Trivial Relocatability For C++26

Proposal for an alternative approach to trivial relocatability

Document #:	P2786R4
Date:	2024-02-09
Project:	Programming Language C++
Audience:	EWG, LEWG

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

Contents

1 Abstract

This paper offers an approach to support *trivial relocatability*, i.e., moving objects in memory by copying their byte representation, 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 a more rigorous design and specification.

2 Revision History

R4: February 2024 (Tokyo mailing)

- Update wording previously relative to N4958 to being relative to N4971
- Additional section showing benchmarks from implementation experience

R3: October 2023 (midterm mailing)

- Targeted at EWG, with a follow-up for LEWG
- Retitled as a main proposal
- Moved library extensions to [P2959R0] and [P2967R0]
- Made formatting, section ordering, and layout fixes
- Included additional motivating use cases
- Reworked the usage examples given the previous changes
- Cleaned up and finalized the (previously draft) proposed grammar diffs and wording
- Added Open Questions section
- Removed `constexpr` from the proposed `trivially_relocate` function

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 (subsequently removed in R3)

R1: May 2023 (pre-Varna mailing)

Midterm mailing following feedback from Issaquah.

The most significant change is that we moved analysis and comparisons with [P1144R6] to a separate coauthored 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
  - Required forward iterators for input range, as we expect to modify and/or destroy elements
  - Added a full family of overloads, consistent with `uninitialized_*` standard algorithms
- Reviewed use of *voidify* as consistent with library text with Issaquah papers applied
- Provided a complete specification for `relocate` that handles overlapping ranges
- Revised concerns with application to `swap`, deferring any further work to a separate paper
- Struck redundant `inline` from definition of `is_trivially_relocatable_v`

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 be semantically equivalent to a move construction immediately followed by a destruction of the source object (though exceptions, while not encouraged, are not expressly forbidden).

Any trivially copyable type is *trivially relocatable* by default. Many other types, even those that have nontrivial 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 `std::vector`, `std::unique_ptr`, and `std::shared_ptr`.

Note that simply doing a bitwise copy of these objects that are not trivially copyable 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 that 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.

Throughout this paper, a **bold typeface** will be used for terms defined herein, with the exception of the proposed wording, and for which the conventions used in the Standard will apply.

4 Motivating Use Cases

4.1 Efficient `vector` growth

Suppose we have a move-only type, `class MoveOnlyType` (e.g., a unique ownership smart pointer), and we wish to hold a `std::vector` of these types, `std::vector<MoveOnlyType>`. Simply emplacing five of these objects would require that `MoveOnlyType`'s move constructor and destructor be called seven 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 five emplacements would require only three `memmove` operations, with no additional calls to `MoveOnlyType`'s move constructor and destructor.

For this example, we are assuming that we have 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 nonallocating empty state and thus 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 nonthrowing move constructor, `vector`s of such `list`s 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 the program does not have to restore the dying object into a destructible state; there is no window of opportunity to access the dying object in an invalid state.

4.3 Moving in-place or small-buffer-optimized type-erased types

Trivial relocation can be used to deduplicate the code generated by type-erasing wrappers like `any`, `function`, and `move_only_function`. For these types, a move of the wrapper object is implemented in terms of a relocation of the contained object. (See, for example, libc++'s `std::any`.) In general, the relocate operation must have a different instantiation for each different contained type `T`, leading to code bloat. But every **trivially relocatable** `T` of a given size can share the same instantiation.

Note: This use case was originally stated in [P1144R7] but is also applicable to this proposal.

4.4 Moving fixed-capacity containers like `static_vector` and `small_vector`

The move constructor of `fixed_capacity_vector<R,N>` can be implemented naïvely as an element-by-element move (leaving the source `vector`'s elements in their moved-from state) or, for suitable types `R`, more efficiently as a trivial relocation (ending the lifetime of the source `vector`).

Note: `boost::container::static_vector<R,N>` currently implements the naïve element-by-element move strategy, but after LEWG feedback, `static_vector` as proposed in [P0843R5] does permit the faster relocation strategy.

Note: This use case was originally stated in [P1144R7] but is also applicable to this proposal.

4.5 `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 as allocated memory does not reside within the owning object itself, many nontrivial allocator-aware types can be trivially relocatable if syntax to express this property is made available in the language.

4.6 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 and in an experimental branch to explore language extensions, `pbastd::is_trivially_relocatable` was used to demonstrate relocation of types using `std::pmr::polymorphic_allocator`. This experimental model was a pure library extension and was therefore unable to take advantage of any of the compiler features proposed in this paper. 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. In this experimental model, the new trait detects trivially copyable types as also being trivially relocatable by default, while other types default to being not trivially relocatable. This part can be covered by a library emulation, implementing the new trait in terms of `std::is_trivially_copyable`. Note that user specialization would not be permitted for a standardized type trait (per 21.3.2 [meta.rqmts]p4) without explicit permission from the Standard, which is not something granted for any other traits that reflect the value of core-language properties of a type.

6 Implementation Experience

Corentin Jabot successfully implemented this in clang, and performed benchmarks of vector growth in libc++. For details see Optimize vector growing of trivially relocatable types.

```
--------------------------------------------------
Benchmark                             old       new
--------------------------------------------------
bm_grow<int>                      1354 ns    1301 ns
bm_grow<std::string>              5584 ns    3370 ns
bm_grow<std::unique_ptr<int>>     3506 ns    1994 ns
bm_grow<std::deque<int>>         27114 ns   27209 ns
```

7 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, defining the operation we wish to optimize is sufficient.

— **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 not to be **trivially relocatable** by means of the `trivially_relocatable` keyword with value `false`.

— **implicitly trivially relocatable**: A type is **implicitly trivially relocatable** if its (selected) destructor is neither user provided nor deleted, it has no virtual base classes, all its base classes (if any) are trivially relocatable, all its nonstatic nonreference data members (if any) are trivially relocatable, and the constructor selected for *direct-non-list-initialization* from a single xvalue 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` versus `protected` versus `private`) has no bearing on whether that class is **implicitly trivially relocatable**. This lack of 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 be neither deleted nor user provided.

  — The copy constructor is irrelevant unless it inhibits the declaration of the move constructor; then a class is not **implicitly trivially relocatable** unless the copy constructor is implicitly defined (i.e., defaulted on its first declaration).

  — 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.

  — Note that the selected destructor does not need to be *trivial*, as we may have **trivially relocatable** members or bases that have nontrivial destructors.

— **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 nonstatic data members that are not **trivially relocatable** nor any base classes that are not **trivially relocatable**, nor may it have any virtual base classes. I.e., adding the keyword with value `true` to a class that does not qualify is a diagnosable error.

Note that we are proposing to permit, by means of the `trivially_relocatable` keyword, types that would otherwise be noncopyable and nonmovable to be (trivially) **relocatable**. For this reason, we cannot define **relocate** and **relocatable** in terms of a move construction followed by a destruction. The ability to explicitly make a type trivially relocatable enables providing a customized (and thus nontrivial) move constructor and destructor while declaring that the compound operation is trivial.

8 Proposed Language Changes

Our proposal changes and extends C++26 as follows.

8.1 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 nonstatic 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**.

8.2 New semantics

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. Because the compiler cannot know if a specific `memcpy` or `memmove` call is intended to duplicate or to move an object, we propose introducing a new function template that is restricted to **trivially relocatable** types. The purpose of the new function template is to call `memmove` on our behalf and to also signify to the compiler and other source analysis tools that the lifetime of the new object(s) has begun — similarly to calling `start_lifetime_as` on the destination location(s) — and that the lifetime of the original object(s) has ended.

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 users are not permitted to copy the bytes to perform a relocation themselves, unlike with trivial copyability, although byte copies would continue to work for trivially copyable types.

8.3 New syntax

To enable trivial relocatability to be useful for more complicated types (i.e., those that are not **trivially copyable**), explicitly marking types that are **trivially copyable** as **trivially relocatable** must be possible. 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:

```
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**
  - with value `false` to explicitly remove **trivial relocatability** from a class

The Boolean predicate is optional, with a plain `trivally_relocatable` specifier 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. Two notable implications of this approach are worth highlighting.

- 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.

- Rendering, by means of the keyword and value `false`, any type — even a **trivially copyable** type — to not be **trivially relocatable** is possible.

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. We clearly saw that reasoning about our examples was much simpler when the trivial relocation specification could be trusted to mean literally what it said.

For an example of where this may be useful in practice, see the small buffer optimization example in "Contingent trivial relocatability" below.

8.3.1 Diagnosable errors

In a nondependent context, marking a type as **trivially relocatable** would be a diagnosable error if the type has a virtual base class or if it comprises any bases or nonstatic members that are not **trivially relocatable**.

See also "Why are virtual base classes not trivially relocatable?" below.

8.4 New type trait

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;
```

The `std::is_trivially_relocatable<T>` trait has a base characteristic of `std::true_type` if `T` is **trivially relocatable** and `std::false_type` otherwise.

Note that the `std::is_trivially_relocatable` trait reflects the underlying property that a type has, and like all similar traits in the Standard Library, it must not be *user specializable*. Compilers themselves are expected to determine this property internally and should not introduce a library dependency such as by instantiating this type trait.

Note that we expect 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).

8.5 New relocation function `trivially_relocate`

As stated in "New semantics" (above), we are proposing a new function, `trivially_relocate`, which is the unique entry point into the core magic that tracks and manages object lifetimes in the abstract machine.

Note that this initial proposal does not provide a single-object relocation function as our primary motivation is to optimize relocating objects in bulk, which is expected to be the common use case. Adding single-object `trivially_relocate` functions would be easy, 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:

```
template <class T>
   requires (is_trivially_relocatable_v<T> && !is_const_v<T>)
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`. Unlike `memmove` on its own, this function template is restricted to trivially relocatable types rather than to implicit lifetime types.

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.

In addition to performing `memmove`, the function also has the following two important effects that matter to the abstract machine but have no apparent physical effect (i.e., these effects do not change bits in memory), much like `std::launder`.

— The `trivially_relocate` function ends the lifetime of the objects `*begin`, `*(begin+1)`, …, through to `*(end-1)`. This means accessing these objects or attempting to destruct any of them will be *undefined behavior*.

— The `trivially_relocate` function 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)`.

Note that the second item — i.e., 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 currently no mechanism to just end the lifetime(s) of the source object(s) without performing other actions.

Note that the above example should *not* be interpreted as indicating an intention to create a new Standard Library function `start_lifetime_as_array_without_preconditions`. The above equivalence is shown for illustration purposes only.

The existing `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. The copying and imbuing life *must* 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.

The `trivially_relocate` function is intended to support overlapping source and destination ranges, just like `memmove`. In the event that the ranges are overlapping, then care needs to be taken around the management of the lifetime of objects relocated out of or into the overlap.

Finally, observe that this function is *not* `constexpr`. The reasons behind this are discussed in detail below — see "`constexpr` support for `std::vector` and `std::string`".

8.6 Examples of use

8.6.1 Simple example without predicate

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

```
struct BaseType {
   // simple base class, trivially relocatable by default
};

struct MyRelocatableType trivially_relocatable : BaseType {
   // class definition details

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

struct MyNonRelocatableType : BaseType {
   // class definition details

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

static_assert( is_trivially_relocatable_v<MyRelocatableType>   );
static_assert(!is_trivially_relocatable_v<MyNonRelocatableType>);
```

8.6.2 Simple examples with predicate

The Boolean predicate form, `trivially_relocatable(true)`, can also be used to opt into trivial relocatability. Alternatively, `trivially_relocatable(false)` can be used to opt out of the behavior for a type that might otherwise be **trivially relocatable** by default.

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

```
struct Relocatable    trivially_relocatable(true )
{
  // trivially relocatable
};

struct NonRelocatable trivially_relocatable(false)
{
  // not trivially relocatable
};

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

Clearly, `Relocatable` is a trivially relocatable class type, and `NonRelocatable` is not a trivially relocatable class type. We will use these classes to illustrate how similar but subtly different class templates then behave.

We can 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 Example1 {
  TYPE data;
};

static_assert( is_trivially_relocatable_v<Example1<Relocatable>>   );
static_assert(!is_trivially_relocatable_v<Example1<NonRelocatable>>);
```

We can also use `trivially_relocatable(false)` to remove trivial relocatability from the template class:

```
template<class TYPE>
class Example2 trivially_relocatable(false)
{
private:
  TYPE data;
};

static_assert(!is_trivially_relocatable_v<Example2<Relocatable>>   );
static_assert(!is_trivially_relocatable_v<Example2<NonRelocatable>>);
```

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

8.6.2.1 Demonstrating trivial relocatability of dependent types

However, an important 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. In this example, we are concerned with the case of a class template that provides its own special members and thus needs to supply a trivial relocation specification to forward the trivial relocatability of its dependent members:

```
struct Relocatable    trivially_relocatable(true )
{
  // trivially relocatable
};

struct NonRelocatable trivially_relocatable(false)
{
  // not trivially relocatable
};

template<class TYPE>
class Example3 trivially_relocatable(is_trivially_relocatable_v<TYPE>)
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~Example3() {}  // user-provided destructor, so not implicitly relocatable
};

static_assert( is_trivially_relocatable_v<Example3<Relocatable>>);
static_assert(!is_trivially_relocatable_v<Example3<NonRelocatable>>);
```

The examples look simple and may lead us to think, “Why am I messing with all this template syntax when the simple `Example` works?” We must remember that 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.

8.6.2.2 Example of limiting instantiation based on trivial relocatability

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

```
struct Relocatable    trivially_relocatable(true )
{
  // trivially relocatable
};

struct NonRelocatable trivially_relocatable(false)
{
  // not trivially relocatable
};

template<class TYPE>
   requires is_trivially_relocatable_v<TYPE>
class Example4 trivially_relocatable
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~Example4() {}  // user-provided destructor, so not implicitly relocatable
};

static_assert(
   is_trivially_relocatable_v<Example4<Relocatable>>   ); // well formed
static_assert(
  !is_trivially_relocatable_v<Example4<NonRelocatable>>); // ill formed
```

Observe that the static assertion for `Example4<NonRelocatable>` is ill formed not because that `static_assert` fails, but rather because the `Example4` template cannot be instantiated for `NonRelocatable` at all; i.e., `Example4` is a template that wraps only trivially relocatable types and can thus can guarantee that it is always trivially relocatable.

8.6.3 Examples of diagnosable errors

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

```
struct Relocatable    trivially_relocatable(true )
{
  // trivially relocatable
};

struct NonRelocatable trivially_relocatable(false)
{
  // not trivially relocatable
};

template<class TYPE>
class Example5 trivially_relocatable
{
private:
  TYPE value_a;
  TYPE value_b;
public:
  ~Example5() {}  // user-provided destructor, so not implicitly relocatable
};

static_assert( is_trivially_relocatable_v<Example5<Relocatable>>   );
static_assert(!is_trivially_relocatable_v<Example5<NonRelocatable>>); // ill formed
```

The `Example5` 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, whereas the error message for `Example4` in the previous example would be related to violating the type constraints of the `requires` clause.

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

```
template<class TYPE>
struct Erroneous trivially_relocatable
{
  NonRelocatable 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.

8.6.4 Contingent trivial relocatability

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

```
template<class T>
class Container trivially_relocatable(is_trivially_relocatable_v<T>() ||
                                      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 we make no effort to further constrain on type. This might be how we approach retrofitting trivial relocatability into an existing library without raising ABI concerns.

9 Design Choices

9.1 No library support is mandated by *this* paper

This extension is intended to be fully backward compatible, and other than the introduction of one trait (`is_trivially_relocatable_v`) and one utility function (`trivially_relocate`), no library changes are *required.* Library implementers may, if they so desire, take advantage of this feature to improve performance, but they are not mandated to do so. We firmly believe that the behavior we are enabling reflects existing use cases where users risk undefined behavior that "just works" today.

We defer library discussion to two further papers targeting LEWG. The first is [P2959R0], which addresses additional library concerns with the specification of `std::vector` (and other *block-based* containers) that would limit the applicability of trivial relocation within the Standard Library. The second paper [P2967R0] will more broadly address the idea of how the Standard Library should support relocation in general, proposing new relocation functions and proposing a policy for whether Standard Library classes should specify whether they are trivially relocatable and, if so, in which circumstances or whether trivial relocatability is purely a QoI concern as an optimization for library vendors.

9.2 `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.

The possible introduction of range functions into the Standard Library is not discussed here as it will be covered in two subsequent papers, [P2959R0] and [P2967R0].

9.3 Type trait vs. concept

Existing library facilities in this space, such as those 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.

Specifying a concept in terms of the proposed trait would be simple, but the keyword 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.

9.4 Contextual keyword vs. attribute

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

We expect templates to make use of the `trivially_relocatable` syntax to express trivial relocatability and prefer to avoid the extra work of 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 how a pack expansion would work 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.

One benefit of using an attribute would be 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 parenthesis of the Boolean expression.

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!
                                  // Is Bar the predicate of trivially_relocatable?

struct Foo trivially_relocatable (true) { };
struct Foo trivially_relocatable (true) (Bar) { };
                                  // Always a syntax error today,
                                  // but maybe we'd like to use this tomorrow.
```

Note that this syntax 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. Alternatively it would be possible to specify that

```
struct Foo trivially_relocatable (Bar)
```

is always interpreted as `Bar` being the predicate for `trivially_relocatable`, not a separate parenthesized annotation. Those wanting it to be a novel, separate parenthesized annotation would then be required to use

```
struct Foo trivially_relocatable(true) (Bar)
```

We are not aware of any proposals for such an extension at this point, so the above example is an entirely hypothetical demonstration to show that this proposal would not preclude the use of such a syntax in the future.

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

One of the motivations behind this proposal was a desire to support simple and practical implementations of `vector`-like types that wish to optimize on the availability of trivial relocation. As both `vector` and `basic_string` are usable in constant expressions since C++20, this implies a desire to support the relocation of objects in transient dynamic storage during constant evaluation to avoid unnecessary `if consteval` magic in their implementations.

As will be discussed in library papers [P2959R0] and [P2967R0], this is achievable without any need for the `trivially_relocate` function itself to be `constexpr`, by placing the `if consteval` logic into a more generic relocation relocation function (proposed for the Standard Library) that *is* declared as `constexpr`.

In addition, our implementation experience has shown that there would be considerable challenges around enabling the `trivially_relocate` function to be used during constant evaluation.

10 Open Questions

10.1 Most Vexing Parse

The `trivially_relocatable` keyword is subject to the Most Vexing Parse. This has not been addressed in the current version of this paper.

An outstanding question is whether the proposed grammar should be adjusted to cater for this scenario, an example of which would be:

```
struct A trivially_relocatable(bool(my_constexpr_value)) {};
```

10.2 Feature test macros

The proposed grammar has two feature test macros:

```
__cpp_trivial_relocatability
```

and

```
__cpp_lib_trivially_relocatable
```

An open question is whether we require both of these.

11 Known Concerns

11.1 Separately managed objects

Performing trivial relocations is generally inappropriate 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 nonstatic data member within a class. Adding compiler support to better observe trivial relocations means we might get warnings on such misuse. (This concern is similar to destroying and recreating an object in-place. In such cases, recreating the object before its destructor will be called implicitly is essential — hence a warning and not an error since 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:

```
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 than assigning a new value, or for destruction.

Note that **trivial relocation** cannot happen without the user explicitly marking the class as **trivially relocatable**, because the default rules for **implicit trivial relocatability** handle this use case by requiring that move constructors not be user provided.

11.3 Active element of a union

When a union is trivially relocated, the active element of the union must follow along, as accessing the relocated active element would be UB. Because compilers typically do not explicitly track the active member except during constant evaluation, we think this requirement would have minimal impact on implementations. However, for the purpose of static analysis or for compilers seeking undefined behavior to exploit for optimizations, adding the guarantee to propagate the active element through the "compiler magic" in the `trivially_relocate` function is necessary. Note that this guarantee must apply to nonstatic 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, as this does not require a change to the name mangling of any types marked as trivially relocatable and there is no change in object layout. Our initial implementations have confirmed this.

We deliberately avoid applying the `trivially_relocatable` trait to the Standard Library, deferring that work to future library papers.

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 is invoked. However, replacing a `const` object with such a storage duration in the same manner is UB (6.7.2 [intro.object]p10).

To protect from accidentally triggering UB, the special function to trivially relocate objects accepts only objects that are not `const` qualified objects. If the user knows they are dealing with objects of dynamic storage duration, they can cast away `const`ness before the call with a `const_cast` but must do so explicitly, acknowledging their intent.

Similarly, `const`-qualified nonstatic data members satisfy the definition of trivially relocatable and thus do not disqualify class types with such nonstatic 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 nonstatic members.

11.6 Trivially relocatable is not trivially swappable

For optimization, one popular idea 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 nonstatic 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 it 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 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

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

12.1 A smarter default for dependent templates

For the `trivially_relocatable` specifier lacking a predicate, we considered an alternative design with a predicate that, rather than defaulting to `true`, 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 nonstatic 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. Marking a class template having nondependent bases or nonstatic data members that were, in turn, not trivially relocatable would still be an error.

We rejected this design, as it would ascribing multiple possible meanings to the simple `trivially_relocatable` specifier, on the basis that this is likely to cause confusion.

Further, the user is able to write exactly that requirement as a fold expression themselves if that is the semantic they desire; if the folded constraint were the default, it would be much harder for a user to achieve the semantic we are proposing.

12.2 Ignoring `trivially_relocatable` like `constexpr`

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

We rejected this direction because it added complexity and broke 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, nor is it trivially copyable.

13.2 Are reference types trivially relocatable?

No, nor are they trivially copyable.

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

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

A class with a reference member can be trivially relocatable for the same reason such a class can be trivially copyable. Strictly speaking, reference members are not nonstatic 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 or trivially relocatable as a distinct type in their own right. This is subtle wording for the unwary but has been standard practice for many years.

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

Yes, if the unqualified type is trivially relocatable.

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

No, see "Relocating `const`-objects". While `const`-qualified types are trivially relocatable and thus 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

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

Yes, see "New semantics".

13.7 Why are virtual base classes not trivially relocatable?

Because they are not trivially copyable and because the implementation of virtual base classes on some platforms involves an internal pointer, virtual base classes are not trivially relocatable.

We believe that implementing virtual bases such that trivial copyability and relocatability would not be a concern is possible, as all the runtime fix-ups can be resolved in the initial object construction. However, whether all implementations use such a layout is unclear, and forcing trivial operations may be an ABI break.

We are of the opinion that this low-level behavior should be kept 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.

We would be happy to remove this restriction, but it must 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.

Note that no issues occur 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.

13.8 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 a notion that we have been familiar with since C++17, when "mandatory copy elision" started propagating noncopyable and nonmovable return values. However, relocation is not the same as copy elision, so objections arose 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 inhibition changes only the default, preventing accidental relocation of noncopyable or nonmovable 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 by [CWG1734], which landed in C++17.

14 Proposed Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4971], the latest draft at the time of writing.

14.1 Add new identifier with a special meaning

5.10 [lex.name] Identifiers

Table 4: Identifiers with special meaning [tab:lex.name.special]

`final`	`import`	`module`	`override`	`trivially_relocatable`

14.2 Specify trivially relocatable types

Editorial note: *We have separated each sentence to improve clarity rather than trying to identify the definition of so many terms as a single paragraph.*

6.8.1 [basic.types.general] General

9 Arithmetic types (6.8.2 [basic.fundamental]), enumeration types, pointer types, pointer-to-member types (6.8.4 [basic.compound]), `std::nullptr_t`, and cv-qualified (6.8.5 [basic.type.qualifier]) versions of these types are collectively called *scalar types*.

Scalar types, trivially copyable class types (11.2 [class.prop]), arrays of such types, and cv-qualified versions of these types are collectively called *trivially copyable types*.

Scalar types, trivial class types (11.2 [class.prop]), arrays of such types, and cv-qualified versions of these types are collectively called *trivial types*.

Scalar types, trivially relocatable class types (11.2 [class.prop]), arrays of such types, and cv-qualified versions of these types are collectively called *trivially relocatable types*.

Scalar types, standard-layout class types (11.2 [class.prop]), arrays of such types, and cv-qualified versions of these types are collectively called *standard-layout types*.

Scalar types, implicit-lifetime class types (11.2 [class.prop]), array types, and cv-qualified versions of these types are collectively called *implicit-lifetime types*.

14.3 Address trivial relocation of lambdas

7.5.5.2 [expr.prim.lambda.closure] Closure types

3 The closure type is declared in the smallest block scope, class scope, or namespace scope that contains the corresponding *lambda-expression*.

[*Note 1:* This determines the set of namespaces and classes associated with the closure type (6.5.4 [basic.lookup.argdep]). The parameter types of a *lambda-declarator* do not affect these associated namespaces and classes. —*end note*]

The closure type is not an aggregate type (9.4.2 [dcl.init.aggr]) and not a structural type (13.2 [temp.param]). An implementation may define the closure type differently from what is described below provided this does not alter the observable behavior of the program other than by changing:

((3.1)) — the size and/or alignment of the closure type,

((3.2)) — whether the closure type is trivially copyable (11.2 [class.prop]), or

((3.x)) — whether the closure type is trivially relocatable(11.2 [class.prop]), or

((3.3)) — whether the closure type is a standard-layout class (11.2 [class.prop]).

An implementation shall not add members of rvalue reference type to the closure type.

14.4 Update grammar to support `trivially_relocatable`

11.1 [class.pre] Preamble

1 A class is a type. Its name becomes a class-name (11.3 [class.name]) within its scope.

> *class-name* :
>   *identifier*
>   *simple-template-id*

A *class-specifier* or an *elaborated-type-specifier* (9.2.9.4 [dcl.type.elab]) is used to make a *class-name*. An object of a class consists of a (possibly empty) sequence of members and base class objects.

> *class-specifier* :
>   *class-head* `{` *member-specification*$_{opt}$ `}`

> *class-head*:
>   *class-key attribute-specifier-seq*$_{opt}$ *class-head-name* ~~*class-virt-specifier*$_{opt}$~~ *class-context-seq*$_{opt}$ *base-clause*$_{opt}$
>   *class-key attribute-specifier-seq*$_{opt}$ *class-triv-reloc-expr*$_{opt}$ *base-clause*$_{opt}$

> *class-head-name* :
>   *nested-name-specifier*$_{opt}$ *class-name*

> *class-context-seq*:
>   *class-context-keyword class-context-seq*$_{opt}$

> *class-context-keyword*:
>   *class-triv-reloc-spec*
>   *class-virt-specifier*

> *class-triv-reloc-spec*:
>   `trivially_relocatable`
>   *class-triv-reloc-expr*

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

> *class-virt-specifier* :
>   `final`

> *class-key* :
>   `class`
>   `struct`
>   `union`

A class declaration where the *class-name* in the *class-head-name* is a *simple-template-id* shall be …

4 [*Note 2:* The *class-key* determines whether the class is a union (11.5 [class.union]) and whether access is public or private by default (11.8 [class.access]). A union holds the value of at most one data member at a time. —*end note*]

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

x 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*$_{opt}$ `trivially_relocatable` without a *constant-expression* is equivalent to the *class-triv-reloc-spec*$_{opt}$ `trivially_relocatable(true)`.

5 If a class is marked with the *class-virt-specifier* `final` and it appears as a *class-or-decltype* in a base-clause (11.7 [class.derived]), the program is ill-formed. Whenever a class-key is followed by a *class-head-name*, the identifier final, and a colon or left brace, final is interpreted as a *class-virt-specifier*.

[*Example 2:*

```
struct A;
struct A final {}; // OK, definition of struct A,
                   // not value-initialization of variable final

struct X {
  struct C { constexpr operator int() { return 5; } };
  struct B final : C{}; // OK, definition of nested class B,
                        // not declaration of a bit-field member final
};
```

—*end example*]

y Whenever a *class-key* is followed by a *class-head-name*, the *identifier* `trivially_relocatable`, and a colon or left brace, then `trivially_relocatable` is interpreted as a *class-triv-reloc-spec*.

[*Example 3:*

```
struct A;
struct A trivially_relocatable {};      // OK, definition of struct A,
                                        // not value-initialization of
                                        // variable trivially_relocatable

struct X {
  struct C { constexpr operator int() { return 5; } };
  struct B trivially_relocatable : C{};  // OK, definition of nested class B,
                                         // not declaration of a bit-field
                                         // member trivially_relocatable

};

struct D trivially_relocatable final {}; // OK, definition of struct D,
                                         // does not match other grammar
struct E trivially_relocatable(true) {}; // OK, definition of struct E,
                                         // does not match other grammar
```

—*end example*]

z 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.

6 [*Note 3:* Complete objects of class type have nonzero size. Base class subobjects and members declared with the `no_unique_address` attribute (9.12.11 [dcl.attr.nouniqueaddr]) are not so constrained. —*end note*]

14.5 Specification for trivial relocatable classes

11.2 [class.prop] Properties of classes

2 A *trivial class* is a class that is trivially copyable and has one or more eligible default constructors (11.4.5.2 [class.default.ctor]), all of which are trivial.

[*Note 1:* In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. —*end note*]

x A class `C` is a *trivially relocatable class* if it

- 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 object of type `C` is *direct-non-list-initialized* from an xvalue of `C`, select a constructor that is neither user-provided nor deleted.

[*Note A:* Accessibility of the special member functions is not relevant —*end note*]

[*Note B:* Trivially copyable classes are implicitly trivially relocatable unless they have a `trivially_relocatable` predicate that evaluates to `false` —*end note*]

[*Note C:* A type with non-static members that const-qualified or references can be trivially relocatable —*end note*]

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

3 A class `S` is a *standard-layout class* if it:

(3.1) ...

Design note:

Declaring a class as trivially relocatable is possible, by means of the `trivially_relocatable(true)` specification, even if that class has user-provided special members (see "New syntax"). Note that such a declaration is not permitted to break the encapsulation of members or bases and allow for their trivial relocation when they, themselves, are not trivially relocatable.

14.6 Add feature macros

14.6.1 15.11 [cpp.predefined] Predefined macro names

Table 22: Feature-test macros [tab:cpp.predefined.ft]

Name	Value
...	...
`__cpp_template_template_args`	201611L
`__cpp_threadsafe_static_init`	200806L
`__cpp_trivial_relocatability`	TBD
`__cpp_unicode_characters`	200704L
...	...

17.3.2 [version.syn] Header `<version>` synopsis

2 Each of the macros defined in `<version>` is also defined after inclusion of any member of the set of library headers indicated in the corresponding comment in this synopsis.

[*Note 1:* Future revisions of C++ might replace the values of these macros with greater values. —*end note*]

…

```
#define __cpp_lib_transformation_trait_aliases      201304L // freestanding, also in <type_traits>
#define __cpp_lib_transparent_operators             201510L
  // freestanding, also in <memory>, <functional>
#define __cpp_lib_trivially_relocatable             TBD     // also in< type_traits>
#define __cpp_lib_tuple_like                        202207L
  // also in <utility>, <tuple>, <map>, <unordered_map>
```

…

14.7 Add new type trait

21.3.3 [meta.type.synop] Header `<type_traits>` synopsis

```
template< class T >
struct is_trivially_relocatable;

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

21.3.5.4 [meta.unary.prop] Type properties

Template	Condition	Preconditions
`template<class T> struct is_trivially_relocatable;`	T is a trivially relocatable type (6.8.1 [basic.types.general])	`remove_all_extents_t<T>` shall be a complete type or *cv*-`void`

14.8 Specify the trivial relocation function

14.8.1 `trivially_relocate`

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

20.2.6 [obj.lifetime] Explicit lifetime management

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

a **Preconditions**:

(a.1) — `end` is reachable from `begin`.

(a.2) — [`new_location`, `new_location + (end - begin)`) 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`.

b **Effects**: Where pointers `begin` and `new_location` represent the same address or where pointers `begin` and `end` represent the same address, this call has no effect. Otherwise, the following steps are performed:

(b.1) — Perform a byte copy equivalent to `memmove(new_location, begin, sizeof(T) * (end - begin));`

(b.2) — Implicitly end the lifetime of all objects that are in the range [`begin`, `end`) but that are not in the range [`new_location`, `new_location + (end - begin)`) without running their destructors, as if the storage were reused by another object (6.7.3 [basic.life]).

(b.3) — For each of the locations `p` = `new_location + i` for all `i` in the range [0, `(end - begin)`): Implicitly create 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 implicit creation. 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.

c **Returns**: A pointer to the *a* at location `new_location` defined in the Effects paragraph.

d **Throws:**: Nothing.

e **Remarks**: For every union object or subobject in the relocated range [`new_location`, `new_location + begin - end`), the active member is the active member of the corresponding union object or subobject from the original range [`begin, end`).

[*Note:* A likely implementation will use compiler-specific functionality that simply calls `memmove` and updates its notion of the object lifetime. —*end note*]

15 Acknowledgements

This document is written in markdown and depends on the extensions in `Pandoc` and `mpark/wg21`, for which we would like to thank the authors of those extensions and associated libraries.

The authors would also like to thank Brian Bi and Joshua Berne for their assistance in proofreading this paper, especially the proposed Core wording.

Thanks to Lori Hughes for reviewing this paper and providing valuable editorial feedback.

The authors are also grateful to Corentin Jabot for both reviewing this paper and providing an example implementation of this proposal.

Also, this paper has been 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.

16 References

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

[N4971] Thomas Köppe. 2023-12-18. Working Draft, Programming Languages — C++.
https://wg21.link/n4971

[P0843R5] Gonzalo Brito Gadeschi. 2022-08-14. static_vector.
https://wg21.link/p0843r5

[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

[P1144R7] Arthur O'Dwyer. 2023-03-10. std::is_trivially_relocatable.
https://wg21.link/p1144r7

[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

[P2959R0] Alisdair Meredith. 2023-09-15. Relocation Within Containers.
https://wg21.link/d2959r0

[P2967R0] Alisdair Meredith. 2023-10-15. Relocation Has A Library Interface.
https://wg21.link/d2967r0