

Trivial Relocatability For C++26

Proposal to safely relocate objects in memory

Document #: P2786R10
Date: 2024-11-20
Project: Programming Language C++
Audience: EWG, LEWG
Reply-to: Mungo Gill
<mgill83@bloomberg.net>
Alisdair Meredith
<ameredith1@bloomberg.net>
Joshua Berne
<jberne4@bloomberg.net>
Corentin Jabot
<corentinjabot@gmail.com>
Pablo Halpern
<phalpern@halpernwrightsoftware.com>
Lori Hughes
<lori@lorihughes.com>

Contents

1 Abstract	4
2 Revision History	5
R10: November 2024 (Wrocław LEWG)	5
R9: November 2024 (Wrocław meeting)	5
R8: October 2024 (pre-Wrocław mailing)	5
R7: September 2024 (midterm mailing)	5
R1-6: January 2023 – April 2024	6
3 Document Conventions	7
3.1 Typography	7
3.2 Definitions	7
3.3 Core-language additions	7
3.4 Library additions	7
4 Introduction	9
5 Basic Ideas	10
5.1 Trivial relocatability	10
5.2 Replaceability	10
5.3 Examples of trivially relocatable and replaceable types	10
5.4 Examples of trivially relocatable types that are <i>not</i> replaceable	11
5.5 Examples of replaceable types that are not <i>trivially</i> relocatable	11
5.6 Examples of Standard Library types that must defer to the implementation	11
5.7 Independent features	11
6 Technical Background	12

7	Basic Design Principles	13
7.1	Create a feature for users, not just the Standard Library	13
7.2	A formal specification is built around object lifetimes	13
7.3	Behavior must be reliable	13
7.4	Guard against accidental undefined behavior	13
8	Core Proposal	14
8.1	Trivial relocatability	14
8.2	Replaceability	17
8.3	(Not) Optimizing <code>std::swap</code>	18
9	Simple Worked Examples	19
9.1	Exposition-only classes	19
9.2	Rule of zero	19
9.3	Impact of destructors	20
9.4	Impact of move constructors	21
10	Plans for Library Update	22
10.1	Immediate updates	22
10.2	ABI concerns	22
10.3	Specific follow-ups	23
10.4	Consumer APIs	23
11	Use Cases	25
11.1	Optimizing <code>std::vector</code> at run time	25
11.2	Optimizing <code>std::optional</code> to be trivially relocatable and replaceable	25
12	Implementation Experience	27
13	FAQ	28
13.1	Is <code>void</code> trivially relocatable?	28
13.2	Are reference types trivially relocatable?	28
13.3	Why can a class with a reference member be trivially relocatable?	28
13.4	Are <i>cv</i> -qualified types, notably <code>const</code> types, trivially relocatable?	28
13.5	Can <code>const</code> -qualified types be passed to <code>trivially_relocate</code> ?	28
13.6	Can types that are not implicit-lifetime types be trivially relocatable?	28
13.7	Why are virtual base classes not trivially relocatable?	28
13.8	What happens if a <code>relocate</code> operation throws?	29
13.9	Why do deleted special members inhibit implicit trivial relocatability?	29
13.10	Can the compiler transform argument passing with trivial relocation?	29
13.11	Can the Standard Library containers use this new feature internally?	29
13.12	Do implementations need to mark classes <code>trivially_relocatable</code> to benefit?	31
13.13	Can I mark as trivially relocatable a type that is not replaceable?	31
13.14	Can I mark a type as replaceable but not trivially relocatable?	31
13.15	What happened to the predicates for the contextual keywords?	31
13.16	Why is copy-replacement unsupported?	31
13.17	Why is marking a class that can <i>never</i> be trivially relocatable not ill-formed?	31
13.18	Why is there no <code>is_trivially_replaceable</code> trait?	32
13.19	Is it UB to mark a nonconforming type as trivially relocatable?	32
13.20	Is it UB to mark a nonconforming type as replaceable?	32
13.21	Why does replaceability not require trivial relocatability?	32
13.22	Why does trivial relocation support <code>const</code> data members, but replacement does not?	32
13.23	Why does <code>[library.class.props]</code> explicitly call out permission to use the contextual keywords?	33
13.24	Why are classes with virtual base classes “replaceable”? What does that even mean?	33

14 Illustrative Examples	34
14.1 Standard vector	34
14.2 Conforming implementation of a trivially relocatable <code>std::optional</code>	34
14.3 C++26 implementation using internal array	37
15 Proposed Wording	42
15.1 Add new identifiers with a special meaning	42
15.2 Specify trivially relocatable types	42
15.3 Address trivial relocation of lambdas	42
15.4 Update grammar to support <code>memberwise</code> contextual keywords	43
15.5 Specification for trivially relocatable classes	44
15.6 Add feature macros	45
15.7 Library wording	45
15.8 Add new type traits	46
15.9 Specify the compiler-magic functions	46
16 Acknowledgements	48
17 References	48

1 Abstract

Many types in C++ cannot be trivially moved or destroyed but do support trivially moving an object from one location to another by copying its bits — an operation known as *trivial relocation*. Some types even support bitwise swapping, which requires replacing the objects passed to the `swap` function, without violating any object invariants. Optimizing containers to take advantage of this property of a type is already in widespread use throughout the industry but is undefined behavior as far as the language is concerned. This paper provides a mechanism to annotate types as having the appropriate properties to be eligible for these optimizations, along with library interfaces to make use of them in a well-defined manner.

2 Revision History

R10: November 2024 (Wrocław LEWG)

- To increase consensus, reduced scope back to minimal design
 - Dropped the plain `relocate` function
 - Dropped `swap_value_representations` function
 - Defer to QoI for implementations to explore optimizing `swap` as they see fit
 - Renamed keywords to drop `memberwise_`
- Editorial changes to sync with the technical changes above
 - Removed all content referring to `swap_value_representations` and `relocate`
 - Removed most remaining content talking about `swap`
 - Removed library support for nontrivial relocation
 - Added rationale for how `swap` can be optimised with just this simplified paper
 - Removed FAQ entries that were no longer relevant
 - Add a section on future Consumer APIs

R9: November 2024 (Wrocław meeting)

- Added missing precondition to `trivially_relocate` and `relocate`
- Added an extensive set of worked examples to demonstrate primitive functionality
- Added more FAQ entries prompted by reflector discussion
- Fixed core wording
 - Types with deleted destructors can no longer be trivially relocatable or replaceable.
 - To be *eligible for replacement*, a class must have an *eligible* constructor and assignment operator.
- New wording for `swap_value_representations`
- New discussion of contextual keyword renaming

R8: October 2024 (pre-Wrocław mailing)

- Extracted [Document Conventions](#) to above the Introduction
- Added [Basic Ideas](#), a higher-level introduction to the new features
- Listed example types for the different type categories defined by this paper
- Corrected implementation of several example functions
- Extended the FAQ
 - What happens if a `relocate` operation throws?
 - Why is there no `is_trivially_replaceable` trait?
 - Is it UB to mark a nonconforming type as trivially relocatable?
 - Is it UB to mark a nonconforming type as replaceable?
- Explained how to apply the new features to optimize vectors

R7: September 2024 (midterm mailing)

- Significant redrafting since [\[P2786R6\]](#) to better address EWG and LEWG concerns
- Simplified presentation and discussion of trivial relocatability (for detailed history, consult [\[P2786R6\]](#))
- Integrated discussion of `swap`; the present paper supersedes [\[P3239R0\]](#)
- Behavior changes since [\[P2786R6\]](#)
 - User-provided move assignment now prevents a type from being implicitly trivially relocatable
 - The contextual keyword
 - gets a new name, `memberwise_trivially_relocatable`, to better reflect revised semantics
 - is opt-in only
 - deduces relocatability on bases and members
 - No predicate follows the contextual keyword, so no mechanism for opting out is present
 - A new `relocate` function additionally supports nontrivial types and constant evaluation
- Behavior changes since [\[P3239R0\]](#)

- Clarifies that trivial swappability is based on being replaceable and trivially relocatable
- Proposes a new contextual keyword, `memberwise_replaceable`
- Proposes optimizing `std::swap`, using the new properties, and the `swap_value_representations` function

R1–6: January 2023 – April 2024

Early versions of this paper were careful to include comparison and contrast with other papers in this space. That progress is archived by [\[P2786R6\]](#).

The evolution groups requested a clean draft that presents just our proposal and integrates our follow-up papers such that a single coherent design is presented, and revision R7 is the original response to that request.

Discussions following the EWG review of R9 of this paper on November 18, 2024, indicated that consensus would be improved by reducing this paper back to a minimal design focused entirely on language support for trivial relocation, so some of the R7 library extensions, such as optimizing `std::swap`, were removed to separate follow-up papers again.

3 Document Conventions

3.1 Typography

Throughout this paper, a **bold typeface** will be used for terms defined herein and *bold italicized typeface* for terms of art defined herein; the proposed wording, however, will use the conventions of the Standard.

3.2 Definitions

We define a **relocation operation** of a source object as one that ends the lifetime of that object and starts the lifetime of a new object at a new location. Importantly, the destructor is not necessarily run by a **relocation operation**. For types in which move construction and destruction are supported, a relocation can be accomplished by constructing an object in the new location from an xvalue referring to the source object, followed by invoking the destructor of the source object.

We define a **trivial relocation operation** as a **relocation operation** accomplished by performing a bitwise copy of its *object representation* to a new memory location that ends the lifetime of that source object — just as if that (source) object’s storage were used by another object (6.7.3 [basic.life]p5) — and starts the life of a new object at the new location. 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 nontrivial **relocation operation** performed via move construction and destruction (though exceptions, while not encouraged, are not expressly forbidden).

We define **replacement** of a target object by a source object as destroying the target object immediately followed by move construction into the location of the target object from the source object. For many types, this operation is semantically equivalent to a move-assignment operation from the source object to the target object.

3.3 Core-language additions

We propose a new Core-language definition for a *trivially relocatable type*. This new definition is inspired by the recursive nature and handling of special member functions used in the definition of a *trivially copyable* type. A *trivially relocatable type* is a scalar type, a *trivially relocatable class*, an array of such types, or a cv-qualified version of such a type. A class will be implicitly *trivially relocatable* if all its bases and members are *trivially relocatable* and none of its eligible special member functions are user provided; a contextual keyword will signify that a class may still be *trivially relocatable* even if it has user-provided special member functions.

We similarly propose a new Core-language definition for a *replaceable type* in which a class will be a *replaceable class* if all its bases and members are *replaceable* and none of its relevant special member functions are user provided; a contextual keyword will signify that a class may still be a *replaceable class*, even if it has those user-provided special member functions.

Because **replaceability** is explicitly about the equivalence of assignment with destruction followed by construction, we do not decide that a type is *implicitly replaceable* when the special member functions selected for those operations are user provided.

3.4 Library additions

The Standard Library APIs to support **trivial relocation** comprise

- a type trait to detect **trivial relocatability**
- a function, `trivially_relocate`, that performs relocation on a range of objects by moving their bytes (similarly to `memmove`) while starting the lifetime of the destination objects and ending the lifetime of the source objects

To support common use cases, `trivially_relocate` is specified to support overlapping ranges.

Further, to take advantage of the ability to specify that a type is *replaceable*, we propose the following additional changes to Standard Library API:

— A type trait to detect **replaceability**

Finally, we propose modifications to Standard Library wording to describe when Standard Library types are allowed and expected to have various properties, including **trivial relocatability** and **replaceability**.

4 Introduction

Containers in C++, in particular those like `std::vector` and `std::deque` that manage objects within a range of continuous storage, live and die by the efficiency with which they can move objects around. One of the most common fundamental steps in many of the operations these types perform is that of relocation — taking an element at one location in memory and creating a new element at a different location in memory with the same value and then destroying that original value.

Many frequently used libraries have long recognized that, for many types, the two nontrivial steps of move construction and destruction often combine into a single operation that can be accomplished by a simple bitwise copy followed by discarding the source object instead of evaluating its destructor. Much of the work a move constructor might do to the source object, such as setting pointers to owned data to `nullptr`, is done only to make sure the destructor that will eventually run knows that no data is present that it is still responsible for freeing. By taking advantage of the knowledge that certain types can be relocated by simply copying bits, complex operations that can involve the invocation of many user-provided special members functions can be replaced by single calls to `memcpy`, realizing huge benefits in performance.

The problem, of course, is that moving objects in this fashion that are not trivially copyable violates the C++ object model and is undefined behavior. In this paper, we propose a mechanism to fix that problem.

- Types with no user-provided special member functions are identified as *trivially relocatable* based on whether their bases and nonstatic data members are *trivially relocatable*.
- Users are able to further identify those types whose user-provided special member functions compound into a trivial operation by marking those types with a new class specifier, `trivially_relocatable`.
- The Standard Library then provides a tool to safely perform **trivial relocation operations**.

A more subtle problem occurs where the developer wants to apply optimizations based on *trivial relocation*, but their code was previously taking advantage of library APIs to assume that assignment and destroy-then-construct were interchangeable operations. For types that do not support that property, switching to `trivially_relocate` which emulates move construction will behave differently to code optimized to use assignment instead. This property of a type, that we name **replaceability**, is not programmatically detectable today. For a frequently mentioned example, the C++ Standard Library specification for `std::vector` insert and erase operations allows implementers to relocate elements using assignment, assuming but not requiring that all stored types are *replaceable*; it would be an observable change of behavior for an implementation using assignment to relocate elements in these operations to start using trivial relocation instead — an otherwise valid transformation — unless it could make a compile-time test to verify that the element type is not only *trivially relocatable*, but also *replaceable*.

To incorporate this concept of **replaceability** into the language, we propose further additional changes.

- Types are implicitly recursively *replaceable* if all their members and bases are *replaceable*.
- Through the use of the `replaceable` specifier on a class, users can specify that their user-provided constructors, destructors, and assignment operators still satisfy the appropriate equivalence specified by being *replaceable* as long as all members and bases also have this property.

Put together, we hope this proposal provides a complete picture of how to incorporate into the C++ Standard, in a comprehensible and effective manner, bitwise operations that are already performed by many libraries in the industry.

5 Basic Ideas

This paper introduces two new complementary but independent notions into C++.

5.1 Trivial relocatability

Relocation is the act of moving an object from one memory location to another and is typically achieved by calling the move constructor to make a new object at the new location followed by the calling the destructor of the original object to end its lifetime.

A type is *trivially relocatable* if it can be relocated by copying the bytes of its *object representation* from the old location to the new and the lifetime of the original object can be ended *without* running its destructor. However, C++ object lifetimes do not currently permit types (with the exception of those few types that meet the strict requirements of trivial copyability) to be relocated by means of byte copying (**trivial relocation**).

If the object model were to allow it, most C++ types could safely be trivially relocated. The two known exceptions are types that maintain an internal pointer to a data member and types that register their presence in an external registry that must point back to the object.

This paper proposes adding

- a Core-language definition for *trivially relocatable types*
- a way to explicitly mark types *trivially relocatable* when that cannot be deduced by a compiler
- a type trait to report whether a type is *trivially relocatable*
- a “compiler-magic” function to perform **trivial relocation**, respecting object lifetimes

All the features of **trivial relocation** have decades of experience in which code has relied on compilers not reacting to the use of undefined behavior when copying nontrivial types breaks the C++ object lifetime rules.

5.2 Replaceability

Replaceability is a semantic property of a type, where move assignment is isomorphic to destroy then move-construct. Just like in **trivial relocatability**, a compiler cannot deduce whether a type is *replaceable* if the user provides a move-assignment operator, move constructor, or destructor without extra guidance from the user.

In many cases, a library would like to require or assume **replaceability**, such as for moving elements around a `std::vector` when inserting or erasing elements.

This paper proposes adding

- a Core-language definition for *replaceable* types
- a way to mark types as *replaceable* if the compiler cannot deduce that property
- a type trait to report whether a type is *replaceable*

Note that no part of the language requires types to be *replaceable*; this feature is purely to allow users to mark their types with a property that many libraries seek to exploit.

5.3 Examples of trivially relocatable and replaceable types

If we assume the default template arguments, we would expect the following Standard Library types to be both *trivially relocatable* and *replaceable*:

- `std::shared_ptr`
- `std::future`
- `std::vector`

We would expect the following types to be both *trivially relocatable* and *replaceable* if all their template arguments are both *trivially relocatable* and *replaceable*:

- `std::pair`

- `std::tuple`

5.4 Examples of trivially relocatable types that are *not* replaceable

A variety of types, while *trivially relocatable*, do not maintain the invariants of **replaceability**:

- `std::tuple<T &>`
- `std::pmr` containers
- Any class with a `const` data member

In many contexts, relocation of such types is desirable, especially in user-defined data structures beyond the reach of the Standard Library.

5.5 Examples of replaceable types that are not *trivially* relocatable

The main example in this category is Standard Library containers with debug iterators that track their container with a back-pointer or some other registry, although we can easily imagine user-supplied types with similar constraints. Note that some implementations of `std::basic_string` fall into this category, where the short string optimization maintains a pointer to its internal short buffer.

This category of types would meet preconditions for algorithms in which the semantics of **replaceability** are important, and they might be enforced by the equivalent of Mandates, Constraints, or Preconditions in users' libraries.

5.6 Examples of Standard Library types that must defer to the implementation

We would expect the quality of implementation would decide whether the following types are *trivially relocatable and replaceable* or are just *replaceable*:

- `std::basic_string`, depending on whether the short string optimization maintains an internal pointer
- `std::list`, depending on whether the sentinel node is a nonstatic data member

5.7 Independent features

From the variety of types and usage examples above, we see that while **trivial relocation** and **replacement** are often used together, each has important use cases and neither can be built on top of the other.

6 Technical Background

Some very specific uses of terminology from the C++ Standard are important to understand when reading this proposal and are quickly summarized here.

For decades, C++ developers have been optimizing low-level data structures, such as their own `vector`-like types, by byte-wise copying objects from one location to another, even though doing so is often UB; see earlier papers¹ for rationale.

Earlier revisions of this paper initially proposed language and library extensions, termed **trivial relocation**, to make writing such code well defined and was forwarded to Core where it received a strong review that challenged our assumptions about copying bytes. From the perspective of the C++ abstract machine, we should not be making assumptions about in-memory representations — that is the compiler’s job — and should limit ourselves to copying the *object representation*, leaving the compiler itself to optimize copying and moving the object representations using efficient memory-copying operations.

The Core review in Tokyo 2024 proceeded in parallel to the LEWG review at the same meeting, which subsequently sent the proposal back to EWG, asking for a more complete handling of bitwise operations, notably optimizations for byte-wise swaps. Subsequent feedback, given that `swap` cannot rely on trivial relocation lest it corrupt a potentially overlapped member subobject, was that the full details of `swap` are best left to the library and compiler to work out for themselves and that supplying the two traits for trivial relocatability and for replaceability is sufficient for them to make progress.

¹Much rationale related to **trivial relocation** can be found in [P2786R6], and early discussion of handling `swap` is covered in [P3239R0].

7 Basic Design Principles

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

Efficient implementation of many data structures often needs a means to efficiently move and exchange objects that those data structures are managing, especially for data structures that manage their elements in contiguous storage or in some other location that is not a node at the end of a pointer.

We note that such object manipulation is also a sharp tool that is not expected to see much use other than optimizing the internal management of data structures. While this paper focuses on supplying the essential building blocks of **trivial relocation**, we defer the design of a more usable consumer API to follow-up papers that can debate their varied merits and trade-offs.

7.2 A formal specification is built around object lifetimes

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

7.3 Behavior must be reliable

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

7.4 Guard against accidental undefined behavior

The new Library APIs support only types that would produce well-defined behavior. The specification prefers *Mandates* clauses to *Constraints*: clauses since SFINAE behavior carries no expected benefit and is likely to produce error messages with less useful information.

8 Core Proposal

Our core proposal comprises two parts: **trivial relocation** and **replaceability**, each including the library primitives that are necessary for well-defined use. **Trivial relocation** is a technique already widely used in the industry, and **replaceability** is a more novel property that is exploited directly by optimizing library code based on its availability.

8.1 Trivial relocatability

To ensure that libraries taking advantage of the *trivially relocatable* semantic do not introduce undefined behavior, the model of lifetimes for objects must be extended to allow for relocation of *trivially relocatable types*. Since 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, `std::trivially_relocate`, that is restricted to *trivially relocatable types*. The purpose of the new function template is to efficiently move the *object representation*, typically with a call to `memmove` that also signifies to the compiler (and other analysis tools) that the lifetime of the new object(s) has begun — similar to calling `start_lifetime_as` on the destination location(s) — and that the lifetime of the original object(s) has ended (without running destructors).

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.

8.1.1 New type category: Trivially relocatable

To better integrate language support, we further propose 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, the same applies for the assignment operator for rvalues, and their destructor is neither user provided nor deleted. Conceptually, this definition combines the rules we would follow if there was a new user-definable special member function for relocation *and* when that operation would be trivial.

Note that our notion of relocation relies on being semantically equivalent to move construction of the target followed by destruction of the source. Even though it is not involved in this definition, we still consider assignment operations when deciding if a type is *implicitly trivially relocatable* for the same reasons that we consider assignment when deciding if a type should have an implicitly declared move constructor; any existing type with a particular set of user-provided special member functions should not begin to have new operations considered valid for it if those operations might subvert expectations due to compiling with a new language Standard.

8.1.2 New keyword and explicit rule

Without an opt-in mechanism, the only types that would be implicitly *trivially relocatable* would be those that are already trivially copyable, an important yet relatively small subset of the full universe of types in C++. To enable **trivial relocatability** on the many more interesting types that have nontrivial special member functions, explicitly marking such types must be possible. This marking is needed for only user-defined class types (including unions); hence, we propose 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 is placed like `final`.
```

We propose one new contextual keyword, `trivially_relocatable`, that can be placed in a class-head (on a class definition) to indicate that a type's special operations do nothing that would violate the implicit rule that would make a type *trivially relocatable*.

By means of the `trivially_relocatable` specification, a class will be determined to be *trivially relocatable* if, according to the implicit rules for a *trivially relocatable class*, the class would be *trivially relocatable* if the presence of user-declared special member functions were ignored.

Users considering whether to apply this keyword to a given type that has user-provided special member functions must simply inspect their move constructor and destructor and decide if, when applied together as part of a relocate operation, they have no net effect. Common examples include many types.

- For a resource-owning type, such as `std::unique_ptr`, the newly constructed object will have the same bits as the source object, the source object will have its pointer member set to `nullptr`, and the source object destructor will do nothing because, by the time it runs, that member will be `nullptr`. Simply copying the bytes and discarding the source object achieves the same semantic effect.
- A reference-counting type, however, might increment a count by one when constructing the target object and then decrement that same count by one when destroying the source object. Combining these operations clarifies that the constructor and destructor negate one another.

8.1.3 New type trait: `is_trivially_relocatable`

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 has `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.

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 since users do *not* have permission to specialize standard type traits (unless explicitly granted permission for a specific trait).

We see no particular need to separately detect whether a type has attempted to make itself *trivially relocatable* with the `trivially_relocatable` token or by leaning on the implicit definition.

8.1.4 New relocation function: `trivially_relocate`

As stated in “[Library additions](#),” 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:

```
template <class T>
T* trivially_relocate(T* begin, T* end, T* new_location);
{
    static_assert( is_trivially_relocatable_v<T> && !is_const_v<T> );
    // ... (platform-provided implementation)
}
```

This function template *mandates* that `is_trivially_relocatable_v<T> && !is_const_v<T>` is `true` and has *preconditions* that `end` is reachable from `begin`. Its *postcondition* is that new objects in the range `[new_location,`

`new_location + sizeof(T) * (end - begin)`) have the same object representation as the objects originally in the range `[begin, end)` and that the objects originally in the range `[begin, end)` have ended their lifetime, all accomplished without running any destructors or other clean-up code. Overlapping ranges shall be supported.

On most platforms, this template is functionally equivalent to

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

However, unlike `memmove` on its own, this function template is restricted to *trivially relocatable types* rather than to implicit lifetime types.

Note that this function has the nofail guarantee and can never throw an exception, yet it is not marked as `noexcept` by following the principles of the Lakos Rule. If a function has a narrow contract, then unless that function is likely to be used in conjunction with the `noexcept` operator, the exception specification should be left to the library as QoI.

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`.

1. The `trivially_relocate` function ends the lifetime of the objects `*begin`, `*(begin+1)`, ..., through to `*(end-1)`. This ending of the objects' lifetimes means accessing these objects or attempting to run their destructors will be *undefined behavior*.
2. 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)`.
3. These operations are a single action, and for any locations where an overlap occurs between the source and target ranges, an existing object will be destroyed and a new object will be created in its place.

The current library-level mechanism to start the lifetime of an object without invoking a constructor is `std::start_lifetime_as`, a function that works for only implicit lifetime types that must have trivial default constructors. *Trivially relocatable types*, however, include a much wider range of types, including many that establish and maintain invariants in their special member functions and thus cannot be implicit lifetime types.

A tool for ending lifetimes is similarly unavailable in the Standard Library today. This task can be accomplished by reusing the storage of an object, but that requires modifications of some sort.

The `trivially_relocate` function, therefore, is interacting with the abstract machine in ways that are not currently available. Importantly, for many of the types we are concerned with (e.g., `std::vector`, `std::unique_ptr`, and so on), the component steps of the **relocation operation** are decidedly not trivial, so we are compelled to make this single function responsible for the needed compiler magic.

To remove the need for a larger family of functions and avoid overly limiting cases in which **trivial relocation** might be applied, the `trivially_relocate` function is intended to support overlapping source and destination ranges, just like `memmove`. If the ranges are overlapping, the implementation must take care around the management of the lifetime of objects relocated out of or into the overlap. Support for overlapping ranges also mitigates the risk of introducing undefined behavior on memory operations that is the cause of so much reputational harm to C++ at the moment; if we leave detecting and responding to the case of overlapping ranges to our consumers, something they do not have to do in the case of `memmove` today, then we invite a different, more pernicious UB to replace the code widely tested and relied upon in production today.

Note that this initial proposal *deliberately* does not provide a single-object relocation function. Our primary motivation based on current experience is to optimize relocating objects in bulk, which is expected to be the initial common use case. While adding single-object `trivially_relocate` functions would be simple, the effect can be achieved by calling the pointer-range function with a range of a single object. We prefer for users to find relocating from local variables and data members more cumbersome since that will give them pause to think about what they are doing and will call attention during code review. Relocating a single object is like running its destructor and leaving the problem in a fragile state that needs to be recovered before leaving the scope of

the relocated object. For such cases, we plan future language support that can better handle the relocated-away (or destroyed) state, and we believe it foolish to deliver a feature that provides a simple way to *enable* more user UB when the feature's stated design goal is to *reduce* the occurrence of UB in practice.

8.2 Replaceability

In addition to **trivial relocation**, we introduce the orthogonal notion of **replaceability**. An object of type T is *replaceable* by an object of type U if destroying the object of type T and reconstructing an object of type T in its place from an xvalue of type U is equivalent to assigning to the original object of type T with an xvalue of type U. Note that **replacement** updates an object's value, so **const**-qualified objects are never *replaceable*.

Replaceability is an important property when we want to transform **relocation** into assignment or vice versa. Containers such as `std::vector` already make a general assumption that all types are *replaceable*, but other functions, such as `std::swap`, do not make such an assumption, so we provide a mechanism to identify those types providing guarantees using this new property.

8.2.1 New type category: Replaceable type

A type T is a *replaceable type* if every object of type T is *replaceable* by every other object of type T. Note that *replaceable types* must be object types; function types, reference types, and `void` are never *replaceable*.

A cv-unqualified type T will implicitly be a *replaceable type* if all its bases and nonstatic members are *replaceable types* and if it has no user-provided move constructor, move-assignment operator, nor destructor.

8.2.2 New keyword and explicit rule

To enable **replaceability** to be useful for classes with user-provided special member functions, explicitly marking class (including union) types as potentially *replaceable* must be possible (just like for *trivially relocatable types*). To that end, we propose adding a new contextual keyword, `replaceable`, as part of the class definition (mirroring the design of `trivially_relocatable`).

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

A class can be marked with both `trivially_relocatable` and `replaceable`; in fact, we expect many uses of `replaceable` to also require `trivially_relocatable`.

8.2.3 New type trait: `is_replaceable`

To expose the **replaceability** property of a type to library functions seeking to provide appropriate optimizations, we propose a new trait, `std::is_replaceable<T>`, that enables the detection of *replaceable types*:

```
template< class T >
struct is_replaceable;

template< class T >
constexpr bool is_replaceable_v = is_replaceable<T>::value;
```

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

Note that the `std::is_replaceable` 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.

8.3 (Not) Optimizing `std::swap`

`std::swap` usage differs significantly from **trivial relocation** in several ways. `std::swap` is an existing well-specified function with a wide contract, and it starts and ends with two valid objects and cannot end the lifetimes of either without vastly changing its current expected behavior; users will have no well-defined way to implement a safe, general purpose, byte-wise **swap**. However, Standard Library implementations are not constrained by simple things like undefined behavior, so vendors will remain free to provide such optimizations as a QoI feature that relies on `is_trivially_relocatable` and `is_replaceable` to spot candidate types; the implementation would still need to rely on compiler intrinsics to avoid the dangers inherent in nontransparent **replacement**, but techniques to evade this problem are known.

Note that the revisions R7–9 of this paper did extensive work to guarantee a byte-wise **swap**, but ultimately those extensions were deemed complex, distracting, and nonessential. They may return in a follow-up paper if this paper (P2786) is adopted.

8.3.1 Untyped nested subobjects

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

When an object is relocated, all its nested subobjects, including those of dynamic storage duration stashed in member arrays, must be relocated too.

9 Simple Worked Examples

To help dispel confusion and misunderstanding, we present a variety of simple classes that illustrate most of the concerns regarding whether a type will be *trivially relocatable*, *replaceable*, neither, or both. For reference, we will also note whether such types are trivially copyable as well.

9.1 Exposition-only classes

The following exposition-only classes have their semantics defined by their documentation comments. They are used throughout the rest of this section to illustrate the interaction of the proposed new facilities with both implicit and explicit deduction of the new properties with a relevant variety of data members.

```
struct Empty {};

static_assert(    is_trivially_copyable_v<X>);
static_assert(    is_trivially_relocatable_v<X>);
static_assert(    is_replaceable_v<X>);

struct Non-Trivial {
    // Implementation details are elided.
    // Non-Trivial is neither trivially copyable, trivially relocatable, nor replaceable.
};

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);

struct Immobile {
    Immobile(Immobile&&) = delete;
    Immobile& operator=(Immobile&&) = delete;

    Immobile() = default;
};

static_assert(not is_trivially_copyable_v<X>);
static_assert(    is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);
```

9.2 Rule of zero

```
struct X{};

static_assert(    is_trivially_copyable_v<X>);
static_assert(    is_trivially_relocatable_v<X>);
static_assert(    is_replaceable_v<X>);

struct X : Empty {};

static_assert(    is_trivially_copyable_v<X>);
static_assert(    is_trivially_relocatable_v<X>);
static_assert(    is_replaceable_v<X>);
```

```

struct X : virtual Empty {};

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(    is_replaceable_v<X>);           // Replaceable types can have virtual bases.

```

```

struct X trivially_relocatable : virtual Empty {};

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>); // Trivially relocatable types never have
// virtual bases.
static_assert(    is_replaceable_v<X>);

```

```

struct X {
    Non-Trivial data;
};

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);

```

9.3 Impact of destructors

```

struct X {
    ~X() = default;
};

static_assert(    is_trivially_copyable_v<X>);
static_assert(    is_trivially_relocatable_v<X>);
static_assert(    is_replaceable_v<X>);

```

```

struct X {
    ~X();
};

X::~X() = default;

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);

```

```

struct X {
    virtual ~X() = default;
};

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);

```

```

struct X {
    ~X() = delete;
};

static_assert(not is_trivially_copyable_v<X>);

```

```
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);
```

9.4 Impact of move constructors

```
struct X {
    X(X&&) = default;
};

static_assert(    is_trivially_copyable_v<X>);
static_assert(    is_trivially_relocatable_v<X>);
static_assert(    is_replaceable_v<X>);
```

```
struct X {
    X(X&&);
};

X::X(X&&) = default;

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);
```

Note: This class has an implicitly deleted copy constructor and an implicitly deleted copy-assignment operator.

```
struct X {
    X(X&&) = delete;
};

static_assert(not is_trivially_copyable_v<X>);
static_assert(not is_trivially_relocatable_v<X>);
static_assert(not is_replaceable_v<X>);
```

10 Plans for Library Update

We plan to enable the adoption of these new features in follow-up papers targeting LEWG.

10.1 Immediate updates

In addition to specifying the type traits and library functions that enable the facilities, we should update the library frontmatter to indicate whether and how the Library is allowed to use these features to enhance their QoI.

Clearly, under the as-if rule, the Library immediately gets permission to optimize algorithms and functions for *trivially relocatable* and *replaceable* types where such optimizations are not observable. For example, `std::vector` could optimize many of its operations for such types, given a suitable allocator, such as the default `std::allocator`. No updates to the Library specification are needed for these optimizations, and follow-up papers that suggest changing specifications to allow such optimizations that *would* be observable should be properly directed to LEWG.

The other category of interest is whether Library types themselves can — or should — be *trivially relocatable* or *replaceable*. For example, any implementation of `std::vector` should be able to satisfy the requirements to be both *trivially relocatable* and *replaceable* for any element type as long as its allocator has those properties; we might want to mandate that `std::vector` is relocatable and/or *replaceable* in such cases. Conversely, in the two common implementation strategies for `std::list`, the *sentinel node* is either dynamically allocated or stored directly in the footprint of the `list`. The dynamic node case is always *trivially relocatable* and *replaceable*, but the in-place representation is neither; however, the in-place representation is *nothrow-movable*, whereas the dynamic case must allocate a new node, which can potentially throw. In both cases, **relocation** will never throw, but different trade-offs must be considered when choosing an implementation strategy, and such cases are almost always better left for implementation QoI (especially since ABI concerns might require consideration).

When granting permission for implementations to use keywords that are in addition to those specified by the C++ Standard, we have taken two approaches that we will term the **noexcept** approach and the **constexpr** approach. In the **noexcept** approach, an implementation is granted permission to add **noexcept** specifications to functions as long as those specifications do not invalidate other aspects of the function contract; i.e., an exception specification cannot be added to a virtual function or to a function that is specified to throw exceptions. Conversely, the **constexpr** approach disallows adding **constexpr** to a function that is not declared as **constexpr** in the C++ Standard.

For the purposes of this paper, we believe the minimal necessary specification should use the **noexcept** approach, and we propose the appropriate wording to say so. That choice will allow implementations to experiment with the feature and then provide clear recommendations for specific cases as follow-up LEWG papers.

10.2 ABI concerns

We believe no ABI concerns exist for libraries applying these new features throughout the Standard Library, even as unspecified QoI improvements.

The name-mangling of a type should not depend on whether it is either *trivially relocatable* or *replaceable*. While these properties can be determined through type traits, by definition of being a new feature, no existing code will be SFINAE-enabled on these traits. Updating the internal layout of any Standard Library type to accommodate optimizations using these traits should be unnecessary.

The main concern might be adding constraints to implementation-specific functions used to dispatch to optimized algorithms, such as when growing a vector. In these cases, to avoid introducing new mangled names that would affect link compatibility, **if constexpr** within the dispatching function could be used to enable a fully link-compatible library.

One situation that should be called out is when a library wants to adopt an optimization with an observable behavior change, such as relocating a nonreplaceable type where previously assignment was used. The same

concerns would arise as with any other change of unspecified behavior or even a typical bug fix, and library vendors may choose to be conservative and postpone making those QoI changes.

10.3 Specific follow-ups

In a follow-up paper, we intended to propose adding a new specification element, *Class properties*, for any specification related to class properties 11.2 [class.prop]. The Standard Library already makes some effort to specify whether a class must be trivially copyable, standard layout, and so on, and we believe tracking such specification would be more maintainable with a consistent presentation and using a consistent form.

Once we have a *Class properties* element, we can then review all library classes and decide whether to specify the **trivial-relocatability** behavior for that class, which might be conditional on its template arguments if it is a class template. We might also deliberately defer specifying behavior to allow for implementations making different choices, such as node-based containers allocating their end node vs. storing the pointers in the container's object representation.

Finally, once we have an easy way to document class properties, we might consider making stronger guarantees on existing library components where such specification would be useful, e.g., clarifying which types are implicit lifetime.

We would propose moving the specification for the following properties in this new element

- *Trivially Copyable*
- *Standard Layout*
- *Implicit Lifetime*
- *Structural*
- *Aggregate*
- *Empty*
- *Bitmask*

along with the two new properties specified in this paper

- ***Trivially Relocatable***
- ***Replaceable***

The following clauses in the Standard Library specification would then include additional notes regarding this new element and updated specification:

- 16.3.2.4 [structure.specifications] — class properties as well as invariants
- 16.3.3.3.5 [customization.point.object] — may be mildly reformulated with the new specification element
- 16.3.3.5 [objects.within.classes] — may be constraining which members may be added

10.4 Consumer APIs

In this revision of P2786 (i.e., R10), we have deliberately removed the consumer facing API in order to bring multiple proposals by different authors to the next meeting, without prejudice of seeing just one of those proposals before the rest were ready. The proposals we expect are shown below, and for brevity, we list just the expected benefits and concerns when compared to directly calling `trivially_relocate`.

10.4.1 Extend the uninitialized memory algorithms

- `constexpr`
- Support for nontrivially-relocatable types
- Consistent forward-looking library interface
- Specific algorithm for each direction
 - User is responsible for avoiding UB in overlaps
- Supports a wide array of primitive iterator categories
 - May need contiguous iterators for the special cases using trivial relocation
- Will modernize and address API concerns with all the other uninitialized APIs

- Will provide complete support for `ranges` algorithms too

10.4.2 Provide a simple `relocate` function for non-trivial types passed as pointers

- `constexpr`
- Support for nontrivially-relocatable types
- Automatically dispatches to trivial relocation where it can
- Nofail
- `nothrow_relocatable` mandate
- Concern that it is a duplicative distraction to the uninitialized APIs

10.4.3 Provide an API to easily relocate a single object

- `constexpr`
- `relocate_at` to support pass-by-value
- Significantly easier to use
 - for good or ill

11 Use Cases

11.1 Optimizing `std::vector` at run time

`std::vector` can optimize moving elements into a new buffer by relying strictly on **trivial relocation** when the allocator does not implement `construct` and `destroy`. A library paper targeting the broader issue of optimizing containers for allocators that use the `construct` and `destroy` customization points will follow since that is a concern for more than just **trivial relocation**.

We find that the current specification allows for **trivial relocation** on `insert` and `erase`, although that use might produce a change of semantics that implementations using assignment prefer to avoid. Hence, we will leave the choice to implementers and their interpretation of the specification.

We expect to provide a library-specific paper to address the semantics of inserting into and erasing from a `std::vector` that is independent of **trivial relocation** concerns and that leans heavily into *replaceability*.

11.2 Optimizing `std::optional` to be trivially relocatable and replaceable

If `std::optional` is implemented with a variant member (anonymous union) and a boolean flag to indicate if the `optional` is engaged, then memberwise determination of both **trivial relocatability** and **replaceability** will produce the correct property. Typical usage might be something like the following example, which clearly shows that any `optional` implementation is going to provide implementations of all the special member functions and thus require use of both contextual keywords.

Original	Optimized
<pre>template <class T> class optional { union { T d_object; }; bool d_engaged{false}; public: using value_type = T; ... };</pre>	<pre>template <class T> class optional <u>trivially_relocatable</u> <u>replaceable</u> { union { T d_object; }; bool d_engaged{false}; public: using value_type = T; ... };</pre>

Note that to support the `constexpr` operations required by the Standard, a union-based implementation is the only known way to conform. However, if we were not concerned about `constexpr` evaluations, then we might choose to store our active element in an array of bytes. Unfortunately, adding the `trivially_relocatable` or `replaceable` properties to the class definition will give our class that same property — *even when the array member is used as storage for a type without those properties* — since an array of `std::byte` is both **trivially relocatable** and **replaceable**.

This problem can be resolved in several ways, but the key is to include a data member that is *conditionally trivially relocatable* or *replaceable*. This resolution is most easily achieved by adding, to the class, an empty data member that ideally can preserve the object layout and ABI.

```
template <bool = true>
struct OptionallyRelocatable {};
```

```

template <>
struct OptionallyRelocatable<false> {
    ~OptionallyRelocatable(){}
};

static_assert( std::is_trivially_relocatable_v<OptionallyRelocatable<>>);
static_assert(!std::is_trivially_relocatable_v<OptionallyRelocatable<false>>);

static_assert( std::is_replaceable_v<OptionallyRelocatable<>>);
static_assert(!std::is_replaceable_v<OptionallyRelocatable<false>>);

```

Original

Optimized

```

template <class T>
class optional {

    alignas (T)
    std::byte d_object[sizeof (T)];

    bool      d_engaged{false};

public:
    using value_type = T;

    ...
};

```

```

template <class T>
class optional
    trivially_relocatable
    replaceable {
    alignas (T)
    std::byte d_object[sizeof (T)];
    union {
        bool      d_engaged{false};
        OptionallyRelocatable<
            std::is_trivially_relocatable_v< T>
            &&std::is_replaceable_v< T>> _;
    };

public:
    using value_type = T;

    ...
};

```

Note that in the above implementation, even though we have made a union to contain our empty conditionally relocatable object, the `d_engaged` member will always be active. A similar conditional *replaceable* object would have the same implementation and be simple to add as well.

12 Implementation Experience

An implementation of this proposal is available as a fork of Clang and can also be accessed on [Compiler Explorer](#).

In addition to the handling of the new keywords and class properties, the implementation relies on built-in type traits for `is_trivially_relocatable` and `is_replaceable`, which are not different than other type traits of the same nature

Our Clang implementation of `trivial_relocate` is implemented in terms of `memcpy`. We did not add the necessary machinery to end and start lifetimes since that task is unsupported by the Clang front end and the LLVM optimizer (a known deficiency of LLVM rather than with our implementation). In general, starting and ending lifetimes requires an implementation to add some optimization fences so that optimizers that perform type-based alias analysis are not overly eager and inappropriately prune all code that depends on the new object lifetimes. Either way, adding such fences to an implementation that supports `start_lifetime_as` would present no notable challenges. We have not explored whether sanitizers would need to be made aware of these function semantics.

Note that Clang already supports the notion of *trivially relocatable types* in production, although with no opt-in mechanism. This property is used in the implementation of `std::vector` in `libc++` (once again demonstrating an industry need for this feature, as well as deployment experience with very similar ideas).

Clang also offers a `[[clang::trivial_abi]]` type attribute that allows a type to be passed in registers when its destructor/constructor pair can be replaced by a `memcpy`. Types with that attribute can be passed in a register, which affects calling convention, and therefore ABI.

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 `trivially_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 we cannot create a pointer-to-data-member to one; they deliberately escape 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 wording is subtle and can entrap 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. 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 exclusion can be skirted using `const_cast` if doing so would not introduce undefined behavior.

13.6 Can types that are not implicit-lifetime types be trivially relocatable?

Yes, and our experience tells us to expect the majority of types, even those that own resources and have nontrivial move constructors and destructors, to still be *trivially relocatable*.

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 since all the needed data for indirection could be stored as offsets instead of direct pointers. However, whether all implementations could use such a layout or are able to switch to such a layout is unclear. Forcing this support might also require an ABI break.

In our opinion, this low-level behavior should be kept consistent across platforms, rather than left as an unspecified QoI concern, since 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 consistency must be maintained 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 since virtual function-table implementations do not take a pointer back into the class, so the vtable pointer can be safely relocated.

13.8 What happens if a relocate operation throws?

Relocation operations must be no-fail, so they do not permit exceptions; if a relocate operation were allowed to fail, whether the failed state had 0, 1, 2, or more valid objects would be unknowable, essentially leaving the program in an undefined state that cannot be cleaned up correctly, which is a significant problem with objects holding resources like a locked mutex.

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 a notion that we have been familiar with since C++17 when *mandatory copy elision* started propagating non-copyable 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 via a backdoor method. 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 by [CWG1734] to exclude types that deleted all copying operations and which landed in C++17.

13.10 Can the compiler transform argument passing with trivial relocation?

As currently specified, we do not yet enable such support. We believe that this could be accomplished with the appropriate allowances (which already exist for trivially copyable types), but significant work in platform ABIs would be needed to make this happen, similar to what is needed to support Clang's `[[trivial_abi]]` attribute.

To enable bitwise parameter passing, such as through registers, for *trivially relocatable types*, we would need to enable the compiler to freely create extra instances of our objects when passing arguments and return results from functions, which would then enable a compiler to pass the data itself via a register. Importantly and unlike for trivially copyable types (which have trivial destructors), major changes would be needed to ensure that the receiver of the final object is aware that it is now responsible for destruction of that object since currently the creator of parameters is responsible for their destruction on many ABIs.

A separate proposal for argument passing by relocation was offered in [P2839R0] but was not reviewed favorably on its initial presentation to EWG.

13.11 Can the Standard Library containers use this new feature internally?

Yes, where the current specification is permitted to use move construction to **relocate** an object (e.g., when growing or when moving objects within a `vector`), this feature can be used instead for *trivially relocatable types*.

A common misconception implies that `vector` is required to use assignment when inserting into or erasing from a `vector` (other than at the back). This requirement is not, however, explicitly specified in the Standard. The misunderstanding stems from a number of places, which are addressed individually in the subsections below.

However, even if an implementation is *allowed* to switch from assignment to relocation for arbitrary *trivially relocatable* types, it would likely choose to do so for only such types that are also *replaceable* in order to avoid silently changing behavior for customers relying on such types.

13.11.1 Complexity constraints

The first source of this misunderstanding is that people incorrectly consider the requirement to be implied from (23.3.11.5 [vector.modifiers]p5), which states for `vector::erase`:

Complexity: The destructor of T is called the number of times equal to the number of the elements erased, but the assignment operator of T is called the number of times equal to the number of elements in the `vector` after the erased elements.

This complexity existed in C++98, and the only revision has been a change in C++11 where the text “assignment operator” was updated to “move assignment operator.” Note that `vector::insert` has no such complexity requirement; it is specified only for the `vector::erase` operation.

The misconception also comes from the following sentence in (16.3.2.4 [structure.specifications]p7):

Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees meet the requirements.

This statement is *not*, therefore, a mandate from the Standard that calls to `vector::erase` shall use the assignment operator as long as the implementation performs as well as or better than the specified complexity. Given that the `trivially_relocate` function as specified in this paper is guaranteed to perform a copy of bytes of the object representation, it must outperform the complexity requirement, and the Standard, therefore, permits implementations to use the `trivially_relocate` function for `vector::erase` operations.

13.11.2 Precondition specifications

The second source of this misunderstanding stems from phrases such as the following in (23.2.4 [sequence.reqmts]p29):

```
a.insert(p, rv)
```

Preconditions: T is *Cpp17MoveInsertable* into X. For `vector` and `deque`, T is also *Cpp17MoveAssignable*.

Effects: Inserts a copy of `rv` before `p`.

Although this specification requires that statements of the form `t = rv` be well-formed, it does *not* impose any limitations on implementations to use assignment when moving objects around internally.

Although the requirement that a type be *Cpp17CopyAssignable* or *Cpp17MoveAssignable* does impose semantic requirements on the assignment operator(s), the requirements are vague and specified in terms of a notion of “value” that is not defined in the Standard; see (16.4.4.2 [utility.arg.requirements]tab:cpp17.moveassignable). This requirement was added in C++11 and has not been revisited since then.

The above explanation refers to `vector::insert(p, rv)`, but the same argument applies to similar preconditions on other member functions. Observe that the postconditions are identical for all sequence containers, including those, such as `list`, that do not require *Cpp17MoveAssignable* as a precondition.

13.11.3 Conclusion

In other words, although most implementations of `vector::erase` and `vector::insert` currently use assignment, which is generally assumed the most efficient approach currently available, implementations are under no obligation whatsoever to do so. The various member functions of `vector` guarantee only that values will be moved around but grant implementations complete freedom as to how that action should be performed, whether by means of (move) assignment, (move) construction, or any other mechanism. Implementations will, therefore, be permitted to perform this move by means of `trivially_relocate` for types that are *trivially relocatable*.

In fact, some implementations avoid using assignment for some operations (for reasons of efficiency); see the linked examples for [GCC](#) and [LLVM](#).

Note that all the comments above apply equally to `deque` as well as to `vector`.

Note also that this lack of a clear requirement exposes an existing ambiguity for `vector::insert` and `vector::erase` operations where, for the contained type, move-assign plus destroy is not equivalent to destroy plus move-construct. That ambiguity is an issue that exists at the moment, and while we might address it with a future, orthogonal proposal, a solution is not required for **trivial relocation** as specified by this paper.

Similarly, we might choose to clarify the complexity and requirements clauses above at some point in the future, but that clarification is not required by this proposal and has been left for another time.

13.12 Do implementations need to mark classes `trivially_relocatable` to benefit?

No, although some classes will need to be annotated to qualify as *trivially relocatable*. For example, the most common implementations of `std::array`, `std::pair`, and `std::tuple` will be implicitly *trivially relocatable* if all their members are *trivially relocatable*. `std::vector` can safely be marked as *trivially relocatable* if its allocator and pointer types are *trivially relocatable*. `std::list` might be marked as *trivially relocatable* if it allocates its tail node but not if the tail node is embedded in the object representation itself.

Once we establish a policy of how much we want to guarantee and how much we want to leave open to implementer choice, a follow-up paper will address desired guarantees for **trivial relocatability** in the Standard Library.

13.13 Can I mark as trivially relocatable a type that is not replaceable?

Yes! For example, this would be appropriate for types having data members that are references or using `std::pmr::polymorphic_allocator` or any other type that does not propagate on `swap`.

13.14 Can I mark a type as replaceable but not trivially relocatable?

Yes! This proposal does not offer any immediate advantages for doing so, but we expect to build on **replacement** to optimize other features, such as assignment, in the future.

13.15 What happened to the predicates for the contextual keywords?

An earlier version of this proposal included the option to add a predicate following each of the new contextual keywords to activate or inhibit their behavior. This feature was dropped for introducing too much complexity, including a new vexing parse to resolve, and having vague semantics when the predicate is `false` but the implicit specification would have been `true`. Given the rarity of such cases and the relative simplicity of the library-based workaround above, we chose to keep the core proposal as simple as possible, following EWG guidance.

13.16 Why is copy-replacement unsupported?

In practice, only **replaceability** of objects of type `T` from `x` values of type `T` seems relevant to the operations we are likely to optimize. We have, therefore, simplified the design to focus solely on such **replacement** (which could be termed move-replacement were we being pedantic) and not overcomplicate the language or users' lives by adding even more properties to consider.

13.17 Why is marking a class that can *never* be trivially relocatable not ill-formed?

A class with a virtual base class can never be *trivially relocatable*, so why is adding the `trivially_relocatable` identifier to that class not ill-formed?

This case is still well-formed, but the class will indeed never be *trivially relocatable*, and the type trait will deterministically always return `false`. However, this type may also be used as a base class or data member when instantiating a class template, and we do not want to add complexity by considering such special-case instantiations as well-formed when the original case need not be marked as ill-formed.

However, the deterministic case of a direct virtual base class would make for a useful compiler warning. The more general case of a data member or nonvirtual base class not being relocatable (or *replaceable*) is deliberately not an error since we want to support different implementations of the same type that have different properties; e.g., different implementations of `std::list` choose different trade-offs on how to store the sentinel node marking the end of the list, yet some of those choices are *trivially relocatable* and some are not. We want to avoid the inconsistency of deterministically flagging an error when compiling a class with a `std::list` data member in some Standard Library implementations and not in others.

13.18 Why is there no `is_trivially_replaceable` trait?

A common use case is to require types that satisfy both `is_trivially_relocatable<T>` and `is_replaceable<T>`. We could consider whether this use case occurs frequently enough that adding another trait that is the logical conjunction of the two would be valuable.

We opted to omit this trait from our proposal since such a trait is not primitive to the Core-language design of this paper and could easily be added as an amendment in an LEWG follow-up paper well within the timeframe of C++26 if desired.

The lack of a core type category named *trivially replaceable* is another reason to defer to a follow-up paper, and we would be consuming that potential for future vocabulary for a pure library extension. Making that choice before advancing this paper is unnecessary.

Finally, we must recognize that a type that is both *trivially relocatable* and *replaceable* does not have a *trivial* replacement operation. The functionality that such a type enables is to turn a rotate or shift operation into a bitwise one without a change in semantics compared to using assignment for such an operation, but no single **replacement** operation is a bitwise one since that would fail to free resources owned by the original object in the target location.

13.19 Is it UB to mark a nonconforming type as trivially relocatable?

First, the compiler has no way to validate that our class's constructors and destructor do not maintain an invariant that is not relocatable, so the compiler will trust us and enable the type trait. This in itself is not UB, but UB will likely follow when some library code makes a transformation that causes our invariant, such as an internal pointer, to no longer hold. Such UB will occur in the subsequent library call, not in the class definition.

13.20 Is it UB to mark a nonconforming type as replaceable?

Just as erroneously marking a type as *trivially relocatable* can lead to undefined behavior in library calls, so can marking a type as *replaceable*. However, where **replaceability** is used as a constraint without **trivial relocation**, there remain reasonable implementations that do not incur UB. For example, if operations are logged, then the act of writing to a log is typically an observable side effect. Library code that transforms between assignment and destroy-then-construct will have an observable change of behavior, such as the suggested logging, but such changes do not in themselves constitute undefined behavior. The creator of the affected class must decide whether a change of such logging behavior would be problematic and then choose whether to mark their type as *replaceable*.

13.21 Why does replaceability not require trivial relocatability?

While all specified uses of `is_replaceable` in this proposal require that the type be both *replaceable* and *trivially relocatable*, the principle underpinning **replaceability** — i.e., a consistent definition for constructors, destructor, and assignment operators — is highly relevant in a variety of places in the Standard Library. We anticipate this distinct trait being useful to library implementations today, and we expect to see wider adoption in the Standard Library specification once the trait becomes available. For example, `std::vector` expects — but does not require — that its members be *replaceable* to efficiently switch to assignment rather than destroy/construct when replacing its elements during an insert or erase operation. Motivating examples for why we might want to address this design are found in [P2959R0], although the specification of **replaceability** in this paper is now the preferred direction rather than the suggestions proposed in that paper.

13.22 Why does trivial relocation support `const` data members, but replacement does not?

Relocation creates new objects and can safely copy `const` members. **Replacement** overwrites the data in the replaced object, which cannot — and should not — replace `const` data.

13.23 Why does `[library.class.props]` explicitly call out permission to use the contextual keywords?

For the same reason we explicitly grant permission to add `noexcept` to function declarations, even before the exception specification entered the type system, and for the same reasons that implementations cannot experiment with marking functions as `constexpr` due to the observable nature with a (deliberate) lack of explicit permission.

13.24 Why are classes with virtual base classes “replaceable”? What does that even mean?

Classes with virtual bases may be *replaceable* but will never be *trivially relocatable*; just as with trivial copyability, we cannot, at this point, restrict implementations from using implementation strategies for virtual bases that require having self-referential pointers (instead of offsets) that would be invalid if simply copied to a new object.

On the other hand, **replaceability** is a relationship between a type’s constructor, destructor, and assignment operator, all of which are applicable to reason about even for a type with a virtual base class.

In practice, we expect **replaceability** to come into play most often once types like `std::vector` start to prefer relocation (even if not trivial) and use **replacement** (and assignment operators) only for types that declare, by being *replaceable*, that such a strategy is viable. Not allowing such freedom for a vector of objects with virtual base classes would be counterproductive.

14 Illustrative Examples

14.1 Standard vector

When we add the constraints that the Standard imposes on `std::vector`, we find that **replaceability** becomes a useful property. For both insertion and erasure, the Standard likes to assume that elements are *replaceable*, i.e., assignment is interchangeable with destroy-then-move-construct. Within that guarantee, the Standard Library vector can use relocation per our custom vector example, but for types that are relocatable but not *replaceable*, matters become more complicated. That topic will be the subject of a separate paper specific to vector, which is necessary regardless of whether we support relocation in C++26. Having the ability to detect *replaceable* types would be extremely helpful for that follow-up paper.

14.2 Conforming implementation of a trivially relocatable `std::optional`

The following implementation of `optional` satisfies the C++ Standard specification for the members that it implements and provides a minimal test driver. This implementation uses the new feature macro to ensure that the code compiles with both C++23 and C++26 and is *trivially relocatable* if and only if its element type is *trivially relocatable*.

To implement the `constexpr` members, the implementation is required to use a union to represent its internal state when engaged²:

```
#include <cassert>
#include <iostream>
#include <memory>
#include <new>
#include <type_traits>
#include <utility>

template <class T>
class optional
    trivially_relocatable
    replaceable
{
    union {
        T d_object;
    };
    bool    d_engaged{false};

    constexpr T const * address() const noexcept
    { return ::std::addressof(d_object); };

    constexpr T      * address()      noexcept
    { return ::std::addressof(d_object); };

    template<class... Args>
    constexpr void do_emplace(Args&&... args) {
        ::new(address()) T(std::forward<Args>(args)...);
        d_engaged = true;
    }

public:
    using value_type = T;
};
```

²This implementation can be seen compiling on Compiler Explorer here: [compiler-explorer](https://godbolt.org/z/9Kd9z9).

```

constexpr optional() noexcept {}

constexpr optional(optional const & other)
: d_engaged{other.d_engaged} {
    if (d_engaged) {
        ::new(address()) T( other.value() );
    }
}

constexpr optional(optional&& other)
    noexcept(std::is_nothrow_move_constructible_v<T>)
: d_engaged{other.d_engaged}
{
    if (d_engaged) {
        ::new(address()) T( std::move(other).value() );
    }
}

template<class U = T>
    requires (std::is_constructible_v<T, U>
        && !std::is_same_v<std::remove_cvref_t<U>, optional>)
constexpr
explicit(!std::is_convertible_v<U, T>)
optional(U&& arg) {
    do_emplace( std::forward<U>(arg) );
}

constexpr ~optional() {
    static_assert(std::is_replaceable_v< optional>== std::is_replaceable_v< T>);
    static_assert(
        std::is_trivially_relocatable_v< optional>== std::is_trivially_relocatable_v< T>);

    if (d_engaged) {
        d_object.~T();
    }
}

constexpr optional& operator=(optional const & rhs);

constexpr optional& operator=(optional && rhs)
    noexcept(std::is_nothrow_move_assignable_v<T>
        && std::is_nothrow_move_constructible_v<T>) {
    std::cout << "Assignment\n";
    if (!d_engaged) {
        if (rhs.d_engaged) {
            do_emplace( std::move(rhs.value()) );
        }
    }
    else if (!rhs.d_engaged) {
        d_object.~T();
        d_engaged = false;
    }
    else {

```

```

        value() = rhs.value();
    }
    return *this;
}

template<class U = T>
constexpr optional& operator=(U && arg) {
    std::cout << "Assignment\n";
    if (!d_engaged) {
        do_emplace( std::forward<U>(arg) );
    }
    else {
        d_object = std::forward<U>(arg);
    }
    return *this;
}

constexpr T const * operator->() const noexcept
{ assert(d_engaged); return address(); }

constexpr T      * operator->()      noexcept
{ assert(d_engaged); return address(); }

constexpr T const & operator*() const & noexcept
{ assert(d_engaged); return d_object; }
constexpr T      & operator*()      & noexcept
{ assert(d_engaged); return d_object; }
constexpr T      && operator*()      && noexcept
{ assert(d_engaged); return std::move(d_object); }
constexpr T const&& operator*() const&& noexcept
{ assert(d_engaged); return std::move(d_object); }

constexpr explicit operator bool() const noexcept
{ return d_engaged; }
constexpr bool      has_value() const noexcept
{ return d_engaged; }

constexpr T const & value() const &
{ assert(d_engaged); return d_object; }
constexpr T      & value()      &
{ assert(d_engaged); return d_object; }
constexpr T      && value()      &&
{ assert(d_engaged); return std::move(d_object); }
constexpr T const&& value() const&&
{ assert(d_engaged); return std::move(d_object); }
};

constexpr int number(int n) {
    optional<int> x{n};
    return x.value();
}

int a[number(5uz)];

```

```

int main() {
    optional<int> x;
    assert(!x);

    std::cout << "Assignments to x\n";
    x = 3;
    auto y = x;

    x = 4;
    std::cout << "swap x\n";
    std::swap(x, y);

    assert(3 == *x);
    assert(4 == *y);

    optional<std::shared_ptr<int>> p1;

    std::cout << "Assignments to p\n";
    p1 = std::make_shared<int>(3);
    auto p2 = p1;

    p2 = std::make_shared<int>(4);
    std::cout << "swap p\n";
    std::swap(p1, p2);
}

```

14.3 C++26 implementation using internal array

Using an internal array negates the ability to support `constexpr`, but this implementation strategy is used frequently for similar types in other libraries. Managing both *trivially relocatable* and *replaceable* properties with an empty member must be done with care since mistakenly disabling both properties is easy to do when intending to disable only one or the other.³

```

#include <cassert>
#include <cstddef>
#include <iostream>
#include <memory>
#include <new>
#include <type_traits>
#include <utility>

template <bool triviallyRelocatable,
          bool replaceable>
struct ConditionalProperties {};

template <>
struct ConditionalProperties<false,true> replaceable {
    ~ConditionalProperties(){}
};

template <>
struct ConditionalProperties<true,false> trivially_relocatable {
    ~ConditionalProperties(){}
}

```

³This implementation can be seen compiling on Compiler Explorer here: [compiler-explorer](https://godbolt.org/z/9Kd9z9).

```

};
template <>
struct ConditionalProperties<false,false> {
    ~ConditionalProperties(){}
};

static_assert( std::is_trivially_relocatable_v<ConditionalProperties<true,true>>);
static_assert( std::is_trivially_relocatable_v<ConditionalProperties<true,false>>);
static_assert(!std::is_trivially_relocatable_v<ConditionalProperties<false,true>>);
static_assert(!std::is_trivially_relocatable_v<ConditionalProperties<false,false>>);

static_assert( std::is_replaceable_v<ConditionalProperties<true,true>>);
static_assert(!std::is_replaceable_v<ConditionalProperties<true,false>>);
static_assert( std::is_replaceable_v<ConditionalProperties<false,true>>);
static_assert(!std::is_replaceable_v<ConditionalProperties<false,false>>);

template <class T>
class optional trivially_relocatable replaceable {
    alignas (T)
    std::byte d_object[sizeof (T)];
    union {
        bool      d_engaged{false};
        ConditionalProperties<std::is_trivially_relocatable_v<T>,
            std::is_replaceable_v<T>> enforce_properties;
    };

    constexpr T const * address() const noexcept
    { return reinterpret_cast<T const *>(d_object); };
    constexpr T      * address()      noexcept
    { return reinterpret_cast<T      *>(d_object); };

public:
    using value_type = T;

    // 22.5.3.2, constructors
    constexpr optional() noexcept = default;

    constexpr optional(optional const & other) : d_engaged{other.d_engaged} {
        if (d_engaged) {
            ::new(address()) T( other.value() );
        }
    }

    constexpr optional(optional&& other) noexcept(std::is_nothrow_move_constructible_v<T>)
    : d_engaged{other.d_engaged}
    {
        if (d_engaged) {
            ::new(address()) T( std::move(other).value() );
        }
    }

    template<class U = T>
        requires (std::is_constructible_v<T, U>

```

```

        && !std::is_same_v<std::remove_cvref_t<U>, optional>)
constexpr
explicit(!std::is_convertible_v<U, T>)
optional(U&& arg) {
    ::new(address()) T( std::forward<U>(arg) );
    d_engaged = true;
}

// 22.5.3.3, destructor
constexpr ~optional() {
    static_assert(std::is_trivially_relocatable_v<optional> ==
                  std::is_trivially_relocatable_v<T>);
    static_assert(std::is_replaceable_v<optional> ==
                  std::is_replaceable_v<T>);

    if (d_engaged) {
        address()->~T();
    }
}

// 22.5.3.4, assignment
constexpr optional& operator=(optional const & rhs);

constexpr optional& operator=(optional && rhs)
noexcept(std::is_nothrow_move_assignable_v<T>
         && std::is_nothrow_move_constructible_v<T>)
{
    std::cout << "Assignment\n";
    if (!d_engaged) {
        if (rhs.d_engaged) {
            ::new(address()) T( std::move(rhs.value()) );
            rhs.d_engaged = false;
            d_engaged = true;
        }
    }
    else if (!rhs.d_engaged) {
        address()->~T();
        d_engaged = false;
    }
    else {
        value() = rhs.value();
    }
    return *this;
}

template<class U = T>
constexpr optional& operator=(U && arg) {
    std::cout << "Assignment\n";
    if (!d_engaged) {
        ::new(address()) T( std::forward<U>(arg) );
        d_engaged = true;
    }
    else {

```

```

        *address() = std::forward<U>(arg);
    }
    return *this;
}

// 22.5.3.7, observers
constexpr T const * operator->() const noexcept
{ assert(d_engaged); return address(); }
constexpr T      * operator->()      noexcept
{ assert(d_engaged); return address(); }

constexpr T const & operator*() const & noexcept
{ assert(d_engaged); return *address(); }
constexpr T      & operator*()      & noexcept
{ assert(d_engaged); return *address(); }
constexpr T      && operator*()      && noexcept
{ assert(d_engaged); return std::move(*address()); }
constexpr T const&& operator*() const&& noexcept
{ assert(d_engaged); return std::move(*address()); }

constexpr explicit operator bool() const noexcept
{ return d_engaged; }
constexpr bool      has_value() const noexcept
{ return d_engaged; }

constexpr T const & value() const &
{ assert(d_engaged); return *address(); }
constexpr T      & value()      &
{ assert(d_engaged); return *address(); }
constexpr T      && value()      &&
{ assert(d_engaged); return std::move(*address()); }
constexpr T const&& value() const&&
{ assert(d_engaged); return std::move(*address()); }
};

int main() {
    optional<int> x;
    assert(!x);

    std::cout << "Assignments to x\n";
    x = 3;
    auto y = x;

    x = 4;
    std::cout << "swap x\n";
    std::swap(x, y);

    assert(3 == *x);
    assert(4 == *y);

    optional<std::shared_ptr<int>> p1;

```



```
std::cout << "Assignments to p\n";  
p1 = std::make_shared<int>(3);  
auto p2 = p1;  
  
p2 = std::make_shared<int>(4);  
std::cout << "swap p\n";  
std::swap(p1, p2);  
}
```

15 Proposed Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4988].

15.1 Add new identifiers with a special meaning

5.11 [lex.name] Identifiers

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

final	import	module	override	<u>replaceable</u>	<u>trivially_relocatable</u>
-------	--------	--------	----------	--------------------	------------------------------

15.2 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, replaceable class types (11.2 [class.prop]), and arrays of such types are collectively called *replaceable 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*.

15.3 Address trivial relocation of lambdas

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

- ² 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*]

- ³ The closure type is not an aggregate type (9.4.2 [dcl.init.aggr]); it is a structural type (13.2 [temp.param]) if and only if the lambda has no *lambda-capture*. 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.y) — whether the closure type is replaceable (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.

15.4 Update grammar to support memberwise contextual keywords

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* (9.2.9.5 [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-virtproperty-specifier*_{opt} *base-clause*_{opt}
class-key *attribute-specifier-seq*_{opt} *base-clause*_{opt}

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

class-property-specifier-seq:
class-property-specifier *class-property-specifier-seq*_{opt}

class-virtproperty-specifier :
final
replaceable
trivially_relocatable

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]
- 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*.

⁵ The same *class-property-specifier* shall not appear multiple times within a single *class-property-specifier-seq*.

Whenever a *class-key* is followed by a *class-head-name*, one of the identifiers `final`, `replaceable`, or `trivially_relocatable`, and a colon or left brace, the identifier is interpreted as a *class-property-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*]

^u If a class is marked with the *class-property-specifier* `final` and that class appears as a *class-or-decltype* in a base-clause (11.7 [class.derived]), the program is ill-formed.

⁶ [Note 3: Complete objects of class type have nonzero size. Base class subobjects and members declared with the `no_unique_address` attribute (9.12.12 [dcl.attr.nouniqueaddr]) are not so constrained. —*end note*]

15.5 Specification for trivially relocatable classes

Design note:

Declaring a class as trivially relocatable is possible, by means of the `trivially_relocatable` specifier, even if that class has user-provided special members. 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.

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*]

^a A class is *eligible for trivial relocation* unless it has

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

^b A class *C* is *eligible for replacement* unless it has

- a base class that is not a replaceable class, or
- a non-static data member that is not of a replaceable type,
- no eligible constructor that would be selected when an object of type *C* is direct-initialized from an xvalue of type *C*,
- no eligible assignment operator that would be selected when an object of type *C* is assigned from an xvalue of type *C*,
- no destructor.

- c A class `C` is a *trivially relocatable class* if it is eligible for trivial relocation and
 - has the `trivially_relocatable` *class-property-specifier*, or
 - is a union with no user-declared special member functions, or
 - satisfies all of the following:
 - when an object of type `C` is direct-initialized from an xvalue of type `C`, overload resolution would select a constructor that is neither user-provided nor deleted, and
 - when an xvalue of type `C` is assigned to an object of type `C`, overload resolution would select an assignment operator that is neither user-provided nor deleted, and
 - it has a destructor that is neither user-provided nor deleted.
 - d [Note 2: Accessibility of the special member functions is not considered when establishing trivial relocatability. —end note]
 - e [Note 3: A type with non-static data members that are const-qualified or are references can be trivially relocatable. —end note]
 - f [Note 4: Trivially copyable classes are trivially relocatable unless they have deleted special members. —end note]
 - g A class `C` is a *replaceable class* if it is eligible for replacement and
 - has the `replaceable` *class-property-specifier*, or
 - is a union with no user-declared special member functions, or
 - satisfies all of the following:
 - when an object of type `C` is direct-initialized from an xvalue of type `C`, overload resolution would select a constructor that is neither user-provided nor deleted, and
 - when an xvalue of type `C` is assigned to an object of type `C`, overload resolution would select an assignment operator that is neither user-provided nor deleted, and
 - it has a destructor that is neither user-provided nor deleted.
 - h [Note 5: Accessibility of the special member functions is not relevant. —end note]
 - i [Note 6: Trivially copyable classes are replaceable unless they have deleted special members. —end note]
 - ³ A class `S` is a *standard-layout class* if it:
- (3.1) ...

15.6 Add feature macros

Add a `__cpp_trivial_relocatability` feature-test macro to the table in 15.11 [cpp.predefined], set to the date of adoption.

...

15.7 Library wording

Design note: *The first paragraph explicitly captures the status quo that these class properties — the whole set specified in 11.2 [class.prop] — are deliberately left as a quality of implementation feature.*

The second paragraph addresses permission to add the new annotation wherever an implementation might find it useful, without being constrained by its absence from the library specification, much like we grant permission to add `noexcept` specifications to functions of the implementation's choosing. The specification really needs only the second paragraph, but adding a section with the first paragraph gives us somewhere to hang the wording.

16.4.6.X Properties of library classes [library.class.props]

- ¹ Unless clearly stated, it is unspecified whether any class described in Clause 17 through Clause 34 and Annex D is a trivial class, a trivially copyable class, a trivially relocatable class, a standard-layout class, or an implicit-lifetime class (11.2 [class.prop]).
- ² An implementation may add the *class-property-specifier* `trivially_relocatable` to any class whose implementation is eligible for trivial relocation.
- ³ An implementation may add the *class-property-specifier* `replaceable` to any class whose implementation is eligible for replacement.

15.8 Add new type traits

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

```
template< class T >
struct is_replaceable;

template< class T >
struct is_trivially_relocatable;

template< class T >
inline constexpr bool is_replaceable_v = is_replaceable<T>::value;

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_replaceable;	T is a replaceable type (6.8.1 [basic.types.general])	remove_all_extents_t<T> shall be a complete type or <i>cv</i> void
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 <i>cv</i> void

15.9 Specify the compiler-magic functions

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>
    T* trivially_relocate(T* begin, T* end, T* new_location);                   // freestanding

```

20.2.6 [obj.lifetime] Explicit lifetime management

```

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

```

- a *Mandates:* T is a complete type, and `is_trivially_relocatable_v<T> && !is_const_v<T>` is true.
- c *Preconditions:*
 - (c.1) — `[begin, end)` is a valid range.
 - (c.2) — `[new_location, new_location + (end - begin))` denotes a region of 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.
 - (c.3) — `(end - begin) != 1`, or `*begin` points to a complete object (6.7.2 [intro.object]).
- d *Postconditions:*
 - No effect if `new_location == begin`.
 - Otherwise, the range denoted by `[new_location, new_location + (end - begin))` contains objects (including subobjects) whose lifetime has begun and whose object representations are the original object representations of the corresponding objects in the source range `[begin, end)`. If any of the aforementioned objects is a union, its active member is the same as that of the corresponding union in the source range. If any of the aforementioned objects has a non-static data member of reference type, that reference refers to the same entity as does the corresponding reference in the source range. The lifetime of the original objects in the source range has ended.
- e *Returns:* `new_location + (end - begin)`.
- f *Throws:* Nothing.
- g *Complexity:* Linear in the length of the source range.
- h *Remarks:* No constructors or destructors are invoked.

15.9.1 Feature-test macro

Add a new `__cpp_lib_trivially_relocatable` feature-test macro in [version.syn]:

```
#define __cpp_lib_trivially_relocatable 20XXXXL // also in <memory>, <type_traits>
```

16 Acknowledgements

This document is written in Markdown and depends on the extensions in [pandoc](#) and [mpark/wg21](#), and we would like to thank the authors of those extensions and associated libraries.

The authors would also like to thank Brian Bi for his assistance in proofreading this paper, especially the proposed Core wording. Additional thanks to Jens Maurer who helped to greatly refine the wording in advance of its first Core review.

Also, this paper has been greatly improved by feedback from Arthur O’Dwyer, author of [\[P1144\]](#), 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>
- [N4988] Thomas Köppe. 2024-08-05. Working Draft, Programming Languages — C++.
<https://wg21.link/n4988>
- [P1144] Arthur O’Dwyer. `std::is_trivially_relocatable`.
<https://wg21.link/p1144>
- [P2786R6] Mungo Gill, Alisdair Meredith. 2024-05-21. Trivial Relocatability For C++26.
<https://wg21.link/p2786r6>
- [P2839R0] Brian Bi, Joshua Berne. 2023-05-15. Nontrivial relocation via a new “owning reference” type.
<https://wg21.link/p2839r0>
- [P2959R0] Alisdair Meredith. 2023-10-15. Container Relocation.
<https://wg21.link/p2959r0>
- [P3239R0] Alisdair Meredith. 2024-05-22. A Relocating Swap.
<https://wg21.link/p3239r0>