Trivial relocatability options

Proposal for an alternative approach to trivial relocatability

Document #:	P2786R2
Date:	2023-06-02
Project:	Programming Language C++
Audience:	Evolution Incubator Library Evolution Incubator

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

Contents

1 Abstract

This paper examines an approach to support trivial relocatability, building upon ideas in previous papers [P1029R3] and [P1144R6], and leveraging the experience of supporting *bitwise movability* in the BDE library. It embraces the motivation for such a feature given in those papers, while providing what we believe to be a more rigorous design and specification.

2 Revision history.

2.1 R2: June 2023 (Varna meeting)

— Updated most references to P1144 to the May mailing [P1144R8]
— Attempted to clarify the new Terms and definitions

— Added missing example for New syntax
— Moved all non-*essential* functionality to Library Extensions

2.2 R1: May 2023 (pre-Varna mailing)

Mid-term mailing following feedback from Issaquah.

The most significant change is that we moved analysis and comparisons with [P1144R6] to a separate co-authored paper, [P2814R0]. More specific changes are detailed below:

— Added `constexpr` to relocate functions, and to the design decisions.
— Added *// freestanding* comments on every library function.
— Renamed `move_and_destroy` as `uninitialized_move_and_destroy`.
  — Documented adding the algorithm as a design decision
  — Fixed precondition
  — Require forward iterators for input range, as we expect to modify/destroy elements
  — Add full family of overloads, consistent with `uninitialized_*` standard algorithms
— Reviewed use of *voidify* as consistent with library text with Issaquah papers applied
— Provide a complete specification for `relocate` that handles overlapping ranges.
— Revise concerns with application to `swap`, deferring any further work to a separate paper.
— Struck redundant `inline` from definition of `is_trivially_relocatable_v`

2.3 R0: Issaquah 2023

Initial draft of this paper.

3 Introduction

For our purposes, a *trivial relocation operation* is a *bitwise copy* that ends the lifetime of its source object, as-if its storage were used by another object (6.7.3 [basic.life]p5). Importantly, nothing else is done to the source object, in particular **its destructor is not run**. This operation will typically (though exceptions are not forbidden) be semantically equivalent to a move construction immediately followed by a destruction of the source object.

Any trivially copyable type is *trivially relocatable* by default. Many other types, even those which have non-trivial move constructors and destructors, can maintain their correct behavior when trivially relocated — skipping the source object's destructor allows for skipping all bookkeeping updates that might need to be done by the target object's move constructor. This includes many resource-owning types, such as `vector`, `unique_ptr`, and `shared_ptr`.

Note that simply doing a bitwise copy of these non-trivially-copyable objects will, as of C++23, result in undefined behavior (when the copied bytes are treated by later code as an object of the original type). Making this operation well-defined for those types which opt into this behavior is the primary goal of proposing this feature as a language extension. The secondary goal is to implicitly support a wider range of trivially relocatable types. The tertiary goal is to provide better diagnostics when trivial relocation semantics are misused.

4 Motivating use cases

We believe [P1144R8] has done a good job a motivating proposals in this area. Here, we highlight the specific use cases that drive our proposal, just as a reinforcement of the earlier paper.

4.1 Efficient `vector` growth

Suppose we have a move-only type, `class MoveOnlyType` (for example, a unique ownership smart pointer), and we wish to hold a vector of these types `std::vector<MoveOnlyType>`. Simply emplacing 5 of these objects would require that `MoveOnlyType`'s move constructor and destructor be called 7 additional times due to the vector expansion required as more elements are inserted than the capacity (at least in one current implementation of std::vector).

If `MoveOnlyType` were trivially relocatable, and if `std::vector` were to take that into account as an optimization, then the vector expansion caused by these 5 emplacements would require only 3 `memmove` operations, with no additional calls to `MoveOnlyType`'s move constructor and destructor.

For this example we are assuming an initially empty `vector` with no reserve capacity, and that the implementation has a growth strategy of doubling the reserved space when more is required, from 0 to 1 to 2 to 4 to 8.

4.2 Moving types without empty states

Some types do not have a non-allocating empty state, so cannot have a `noexcept` move constructor. One example is a known implementation strategy for `std::list` that always allocates at least a sentinel node. Lacking a non-throwing move constructor, vectors of such list have a painful growth strategy. However, as long as the sentinel does not maintain a back-pointer into its list object, such a type can be trivially relocated as the old object immediately ends its life without running its destructor, so does not have to restore invariants — there is no window of opportunity to access the live object in a state where it has broken invariants.

4.3 `pmr` types are often trivially relocatable

The original motivation for this feature in the BDE library was to ensure efficient movement of allocator aware types, using the allocator model that became standardized in namespace `std::pmr`. As the allocator is simply a pointer to a memory resource, and allocated memory does not reside within the owning object itself, many non-trivial allocator-aware types can be trivially relocatable if an appropriate markup is available.

4.4 Future proposal for language support for allocators

The authors are also working on a separate proposal for direct language support for allocators, based upon the `std::pmr` design ([P2685R0]). That proposal anticipates support for trivial relocatability.

5 Experience at Bloomberg

Bloomberg has relied heavily on low level optimizations enabled by assuming the trivially relocatable model holds. This implementation experience is built on the, so far valid, assumption that no current compilers are optimizing to transform programs based on the specific undefined behaviors we exploit. The emulation is achieved through a type trait, `bslmf::IsBitwiseMovable`. More recently, in an experimental branch to explore language extensions, `pbastd::is_trivially_relocatable` is used to demonstrate relocation of types using `std::pmr::polymorphic_allocator`. This experimental model is a pure library extension, and has no impact unless libraries are written to test this trait before choosing an optimized implementation. In particular, types that are not trivially copyable must opt into the trait with a special traits markup, or by specializing the trait for their relocatable type. Note that user specialization would not be permitted for a standardized type trait, per 21.3.2 [meta.rqmts]p4.

The initial language support we propose is that the new trait will detect trivially copyable types as also being trivially relocatable by default, while other types will default to non-trivially-relocatable. This part can be covered by a library emulation, implementing the new trait in terms of `std::is_trivially_copyable`.

6 New Terms and Definitions

We introduce and specify the following new terms to better communicate our intent. These terms can be found in numerous other proposals, and the definitions proposed here are very similar.

First, we will address the notion of what *relocation* should mean in the context of C++. We believe the topic deserves a higher level treatment, such as described in [P2839R0], but for our purposes it is sufficient to define the operation we wish to optimize.

— **relocate**: To **relocate** a type from memory address `src` to memory address `dst` means to perform an operation or series of operations such that an object equivalent (often identical) to that which existed at address `src` exists at address `dst`, that the lifetime of the object at address `dst` has begun, and that the lifetime of the object at address `src` has ended.

— **relocatable**: To say that an object is **relocatable** is to say that it is possible to **relocate** the object from one location to another.

Next, we define terms specific to the optimization we are proposing in this paper, which will build on a new type category in the core language specification, *trivially relocatable* types.

— **trivially relocatable**: Conceptually, a type is **trivially relocatable** if it can be **relocated** by means of copying the bytes of the object representation and then ending the lifetime of the original object without running its destructor.

— **trivially relocatable type**: A **trivially relocatable type** is a type that is **implicitly trivially relocatable**, and/or is **explicitly trivially relocatable**, and/or is an array of **trivially relocatable types**; otherwise the type is not **trivially relocatable**. Any otherwise **trivially relocatable** type can be declared non-**trivially relocatable** by means of the `trivially_relocatable` keyword with value `false`.

— **implicitly trivially relocatable**: A type is **implicitly trivially relocatable** if it has no user-provided or deleted destructors, no virtual base classes, all its base classes (if any) are trivially-relocatable, all its non-static data members (if any) are trivially-relocatable, and the constructor selected for initialization from a single rvalue of the same type is neither user-provided nor deleted.

  — If a class has an appropriate (move or copy) constructor, then its access level (`public`/`protected`/`private`) has no bearing on whether that class is **implicitly trivially relocatable**. This non-requirement for accessibility follows the same model as the standard specification for **trivially copyable** class types. Similarly, there are no requirements that the destructor be accessible, merely that it is neither deleted nor user-provided.

  — The copy constructor is not relevant unless it inhibits the declaration of move constructors; then a class is not **implicitly trivially relocatable** unless the copy constructor is implicitly defined.

  — Examples of types that are **implicitly trivially relocatable** are trivially copyable types (such as scalar types), aggregates of trivially relocatable types, including arrays of such types, and such aggregates with `const` and/or reference data members. Empty types can satisfy the requirements for an **implicitly trivially relocatable** type.

— **explicitly trivially relocatable**: A type is **explicitly trivially relocatable** if it is a user defined (class) type that is defined with the contextual keyword `trivially_relocatable` and value `true`, with the following proviso:

  — An **explicitly trivially relocatable** class type may not contain any non-static data members that are not **trivially relocatable**, nor any base classes that are not **trivially relocatable**, nor have any virtual base classes. I.e., it is a diagnosable error to add the keyword with value `true` to a class that does not qualify.

It is important to note that we are proposing to permit, by means of the `trivially_relocatable` keyword, types that would otherwise be non-copyable and non-movable to be (trivially) **relocatable**. For this reason we cannot define **relocate** and **relocatable** in terms of a move construction followed by a destruction (the definition used by [P1144R8]). The ability to explicitly make a type trivially relocatable enables providing a customized (and thus non-trivial) move constructor and destructor while declaring that the compound operation is trivial.

7 Proposed Language Changes

Our proposal changes and extends C++23 as follows.

7.1 New type category

To better integrate language support, we further recommend that the language can detect types as **trivially relocatable** where all their bases and non-static data members are, in turn, **trivially relocatable**; the constructor selected for construction from a single rvalue of the same type is neither user-provided nor deleted; and their destructor is neither user provided nor deleted. This definition follows the same principle used in the standard to define trivially copyable.

7.2 New semantics

In order to ensure that libraries taking advantage of the trivially relocatable semantic do not introduce **undefined behaviour**, the model of lifetimes for objects must be extended to allow for relocation of **trivially relocatable** types. As the compiler cannot know if a specific `memcpy` or `memmove` call is intended to duplicate or to move an object, we propose introducing the `trivially_relocate` function template to call `memmove` on our behalf, which signifies to the compiler and other source analysis tools that the lifetime of the new object(s) has begun and the lifetime of the original object(s) has ended:

```
template <class T>
   requires is_trivially_relocatable_v<T>
constexpr
T* trivially_relocate(T* begin, T* end, T* new_location) noexcept;
```

Note that this function is designed to move a range of objects, rather than a single object, as that is expected to be the common use case. Further note that, consistent with its low level purpose often tied to move semantics, this function is denoted with `noexcept` despite having a narrow contract regarding valid and reachable pointers.

This design deliberately puts all “compiler magic” and core-language interaction dealing with the object lifetimes into a single place, rather than into a number of different `relocate`-related overloads. Note that there is no permission for a user to copy the bytes to perform a relocation themselves, unlike with trivial copyability, although that would still work for trivially copyable types.

`trivially_relocate` can be thought of as ending the lifetime of the moved-from objects, followed by a `memmove` , followed by `start_lifetime_as` (or maybe `start_lifetime_as_array`) on the moved-to objects. Unlike `memmove` on its own, it is restricted to trivially relocatable types rather than to implicit lifetime types.

Note that `start_lifetime_as` is constrained to work only for implicit lifetime types whereas this proposal is intended to support all trivially relocatable types, which are often not implicit lifetime types. The different constraints are appropriate in each case. For the currently specified `start_lifetime_as` function, the idea is that we point the compiler to a region of memory, and say “take these bytes of unknown provenance and turn them into objects”. In particular, we might be copying bytes into memory from a stream, and those bytes did not originate as objects in *this* abstract machine. Conversely, `trivially_relocate` takes existing valid objects in memory, copies their bytes to a new location, and asks the compiler to imbue life into specifically those bytes copied from known valid objects. It is important that the copying and imbuing life occur within the same transaction, as that gives the compiler its necessary guarantees. Hence, all the new functionality is bundled into a single `trivially_relocate` function, rather than decomposing into smaller parts that would allow the users to perform the `memmove` themselves.

Finally, observe that this function is `constexpr`. The intent is to support just the relocation of objects in transient dynamic storage, in order to implement the C++20 `constexpr vector` semantics without further `if consteval` magic in the implementation. We have not considered the impact on objects other than those having such dynamic storage duration, and it might merit some more core wording to restrict the `constexpr` applicability in such cases, or to more carefully consider general purpose `constexpr` relocations.

7.3 New type trait

In order to expose the relocatability property of a type to library functions seeking to provide appropriate optimizations, we propose a new trait `std::is_trivially_relocatable<T>` which enables the detection of **trivial relocatability**.

```
template< class T >
struct is_trivially_relocatable;

template< class T >
constexpr bool is_trivially_relocatable_v = is_trivially_relocatable<T>::value;
```

having a base characteristic of `std::true_type` if `T` is **trivially relocatable** and `std::false_type` otherwise.

Note that it is expected that the `std::is_trivially_relocatable` trait shall be implemented through a compiler intrinsic, much like `std::is_trivially_copyable`, so the compiler can use that intrinsic when the language semantics require trivial relocatability, rather than requiring actual instantiation (and knowledge) of the standard library trait. The trait must always agree with the intrinsic as users do *not* have permission to specialize standard type traits (unless explicitly granted permission for a specific trait).

7.4 New syntax

In order to enable trivial relocatability to be useful for more complicated (i.e., non-**trivially copyable**) types, it must be possible to explicitly mark non-**trivially copyable** types as **trivially relocatable**. As this should be an issue only for class types (including unions), we recommend adding a new contextual keyword `trivially_relocatable` as part of the class definition, similar to how `final` applies to classes. E.g.,

```
struct X;                            // Forward declaration does not admit `final`
struct X final {};                   // Class definition admits `final`
struct Y trivially_relocatable {};   // New contextual keyword placed like `final`
```

We propose one new contextual keyword that can be placed in a class-head to attach a trivially relocatable predicate to a class:

— `trivially_relocatable(`*`bool-expression`*`)` which is used:
  — With value `true` to explicitly make a class **trivially relocatable**, and
  — With value `false` to explicitly remove **trivial relocatability** from a class.

The boolean predicate is optional, with a plain `trivally_relocatable` defaulting to `true`.

It is possible, by means of the `trivially_relocatable(`*`bool-expression`*`)` specification, to declare a class as **trivially relocatable** even if that class has a user-defined copy constructor and/or move constructor and/or destructor. This differs from [P1144R8] in the following notable ways:

— Where `trivially_relocatable` is specified with value `true`, we do not require that the move constructor, copy constructor, and/or destructor be public or unambiguous. The `trivially_relocatable` specification takes precedence.

— It is possible to render, by means of the keyword and value `false`, any type, even a **trivially copyable** type, non-**trivially relocatable**.

Our motivation for the explicit specification *always* supplanting the implicit specification, rather than just the case of `true` supplanting `false`, is the confusion we encountered when considering other semantics in alternative designs below. It became clear that it was much simpler to reason about our examples when the trivial relocation specification could be trusted to mean literally what it said.

While we do not have use cases for making trivially relocatable types to be non trivially relocatable, we do not have use cases to disallow it either, and must choose a meaning for the syntax. Our experience with language design in general has been that users will find corners where even the most obscure feature is useful, so prefer to not remove a potentially useful feature that is intuitive from the syntax.

It may be argued that this is a case where [P1144R8] leans on the semantics of the feature where this proposal leans on the syntax.

7.5 Diagnosable errors

In a non-dependant context, it would be a diagnosable error to mark a type as **trivially relocatable** if it comprises any bases or non-static members that are not **trivially relocatable**. Types with virtual base classes are automatically not **trivially relocatable**, as their implementation on some platforms involves an internal pointer. We prefer that this low level behavior is consistent across platforms, rather than left as an unspecified QoI concern, as our current experience has not yet turned up a usage of virtual base classes that would also benefit from this feature.

Note that there are no issues with virtual functions, as virtual function table implementations do not take a pointer back into the class, so the vtable pointer can be safely relocated.

7.5.1 Simple examples without a predicate

The common form is expected to be the simple case, without a predicate.

```
struct MyType trivially_relocatable : BaseType {
   // class definition details

   MyType(MyType&&);  // user supplied
      // Having a user-provided move constructor, `MyType` would not be
      // trivially relocatable by default.  The `trivially_relocatable`
      // annotation trusts the user that this type can indeed be trivially
      // relocated.
};

struct NotRelocatable : BaseType {
   // class definition details

   NotRelocatable(NotRelocatable&&);  // user supplied
      // Having a user-provided move constructor, `NotRelocatable` is not
      // trivially relocatable.
};


struct Error trivially_relocatable : BaseType {
   NotRelocatable member;
      // This class is ill-formed, as it requests to be trivially relocatable,
      // but the compiler can see a non-relocatable data member that cannot be
      // worked around.

   Error(Error&&);  // user supplied
      // There is nothing this move constructor can do to repair the trivial
      // relocatability property, as it is not invoked during trivial
      // relocation.
};
```

7.5.2 Examples using the predicate

The boolean predicate form, `trivially_relocatable(false)`, can be used to opt out of the behavior for a type that might otherwise be **trivially relocatable** by default. However, the main purpose of the predicate is to allow class templates to indicate their trivial relocatability where their opt-in might depend on the supplied template arguments.

For example purposes, let us consider the following two classes:

```
struct Relocatable trivially_relocatable(true ) {}; // trivially relocatable
struct Alternative trivially_relocatable(false) {}; // not trivially relocatable

static_assert( is_trivially_relocatable_v<Relocatable>);
static_assert(!is_trivially_relocatable_v<Alternative>);
```

Clearly, `Relocatable` is a trivially relocatable class type, and `Alternative` is a non-trivially relocatable class type. We will use these classes to illustrate how similar, but subtly different, class templates behave.

As an initial example, we write a simple aggregate that demonstrates we get the expected behavior that correctly deduces trivial relocatability when we have no user-supplied special members:

```
template<class TYPE>
struct Example {
  TYPE value_a;
  TYPE value_b;
};

static_assert( is_trivially_relocatable_v<Example<Relocatable>>);
static_assert(!is_trivially_relocatable_v<Example<Alternative>>);
```

However, for most of our remaining examples we are concerned with the case of a class template that provides its own special members, so needs to supply a trivial relocation specification. The examples look simple, and may lead to thinking "why am I messing with all this template syntax when the simple `Example` works?" but remember, these are deliberately simplified examples to highlight just the relevant code, and the underlying lesson is intended for larger code in practice, where `Example` would clearly not suffice.

As our first example, we write a class template that uses the trivially relocatable specification to forward the trivial relocatability of its dependent members:

```
template<class TYPE>
class Duo trivially_relocatable(is_trivially_relocatable_v<TYPE>)
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~Duo() {}  // User provided destructor so not implicitly relocatable
};

static_assert( is_trivially_relocatable_v<Duo<Relocatable>>);
static_assert(!is_trivially_relocatable_v<Duo<Alternative>>);
```

Next, we use type constraints in a `requires` clause instead, so see how the behavior differs:

```
template<class TYPE>
   requires is_trivially_relocatable_v<TYPE>
class RelocatableDuo trivially_relocatable
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~RelocatableDuo() {}  // User provided destructor so not implicitly relocatable
};
```

```
static_assert( is_trivially_relocatable_v<RelocatableDuo<Relocatable>>);
static_assert(!is_trivially_relocatable_v<RelocatableDuo<Alternative>>); // ill-formed
```

Observe that the static assertion for `RelocatableDuo<Alternative>` is ill-formed not because that `static_assert` fails, but rather, that the `RelocatableDuo` template cannot be instantiated for `Alternative` at all, i.e., `RelocatableDuo` is a template that wraps only trivially relocatable types, and so can guarantee to always be trivially relocatable.

For another example, we can try to make a class template unconditionally trivially relocatable:

```
template<class TYPE>
class TryRelocatable trivially_relocatable
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~TryRelocatable() {}  // User provided destructor so not implicitly relocatable
};

static_assert( is_trivially_relocatable_v<TryRelocatable<Relocatable>>);
static_assert(!is_trivially_relocatable_v<TryRelocatable<Alternative>>); // ill-formed
```

The `Alternative` instantiation fails again, but this time it fails because the `trivially_relocatable` specification is violated, which is a diagnosable error. The error message is likely to refer to the `value_a` and `value_b` members, where the error message for the `RelocatableDuo` example would be related to violating the type constraints of the `requires` clause.

Note that as an unadorned `trivially_relocatable` specification is equivalent to `trivially_relocatable(true)`, we can also consider the opposite case, `trivially_relocatable(false)`:

```
template<class TYPE>
class NotRelocatable trivially_relocatable(false)
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~NotRelocatable() {}  // User provided destructor so not implicitly relocatable
};

static_assert(!is_trivially_relocatable_v<NotRelocatable<Relocatable>>);
static_assert(!is_trivially_relocatable_v<NotRelocatable<Alternative>>);
```

Here we see both instantiations are again valid, and the trivial relocation specification forces both instantiations to be not trivially relocatable.

As a final example of `Duo`-like types, we consider what happens if one of the members is not type-dependent, and not relocatable:

```
template<class TYPE>
struct Erroneous trivially_relocatable
{
  Alternative value_a;  // ill-formed
  TYPE        value_b;
};
```

This case is ill-formed in all cases, and can be diagnosed in the template definition without waiting for an instantiation.

Another example where the trivial relocation specification might be useful is for trivial relocatability to be contingent on avoiding some small object optimization, such as:

```
template<class T>
class Container trivially_relocatable(sizeof(T) > SHORT_OPTIMIZATION_LIMIT)
{
    // Store small objects with an in-object representation, and dynamically
    // allocate storage for larger objects.

    // ...
};
```

Here we are concerned purely with whether a type is small enough to fit the small object optimization, and make no effect to further constrain on type. This might be how we approach retrofitting trivial relocatability into an existing library without raising ABI concerns.

7.6 Relocation functions

We propose one additional library function to trivially relocate ranges of objects. Note that this initial proposal does not provide a single-object relocation function as our primary motivation is to optimize relocating objects in bulk. It would be easy to add single-object `trivially_relocate` functions, but the effect can be achieved by calling the proposed function with a range of a single object, so we wait to hear that the evolution groups feel sufficiently motivated to request such convenience functions.

7.6.1 `trivially_relocate`

We propose the following function template to relocate trivially relocatable objects by means of a `memmove`. This function is the unique entry point into the core magic that tracks and manages object lifetimes in the abstract machine:

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

This function template is equivalent to:

```
memmove(new_location, begin, sizeof(T) * (end - begin));
```

with the precondition that `end` is reachable from `begin`. It further has the following two very important effects, that matter to the abstract machine but do not have any apparent physical effect (i.e., these effects do not change bits in memory), much like `std::launder`:

— it begins the lifetime of the objects `*new_location`, `*(new_location+1)`, …, through to `*(new_location+end-begin-1)`. If any of the objects or their subobjects are unions, they have the same active elements as the corresponding objects in the range `[begin, end)`.

— it ends the lifetime of the objects `*begin`, `*(begin+1)`, …, through to `*(end-1)`. This means it will be *Undefined Behavior* to access these objects or to attempt to destruct any of them.

Note: the first bullet (beginning the new lifetime(s) of the new object(s)) could be achieved by saying that this is equivalent to:

```
memmove(new_location, begin, sizeof(T) * (end - begin));
std::start_lifetime_as_array_without_preconditions(new_location, sizeof(T) * (end - begin))
```

but there is not currently a mechanism to end the lifetime(s) of the source object(s).

8 Follow-up Library Extensions

Our primary proposal 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 some possible library extensions that would be proposed as a follow-up paper for LEWG, to illustrate how we might resolve some library concerns.

8.1 `relocate`

We also 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.

8.2 `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.

9 The Problem With Vector

The goal of our optimization, through trivial relocation, is to make moving elements around a block container, such as `vector` or `deque`, more efficient. Bloomberg's experience with their implementation of polymorphic memory resources has demonstrated this value for many years. However, as we prepared this paper, Arthur O'dwyer pointed out that we are taking liberties that the standard does not permit. In particular, when erasing an element from the middle of a `vector`, or inserting an element into the middle of a `vector`, the existing elements are currently relocated using `operator=`, and if the result of the assignment operator for an element is not exactly the same as destroying the target element and then move constructing into it, then Bad Things can happen. We did not run into this problem at Bloomberg, as the polymorphic memory resource model ensures that all elements in a container use the same memory resource, and (as polymorphic allocators are equal) assignment and move-construction will produce the same result. However, we do not have an easily testable property to confirm that runtime condition is guaranteed by a `vector` itself — we are relying on implementation knowledge.

'

[container.reqmts] Containers

```
typename X::value_type
```

2 *Result:* `T`

3 *Preconditions:* `T` is *Cpp17Erasable* from `X` (see container.alloc.reqmts, below).

66.3 No `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

24.2.4 [sequence.reqmts] Sequence containers

```
a.erase(q)
```

45 *Result:* `iterator`.

46 *Preconditions:* For `vector` and `deque`, `T` is *Cpp17MoveAssignable*.

47 *Effects:* Erases the element pointed to by `q`.

48 *Returns:* An iterator that points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

To make the concern more visible, consider the following example where we store `std::pmr::vector` elements in a `std::vector`, i.e, the outer `vector` is using the default `std::allocator` that knows nothing about `pmr` memory resources. We will demonstrate, using a lambda, that running through the same code can produce very different results on which elements use which memory resource, depending on whether the `vector` reallocates to increase capacity.

```
#include <cassert>
#include <memory_resource>
#include <utility>
#include <vector>

int main() {
   using Element = std::pmr::vector<int>;
   using Container = std::vector<Element>;

   Container v;
   v.reserve(4);
   assert(4 == v.capacity());     // confirm initial capacity for first run

   using Alloc = std::pmr::monotonic_buffer_resource;
   Alloc a0, a1, a2, a3, a4;      // create 5 distinct allocators to track

    // test case is a lambda to ensure both runs are executing the same source code.
   auto fill = [&](bool firstTime) {

      // create 5 elements, each using a different memory resource
      Element e0{ {1, 2, 3}, &a0};
      Element e1{ {2, 3, 4}, &a1};
      Element e2{ {3, 4, 5}, &a2};
      Element e3{ {4, 5, 6}, &a3};
      Element e4{ {5, 6, 7}, &a4};

      // move the first 4 elements into the container, retaining their resource
      v.emplace_back(std::move(e0));
      v.emplace_back(std::move(e1));
      v.emplace_back(std::move(e2));
      v.emplace_back(std::move(e3));

      // verify each element has retained its memory resource
      assert(&a0 == v[0].get_allocator().resource());
      assert(&a1 == v[1].get_allocator().resource());
      assert(&a2 == v[2].get_allocator().resource());
      assert(&a3 == v[3].get_allocator().resource());

      // emplace an element into the middle of the container
      v.emplace(v.begin() + 2, std::move(e4));

      // Verify the memory resource used by each element after insertion, which
      // is different on the first run where the vector must grow its capacity,
      // compared to any following runs where elements are moved without
      // reallocation.

      assert(&a0 == v[0].get_allocator().resource());
      assert(&a1 == v[1].get_allocator().resource());
      if (firstTime) {
         // for the first run, the vector expands and each element has the right resource
         assert(&a4 == v[2].get_allocator().resource());
         assert(&a2 == v[3].get_allocator().resource());
         assert(&a3 == v[4].get_allocator().resource());
      }
      else {
         // for later runs, the vector rotates elements giving different resources
         assert(&a2 == v[2].get_allocator().resource());
         assert(&a3 == v[3].get_allocator().resource());
         assert(&a3 == v[4].get_allocator().resource());
      }
   };

   fill(true);   // first run to fill the vector `v`

   v.clear();    // reset the vector for a second run without losing capacity

   fill(false);  // next run to fill the vector `v`
}
```

10 Design choices

10.1 No library support is mandated

It is the intention that this extension be fully backwards compatible, and no library changes are *required*. Library implementers may, if they so desire, take advantage of this feature in order to improve performance, but they are not mandated to do so. This is a conservative position until we have confirmed that there would be no ABI concerns from explicitly applying this to the library specification (see ABI compatibility).

This extension relies on core language support, but does not change existing program behavior, even if the **trivially relocatable** property is deduced. It merely enables libraries to detect this property, and apply their own optimizations if they so desire.

Looking ahead to a follow-up paper from LEWG that performs a more detailed analysis, a library component (such as a container) will typically be affected in one of three ways:

1. Some components will, based on the definition in the standard, automatically gain **trivial relocatability** when appropriate based on the contained type. Examples are `array`, `pair` and `tuple`.
2. Some components could be marked as unconditionally **trivially relocatable**, if it is desired to do so in the future, for example `shared_ptr` and `filesystem::path`.
3. Some components could be marked as conditionally **trivially relocatable**, based on the **trivial relocatability** of contained types or other conditions. Examples are `optional` and `variant`.

As library implementations vary, the category into which a particular component is placed may vary.

Note that some types that intuitively seem like they might be unconditionally trivially relocatable are often only conditionally trivially relocatable due to forgotten template parameters with default arguments, such as allocators or the deleter policy for `unique_ptr`. In other cases such as `function`, semantic constraints to support small object optimizations can affect the choice. This is part of the reason to focus this proposal exclusively on the minimal core language support, while deferring a detailed library analysis to a follow-up paper for LEWG if this proposal is accepted.

10.2 Contextual keyword vs. attribute

Our design mandates all behavior regarding trivial relocatability rather than leaving potential usage unspecified, as a quality of implementation issue. In particular, several categories of misuse are expected to produce diagnostic errors.

We expect templates to make use of the `trivially_relocatable` markup, and prefer to avoid putting extra work parsing attributes through the template machinery, although there are no technical limitations here. For example, we believe that a specification relying on existing template wording will be simpler than trying to specify a how a pack expansion works within such an attribute (although the groundwork was laid when `alignas` was an attribute).

Usage of the `trivially_relocatable` markup should be clear and simple, especially with its mandated semantics, much as `final` became one of the first contextual keywords. Notably, `trivially_relocatable` would fall into the grammar in exactly the same location as `final` on a class.

By contrast, [P1144R8] prefers to use an attribute. The most obvious benefit is that an unnamed class can unambiguously use the attribute. When using a contextual keyword, we must limit usage to the case disambiguated by the opening paren of the boolean expression.

It has also been pointed out that the use of a parenthetical bool-expression in this position of the contextual keyword grammar might cause problems if some future language extension wanted to place a parenthetical list there, unrelated to contextual keywords:

```
struct Foo { };
struct Foo (Bar) { };  // always a syntax error today, but maybe we'd like to use this tomorrow
struct Foo final { };
struct Foo final (Bar) { };  // always a syntax error today, but maybe we'd like to use this tomorrow
struct Foo trivially_relocatable { };
struct Foo trivially_relocatable (Bar) { };  // uh-oh!
```

Note that this would not be an issue if the hypothetical extension were to place the new parenthetical *before* the contextual keywords, but that is already a constraint on future design. Such concerns do not arise with the attribute form.

We are not aware of any such hypothetical extensions at this point, but should be aware of our choices.

10.3 Type trait vs. concept

Existing library facilities in this space, such as observing trivial copyability, are rendered as type traits rather than concepts. Such type traits can easily be used to constrain templates in `requires` clauses, but do not participate in subsumption relationships.

It would be simple to specify a concept in terms of the proposed trait, but that trait is squatting on the good name. Note that the contextual nature of the keyword means there is no actual conflict here, but overloading an identifier this way might be confusing for users.

The C++ grammar enforces that concepts cannot be specialized, unlike templates. Specifying as a concept, rather than a type trait, would eliminate an unusual source of potential user error, and might have been the preferred approach for this reason, were it not for the precedent of the existing family of trivial type traits.

10.4 `trivially_relocate` as the single place for compiler magic

When it comes to exposing core language facilities as a library API, we prefer to keep the interaction as small and local as possible, ideally just a single "magic" function to imbue the new behavior.

Looking at a more general purpose library interface, we see the importance of being able to relocate arbitrary ranges of objects, using the traditional move-and-destroy semantic where trivial relocatability is not supported. We believe there are sufficient complexity getting the details right when handling overlapping source/destination ranges that it merits adding to the library. Support for overlapping ranges is inspired by `memmove` allowing for overlapping trivially relocatable ranges.

We offer two range APIs. `relocate` accepts pointers to ranges in memory, supports both movable and trivially relocatable types, supports overlapping ranges, and mandates trivial relocation when supported. `uninitialized_move_and_destroy` is directly inspired by `uninitialized_relocate` in [P1144R8], and supports iterator ranges more broadly. As it is not generally possible to identify overlapping ranges where the iterator types vary, we offer no support for overlapping ranges. Unlike [P1144R8], this specification does *not* permit trivial relocation of the elements, guaranteeing the move-and-destroy semantic.

Note that we deliberately use C++20 requires clauses to constrain these functions. We believe this is important for the tight core/library specification for `trivially_relocate`, but is largely cosmetic to ease review when looking at the other two functions. LWG may prefer to specify such functions with a *Constraints:* function element instead.

10.5 `constexpr` support for `std::vector` and `std::string`

In order to support simple and practical implementations of `vector`-like types that wish to optimize on the availability of trivial relocation, we introduce the `relocate` function that will delegate to the `trivial_relocate` function where possible. As both `vector` and `basic_string` marked as `constexpr` since C++20, we must similarly make these functions `constexpr` unless we want to further complicate the implementation of these types with deeper `if consteval` branches.

Design note: AJM is now thinking that the existing `if consteval` branches in these cases would suffice, and we would simply lose the optimization at compile-time. That might be an acceptable trade-off, while retaining `constexpr` support means the branches would likely look more similar.

11 Known concerns

11.1 Separately managed objects

Performing trivial relocations is generally not appropriate for an object whose lifetime is separately managed, such as a local variable on the stack, an object of static or thread storage duration, or a non-static data member within a class. Adding compiler support to better observe trivial relocations means we may get warnings on such misuse. (This concern is similar to destroying and recreating an object in-place. In such cases it is essential to recreate the object before its destructor will be called implicitly — hence a warning and not an error, as the idiom is already valid.)

11.2 Internal pointers to members

If a user explicitly (and erroneously) marks as **trivially relocatable** a class with an invariant that stores a pointer into an internal structure, then relocation will typically result in UB. For example:

```
class MyClass
trivially_relocatable
{
private:
    int  data_v[2];
    int *data_p;     // data_p will not be valid after a trivial relocation.
public:
    MyClass(int a, int b)
    {
        data_v[0] = a;
        data_v[1] = b;
        data_p    = &(data_v[1]);
    }
    MyClass(MyClass &&other)
    {
        data_v[0] = other.data_v[0];
        data_v[1] = other.data_v[1];
        data_p    = &(data_v[1]);   // NOT copied from other.data_v !!!
    }
};
```

After trivial relocation, `data_p` in the relocated object would point to the address where the member of the old object resided, but that object's lifetime has now ended. UB occurs for any use of that pointer now, other that assigning a new value, or destruction.

Note that this cannot happen without the user explicitly marking the class as **trivially relocatable**, as the default rules for **trivial relocatability** handle this use case by requiring only implicitly defined move constructors.

11.3 Active element of a union

When a union is trivially relocated, the active element of the union must follow along, or it would be undefined behavior to access the relocated active element. As compilers typically do not explicitly track the active member, it is thought that this would have minimal impact on implementations. However, for the purpose of static analysis, or compilers seeking undefined behavior to exploit for optimizations, it is necessary to add the guarantee to propagate the active element through the "compiler magic" in `trivial_relocate` function. Note that this guarantee must apply to non-static data members that are unions too, including anonymous unions and variant data members.

11.4 ABI compatibility

We do not anticipate any ABI compatibility concerns, but have been surprised before. Once the incubator is happy to forward this proposal to evolution, we will ask the ABI group for their opinion to better inform this part of the paper.

We deliberately avoid applying the `trivially_relocatable` trait to the standard library, deferring that work to a separate paper once the ABI implications are properly understood.

11.5 Relocating `const`-objects

The specification for a trivially relocatable type supports const-qualified types, including const-qualified class types. However the `trivially_relocate` function itself is constrained to exclude ranges of `const` objects.

The key concern is that destroying non-const objects with automatic, static, or thread storage duration is valid, as long as those objects are replaced before their destruction in invoked. However, it is undefined behavior to replace a `const` object with such a storage duration in the same manner (6.7.2 [intro.object]p10).

In order to protect from accidentally triggering UB, the special function to trivially relocate objects accepts only non-`const` qualified object. If the user knows they are dealing with objects of dynamic storage duration, they can cast away constness before the call with a `const_cast`, but must do so explicitly, acknowledging their intent.

Similarly, `const`-qualified non-static data members satisfy the definition of trivially relocatable, so do not disqualify class types with such non-static data members from also being trivially relocatable, and the complete object can easily (and safely) be relocated without requiring a `const`-cast. This is the same behavior that is supported for references as non-static members.

11.6 Trivially relocatable is not trivially swappable

One popular idea for optimization is to optimize `std::swap` with a sequence of bitwise relocations. Benchmarks have demonstrated a useful performance boost in standard algorithms that make heavy use of `swap` when we try this.

Unfortunately, the semantics of `swap` have issues beyond the object lifetimes addressed by this paper. In particular, replacing objects in place, as would be done by swap, relies on the principle of *transparently replaceable* objects (6.7.3 [basic.life]p8). Note that the term pertains to objects, not to types.

In particular, potentially overlapping objects cannot be transparently replaced. Common examples of such overlapping objects are non-static data members and base class subobjects of a complete object. `swap` works on such subobjects today as it generally uses assignment to exchange values, not construction in place.

The transparently replaceable property sits outside the type system, so is not amenable to dispatching on type-based traits such as `is_trivially_relocatable` or even a hypothetical `is_trivially_swappable` trait. Note that this same concern applies to trivially relocating data members and base class subobjects in general, even using the trivial relocation facility proposed by this paper.

As this paper is focused on introducing a very specific relocation semantic, based on decades of field experience, we keep this paper tightly focused on that well understood domain, deferring any further discussion of optimizing features like `swap` to another paper that can properly explore its particular concerns.

12 Alternative designs

There were a couple of other directions we considered before landing on the final proposal. We record them here for reference, in case anyone else thinks of these approaches, and wonders if we considered them, or why we rejected them.

12.1 Smarter default for dependent templates

An alternative design we considered for the `trivially_relocatable` specifier lacking a predicate is that, rather than defaulting to `true`, the predicate would default to `(std::is_trivially_relocatable_v<PACK> && ...)` where `PACK` would be a template parameter pack comprising the (potentially empty) set of types of any dependent bases and non-static data members. Hence, `trivially_relocatable` would be a "make me trivially relocatable if possible" request for class templates, rather than forcing an error on instantiation. It would still be an error to mark a class template having non-depended bases or non-static data members that were, in turn, not trivially relocatable.

We rejected this design as likely to be confusing, ascribing multiple possible meanings to the simple `trivially_relocatable` specifier.

12.2 Ignoring `trivially_relocatable` like `constexpr`

To simplify working with class templates, we considered treating a `trivially_relocatable` specifier that evaluates to `true`, including the default case where the predicate is implicitly `true`, like `constexpr` where it is simply ignored at instantiation time if that class template cannot be made trivially relocatable. This would still be expected to diagnose non-dependent reasons for failure eagerly though, like `static_assert`.

We rejected this direction for additional complexity, and breaking the principle of least astonishment where the value of a `trivially_relocatable` specifier can be relied on as accurate.

13 FAQ

13.1 Is `void` trivially relocatable?

No, and it is not trivially copyable either.

13.2 Are reference types trivially relocatable?

No, and they are not trivially copyable either.

13.3 Why not?!

It is not possible to take the address of a reference to pass it to `relocate`. How the compiler implements references is entirely unspecified, and may not need physical storage if the reference never leaves a local scope. As it is not meaningful to ask about copying/relocating a naked reference, rather than the entity it refers to, these trivial properties are `false`.

13.4 Why can a class with a reference member be trivially relocatable?

For the same reason such a class can be trivially copyable. Strictly speaking, reference members are not non-static data members, and you cannot create a pointer-to-data-member to one. They deliberately fall through the relevant wording by not appearing in the list of disallowed entities, despite not being trivially copyable/relocatable as a distinct type in their own right. This is subtle wording for the unwary, but has been standard practice for many a year.

13.5 Are *cv*-qualified types, notably `const` types, trivially relocatable?

Yes, if the unqualified type is trivially relocatable.

13.6 Can const-qualified types be passed to `trivially_relocate`?

No, see Relocating `const`-objects. While const-qualified types are trivially relocatable, and so do not inhibit the trivial relocatability of a wrapping type, they are typically not safe to relocate due to leaving behind a dead object that cannot be replaced using well-defined behavior. Hence, the `trivially_relocate` function is constrained to exclude const-qualified types. This can be worked around using `const_cast` if doing so would not introduce undefined behavior

13.7 Can non-implicit-lifetime types be trivially relocatable?

Yes. See New semantics

13.8 Why are virtual base classes not trivially relocatable?

As they are not trivially copyable either. We believe it is possible to implement virtual bases such that trivial copy and relocatability would not be a concern, as all the runtime fix-ups can be resolved in the initial object construction. However, it is not clear that all implementations use such a layout, and forcing trivial operations may be an ABI break.

We would love to remove this restriction, but this should be kept consistent with the corresponding restriction on trivially copyable. If no current ABIs are affected we might consider normatively allowing, or even encouraging, such an implementation (for both trivialities) as conditionally supported behavior on platforms that would not incur an ABI break.

13.9 Why do deleted special members inhibit implicit trivial relocatability?

Initially we considered allowing trivial relocation of types with these special members functions deleted, based on the notion that we are familiar with the idea since C++17, where “mandatory copy elision” started propagating non-copy/movable return values. However, relocation is not the same as the initial construction occurring at a different location (copy elision), so there were objections to the idea that when a user deliberately removes an operation, we should not *silently* re-enable it by a back door. Note that this changes only the default, preventing accidental relocation of non-copyable non-movable types for which relocatability was neither considered nor intended — if trivial relocatability is desired, such classes can be made **explicitly trivially relocatable** by means of the `trivially_relocatable` keyword.

This design also follows that of the core language for trivial copyability, which was changed to exclude types that deleted all copying operations in C++17 ([CWG1734]).

14 Proposed wording

All changes are relative to [N4928].

Editors' note: Uncompleted wording tasks:

- complete specification for `trivially_relocate` (currently a half-specified mess)
- complete specification for `uninitialized_move_and_destroy`

14.1 Feature macros

In 15.11 [cpp.predefined] add the following macro:

```
__cpp_trivial_relocatability                    TBD
```

Amend 17.3.2 [version.syn] with the following macro to the header `<version>`.

```
#define __cpp_lib_trivially_relocatable         TBD     // also in <memory>, <type_traits>
```

14.2 Grammar for `trivially_relocatable`

Add `trivially_relocatable` to the list of *Table 5: Identifiers with special meaning* ([tab:lex.name.special] in 5.10 [lex.name])

Change the grammar in 11.1 [class.pre] to add **class-context-seq**, **class-context-keyword**, **class-triv-reloc-spec** and **class-triv-reloc-expr** as follows:

```
class-head:
- class-key attribute-specifier-seq_opt class-head-name class-virt-specifier_opt base-clause_opt
- class-key attribute-specifier-seq_opt base-clause_opt
+ class-key attribute-specifier-seq_opt class-head-name class-context-seq_opt base-clause_opt
+ class-key attribute-specifier-seq_opt class-triv-reloc-expr_opt base-clause_opt

+ class-context-seq:
+        class-context-keyword class-context-seq_opt

+ class-context-keyword:
+        class-virt-specifier
+        class-triv-reloc-spec

+ class-triv-reloc-spec:
+        trivially_relocatable
+        class-triv-reloc-expr

+ class-triv-reloc-expr:
+        trivially_relocatable ( constant-expression )
```

Add the following paragraph before 11.1 [class.pre]p5:

Each form of **class-context-keyword** shall appear at most once in a complete **class-context-seq**.

In a **class-triv-reloc-expr**, the **constant-expression**, shall be a contextually converted constant expression of type bool (7.7 [expr.const]). The **class-triv-reloc-spec**$_{\text{opt}}$ `trivially_relocatable` without a **constant-expression** is equivalent to the **class-triv-reloc-spec**$_{\text{opt}}$ `trivially_relocatable(true)`.

EDITORS' NOTE: We probably need to add something similar to p5, or revise p5 to use the new grammar term class-context-seq instead, and extend the example.

14.3 Specification of trivial relocatability

Append to 6.8.1 [basic.types.general]p9:

Scalar types, trivially relocatable class types, and arrays of such types, are collectively called **trivially relocatable types**.

Add the following paragraphs to 11.2 [class.prop]:

A **trivially relocatable class** is a **class** that:

- has a **class-triv-reloc-spec** without a **constant-expression**,
- has a **class-triv-reloc-expr** with a **constant-expression** that evaluates to `true`,
- or satisfies all of the following
  - has no base classes that are not of trivially relocatable type,
  - has no non-static non-reference data members whose type is not a trivially relocatable type,
  - has no virtual base classes,
  - has a selected destructor that is neither user-provided nor deleted,
  - has no **class-triv-reloc-expr** with a **constant-expression** that evaluates to `false`,
  - would, when an instance of the type is direct initialized from an rvalue of the same type, select a constructor that is neither user provided nor deleted.

[Note: accessibility of the special member functions is not relevant — end note]

[Note: trivially copyable class types are implicitly trivially relocatable unless they have a `trivially_relocatable` predicate that evaluates to `false` — end note]

[Note: a type with const-qualified or reference members can be trivially relocatable — end note]

[Note: lambdas are trivially relocatable if and only if their closure type is a trivially relocatable class type — end note]

A class type having **class-triv-reloc-spec** `trivially_relocatable` or **class-triv-reloc-expr**$_{\textbf{opt}}$ `trivially_relocatable` with value `true` specifies that it shall be considered **trivially relocatable** per the proposed definition in 6.8.1 [basic.types.general].

The program is ill-formed if a class with a **class-triv-reloc-spec** whose **constant-expression** is absent or evaluates to `true` has either

- a virtual base class,
- a base class that is not **trivially relocatable**, or
- a non-static data member of non-reference type that is not **trivially relocatable**.

Design note: It is possible, by means of the `trivially_relocatable(true)` specification, to declare a class as trivially relocatable even if that class has user-provided special members (see proposal). Note that this cannot break the encapsulation of members or bases and allow for their trivial relocation when they, themselves, are not trivially relocatable.

14.4 New type trait

Add to the `<type_traits>` header synopsis in 21.3.3 [meta.type.synop]:

```
template< class T >
struct is_trivially_relocatable;

template< class T >
inline constexpr bool is_trivially_relocatable_v = is_trivially_relocatable<T>::value;
```

Add a new entry to table 47 in 21.3.5.4 [meta.unary.prop]

Template	Condition	Preconditions
`template<class T> struct is_trivially_relocatable;`	T is a **trivially relocatable** type	`remove_all_extents_t<T>` shall be a complete type or *cv*-`void`

14.5 Relocation functions

14.5.1 `trivially_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_const_v<T>)
constexpr
T* trivially_relocate(T* begin, T* end, T* new_location) 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 20.2.6 [obj.lifetime]:

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

Preconditions: `end` is reachable from `begin`.

[`new_location`, `new_location + begin - end`) denotes a region of allocated storage that is a subset of the region of storage reachable through (6.8.4 [basic.compound]) `new_location` and suitably aligned for the type `T`.

Effects: Implicitly creates objects (6.7.2 [intro.object]) within the denoted region consisting of an object *a* of type `T` whose address is `p`, and objects nested within *a*, as follows: The object representation of *a* is the contents of the storage prior to the call to `trivially_relocate`. The value of each created object *o* of trivially-relocatable type `U` is determined in the same manner as for a call to `bit_cast<U>(E)` (22.15.3 [bit.cast]), where `E` is an lvalue of type `U` denoting *o*, except that the storage is not accessed. The value of any other created object is unspecified.

Returns: A pointer to the *a* defined in the Effects paragraph.

Throws:: Nothing.

Remarks: The active member of any union objects or subobjects in the relocated range [`new_location, new_location + begi`
is the active member of the corresponding union objects or subobjects from the original range [`begin, end`).

Ends the lifetime of the objects in the range [`begin, end`) without running their destructors, as if the storage were reused by another object (6.7.3 [basic.life]).

[Note: A likely implementation will simply call a compiler intrinsic that calls `memmove` and updates its notion of the object lifetime. —end note]

EDITOR'S NOTE: THIS IS A LIGHTLY MASSAGED COPY OF `start_lifetime_as` SPEC AND NEEDS MORE WORK, IN PARTICULAR WRT USING `bit_cast` TO MAGICALLY IMBUE LIFE INTO NEW OBJECTS

15 Potential Library Extensions

15.0.1 `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);
```

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;
}
```

Throws: Nothing.

15.1 `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.

15.1.1 `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);
```

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;
```

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

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.

16 Acknowledgements

This document is written in markdown, and depends on the extensions in `Pandoc` and `mpark/wg21`.

The authors would also like to thank Joshua Berne for his assistance in proof reading this paper, especially the proposed Core wording. Also, this paper is greatly improved by feedback from Arthur O'Dwyer, author of [P1144R8], who corrected many bad assumptions we made about his paper, and helped bring the technical differences into focus. We also benefited from several examples he shared to help illustrate those differences and misunderstandings.

17 References

[CWG1734] James Widman. 2013-08-09. Nontrivial deleted copy functions.
https://wg21.link/cwg1734

[N4928] Thomas Köppe. 2022-12-18. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4928

[P1029R3] Niall Douglas. 2020-01-12. move = bitcopies.
https://wg21.link/p1029r3

[P1144R6] Arthur O'Dwyer. 2022-06-10. Object relocation in terms of move plus destroy.
https://wg21.link/p1144r6

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

[P2685R0] Alisdair Meredith, Joshua Berne. 2022-10-15. Language Support For Scoped Allocators.

https://wg21.link/p2685r0

[P2814R0] Mungo Gill, Alisdair Meredith; Arthur O‘Dwyer. 2023-05-19. Trivial Relocatability --- Comparing P1144 with P2786.
https://wg21.link/p2814r0

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