needs to support swapping two complete objects that might have such nested subobjects created within their storage. The value representation of the array of `std::byte` is all those bytes, which includes the full object representation of any nested subobjects. The easiest specification would simply invoke pointer end-zap on all such nested subobjects, starting the lifetime of new nested subobjects after the complete objects' value representations have been exchanged. A stronger specification would not invoke pointer end-zap if both complete objects are storing nested subojects of the same complete object type.

8.6 Uninitialized subobjects

By exchanging just the bytes of the value representation, we avoid touching bytes that have indeterminate values, which would be UB. However, if an object comprises any uninitialized member subobjects, then the same issue arises. We note that special dispensation is given to reading and writing raw memory through pointers to `unsigned char` or `std::byte`, and we intend to channel a similar dispensation, without addressing specific implementation details in the formal wording.

9 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4981], the latest draft at the time of writing, with [P2786] applied.

9.1 Specify the `swap_representations` function

9.1.1 `swap_representations`

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

20.2.2 [memory.syn] Header `<memory>` synopsis

```
// 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 <bool force, class T>
  void swap_representations(T& a, T& b);                                 // freestanding
```

```
template <class T>
  T* trivially_relocate(T* begin, T* end, T* new_location);                // freestanding
```

20.2.6 [obj.lifetime] Explicit lifetime management

```
template <bool force, class T>
  void swap_representations(T& a, T& b);
```

1 *Mandates:* `T` is a complete object type, and `is_trivially_relocatable_v<T>` is `true`. If `is_polymorphic_v<T>` is `true` then `force` is `true`.

2 *Preconditions:* If `T` is a polymorphic type, then `T` is the most derived type for the objects `a` and `b`.

3 *Postconditions:* `a` has the value representation that `b` had prior to this function call; `b` has the value representation that `a` had prior the this function call.

4 *Throws:* Nothing.

5 [*Note:* A likely implementation will use compiler-specific functionality that simply swaps the bytes comprising the value representation of both objects without affecting their lifetimes. —*end note*]

10 Appendix A: Example Implementations

10.1 Implement using reflection

Dan Katz, one of the authors behind the reflection proposal [P2996R2], has provided an implementation of a `swap_representations` function that can deliver most of the semantics without resorting to compiler magic, but that is not expected to provide the efficiency of a fully optimized implementation that leans entirely into compiler magic.

```
#include <experimental/meta>

#include <print>
#include <tuple>
#include <type_traits>

namespace __impl {
  template<auto... vals>
  struct replicator_type {
    template<typename F>
      constexpr void operator>>(F body) const {
        (body.template operator()<vals>(), ...);
      }
  };

  template<auto... vals>
  replicator_type<vals...> replicator = {};
}  // namespace __impl

template<typename R>
consteval auto expand(R range) {
  std::vector<std::meta::info> args;
  for (auto r : range) {
    args.push_back(std::meta::reflect_value(r));
  }
  return substitute(^__impl::replicator, args);
}

template <typename T>
void do_swap_representations(T& lhs, T& rhs) {
  // This implementation cannot rebind references, and does not handle const
  // data members --- still need to decide whether we support the latter

  if constexpr (std::is_class_v<T>) {
    // This implementation ensures that empty types do nothing
    [: expand(bases_of(^T)) :] >> [&]<auto base> {
      using Base = [:type_of(base):];
      do_swap_representations<Base>((Base &)lhs, (Base &)rhs);
    };

    [: expand(nonstatic_data_members_of(^T)) :] >> [&]<auto mem>{
      do_swap_representations<[:type_of(mem):]>(lhs.[:mem:], rhs.[:mem:]);
    };
  }
  else if constexpr (std::is_array_v<T>) {
    static_assert(0 < std::rank_v<T>, "cannot swap arrays of unknown bound");
```

```cpp
    using MemT = std::decay_t<decltype(lhs[0])>;
    [:expand([] {
        std::vector<size_t> result;
        for (size_t idx = 0; idx < std::size(result); ++idx)
          result.push_back(idx);
        return result;
    }()):] >> [&]<size_t Idx> {
        do_swap_representations<MemT>(lhs[Idx], rhs[Idx]);
    };
  }
  else if constexpr (std::is_scalar_v<T> or std::is_union_v<T>) {
    // correct for unions without tail padding, including swapping active element
    // will need compiler magic to eliminate tail padding though
    // language ensures to not overwrite tail padding for scalars
    // May be broken if union overloads `operator=`
    T intrm = lhs;
    lhs = rhs;
    rhs = intrm;
  }
  else if constexpr (std::is_reference_v<T>) {
    static_assert(false, "Does not yet rebind references");
  }
  else {
    static_assert(false, "Unexpected type category");
  }
}

template <bool enable = true, class T>
void swap_representations(T& lhs, T& rhs) {
    // need to refactor the API so the user can force on the first argument
    // and deduce the rest
    static_assert(enable || !std::is_polymorphic_v<T>,
                  "Use swap_representations<true> to force swapping a polymorphic type");

    do_swap_representations(lhs, rhs);
}

int main() {
    std::tuple<int, int, int> a = {1, 2, 3}, b = {4, 5, 6};
    std::println("a:  <{}, {}, {}>", get<0>(a), get<1>(a), get<2>(a));
    std::println("b:  <{}, {}, {}>", get<0>(b), get<1>(b), get<2>(b));
    std::println("");

    swap_representations(a, b);
    std::println("a:  <{}, {}, {}>", get<0>(a), get<1>(a), get<2>(a));
    std::println("b:  <{}, {}, {}>", get<0>(b), get<1>(b), get<2>(b));
    std::println("");

    int arr0[] = {1, 2, 3}, arr1[] = {3, 2, 1};
    std::println("arr0:  <{}, {}, {}>", arr0[0], arr0[1], arr0[2]);
    std::println("arr1:  <{}, {}, {}>", arr1[0], arr1[1], arr1[2]);
    std::println("");
    swap_representations(arr0, arr1);
    std::println("arr0:  <{}, {}, {}>", arr0[0], arr0[1], arr0[2]);
    std::println("arr1:  <{}, {}, {}>", arr1[0], arr1[1], arr1[2]);
}
```

10.2 Implement using compiler magic

Corentin Jabot has provided a highly optimized version of this function in his Clang branch for trivial relocation that would be available on Godbolt before the St Louis meeting.

This implementation takes advantage of Clang intrinsics to determine the footprint of an object without any tail padding, and swaps the bytes of the two value representations. This omits swapping tail padding, but does swap internal padding, which is harmless but not something a non-intrinsic operation would be allowed to do.

We do not believe the Clang middle and back ends track the active member of a union in a form that would cause this to be UB in that implementation, but need to verify with experts in those parts of the tool chain to confirm that no further hints need to be passed lower into the compiler in such cases.

11 Acknowledgements

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

Special thanks to Dan Katz, not only for the help figuring out the reflection API, but also for providing an implementation on Godbolt Compiler Explorer that we can test against, and several metafunctions presented in this paper!

Thanks to Brian Bi and Joshua Berne for reviews of early drafts of this paper, especially addressing concerns with the C++ object model and the wording.

Thanks to Corentin Jabot for an implementation of the trivial relocation language facility that underpins this proposal, and the optimized implementation of the proposed library API for Clang.

Thanks to Arthur O'Dwyer for doing all the initial work to demonstrate the value of this optimization, even though we have different ideas for how that optimization should be made available to our users.

And special thanks to Matt Godbolt for his amazing Compiler Explorer tool that makes possible so much of the R&D that goes into papers like this.

12 References

[N4981] Thomas Köppe. 2024-04-16. Working Draft, Programming Languages — C++.
https://wg21.link/n4981

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

[P2414R2] Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, and Anthony Williams. 2023-12-17. Pointer lifetime-end zap proposed solutions.
https://wg21.link/p2414r2

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

[P2996R2] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-02-15. Reflection for C++26.
https://wg21.link/p2996r2

[P3262R0] Alisdair Meredith. Specifying Class Properties.
https://wg21.link/p3262r0