Trivial Relocatability — Comparing P1144 with P2786

Document #:	P2814R0
Date:	2023-05-09
Project:	Programming Language C++
Audience:	Evolution Incubator Library Evolution Incubator

Reply-to:	Mungo Gill <mgill83@bloomberg.net> Alisdair Meredith <ameredith1@bloomberg.net> Arthur O'Dwyer <arthur.j.odwyer@gmail.com>

Contents

1 Abstract

At Issaquah in 2023, papers [P1144R7] and [P2786R0] were presented and examined approaches to support trivial relocatability. After discussion, the EWGI decided that, given the considerable overlap between the two papers, the best approach would be to create a single paper highlighting the commonalities and differences between the two papers, so that EWG and EWGI would be better able to determine the preferred route forward. Thus, this paper seeks to identify the common aspects of [P1144R7] and [P2786R0], showing areas of commonality and areas of difference.

2 Revision history

2.1 R0: May 2023

Initial draft of the paper.

3 Introduction

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

Any **trivially copyable** type is **trivially relocatable** by default. Many other types, even those which have non-trivial move constructors and destructors, can also be safely **trivially relocated**; the bookkeeping updates skipped by not running the target object's move constructor exactly cancel out the bookkeeping updates skipped by not running the source object's destructor. This includes many resource-owning types, such as `std::vector<int>`, `std::unique_ptr<int>`, `std::string` (on libc++), `std::list<int>` (on MSVC), and `std::shared_ptr<T>`.

Both proposals allow the library programmer to *warrant* to the compiler that a type is **trivially relocatable**. Explicit warrants are rarely needed because the compiler can infer **trivial relocatability** for many types.

Note that simply doing a bitwise copy of non-**trivially copyable** objects will, as of C++23, result in undefined behavior (when the copied bytes are treated by later code as an object of the original type). Making this operation well defined for those types that opt into this behavior is one of the goals of both proposals. Another common goal is to implicitly support a wider range of **trivially relocatable** types.

4 Motivating use cases common to both proposals

4.1 Stated in both papers: Contiguous reallocation and efficient `vector` growth

Suppose one has a move-only type, `class MoveOnlyType` (for example, a unique ownership smart pointer), and one wishes to hold a vector of these types, `std::vector<MoveOnlyType>`. Simply emplacing five of these objects would require that `MoveOnlyType`'s move constructor and destructor be called seven additional times due to the vector expansion required as more elements are inserted than the capacity (using the libc++ and libstdc++ implementations of `std::vector`).

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

The authors of both [P1144R7] and [P2786R0] performed benchmarks, and both found that a speedup of between 2.5x and 3x could be achieved for this operation. ([P1144R7] tested `vector<unique_ptr<int>>::resize`, and [P2786R0] tested `vector<string>::resize`.)

4.2 Stated in [P1144R7]: Moving in-place/SBO type-erased types like `any` and `function`

Trivial relocation can be used to de-duplicate the code generated by type-erasing wrappers like `any`, `function`, and `move_only_function`. For these types, a move of the wrapper object is implemented in terms of a relocation of the contained object. (See, for example, libc++'s `std::any`.) In general, the relocate operation must have a different instantiation for each different contained type `C`, leading to code bloat. But every **trivially relocatable** `C` of a given size can share the same instantiation.

Note: Although stated in only one of the two papers, this particular example would apply to both.

4.3 Stated in [P1144R7]: Moving fixed-capacity containers like `static_vector` and `small_vector`

The move constructor of `fixed_capacity_vector<R,N>` can be implemented naïvely as an element-by-element **move** (leaving the source `vector`'s elements in their moved-from state) or efficiently as an element-by-element **relocate** (leaving the source `vector` empty).

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

Note: Although stated in only one of the two papers, this particular example would apply to both.

4.4 Stated in [P2786R0]: Moving types with sentinel nodes

Some types do not have a non-allocating empty state, so they cannot have a `noexcept` move constructor. One example is a known implementation strategy for `std::list` that always allocates at least a sentinel node (as used by the Microsoft STL among others). Lacking a non-throwing move constructor, vectors of such lists can grow only using copy-and-destroy, rather than move-and-destroy, with each copy requiring an additional allocation. However, as long as the sentinel does not maintain a back-pointer into its list object, such a type can be **trivially relocatable** as the old object immediately ends its life without running its destructor and thus does not have to restore invariants; there is no window of opportunity to access the live object in a state where it has broken invariants.

Note: Although stated in only one of the two papers, this particular example would apply to both.

5 Detailed comparison of proposed changes

5.1 New terms and definitions

Both papers introduce a number of new terms but with slightly different definitions.

- **relocate**
  - [P1144R7]: Given an object type `T` and memory addresses `src` and `dst`, the phrase "relocate a T from src to dst" means "move-construct dst from src, and then immediately destroy the object at src."
  - [P2786R0]: To **relocate** a type from memory address `src` to memory address `dst` means to perform an operation or series of operations such that an object equivalent (often identical) to that which existed at address `src` exists at address `dst`, that the lifetime of the object at address `dst` has begun, and that the lifetime of the object at address `src` has ended.
- **relocatable**
  - [P1144R7]: Any type that is both **move-constructible** and **destructible** is **relocatable**.
  - [P2786R0]: To say that an object is **relocatable** is to say that it is possible to **relocate** the object from one location to another.

- **trivially relocatable**
  - [P1144R7]: A type is **trivially relocatable** if its **relocation** operation is **trivial** (which, just like trivial move-construction and trivial copy-construction, means “tantamount to `memmove`”).
  - [P2786R0]: Conceptually, a type is **trivially relocatable** if it can be **relocated** by means of copying the bytes of the object representation and then ending the lifetime of the original object without running its destructor.

As can be seen above, [P1144R7] is focused on the public interface of the class itself, namely the existence and accessibility of move constructors and destructors, whereas in [P2786R0], the focus is on the fundamental semantics of the type itself.

5.2 New type category

Both papers introduce a *trivially relocatable type* as a new type category. * Both papers define that category in a way that can be implicitly deduced for many existing types, notably scalars and arrays, and can recursively deduce **trivial relocatability** in classes comprising only **trivially relocatable** types (with some restrictions around *user-provided* special member functions). * Both papers provide an explicit markup, using the `trivially_relocatable` token, for users to mark their own classes with *user-provided* special member functions as retaining this new property. * Both markups allow for a Boolean predicate for a class to conditionally opt-in to the new property when it is not inferred. * Both papers agree that it is undefined behavior to mark up a class whose move constructor and destructor maintain an invariant that is broken by simple bitwise movement, such as an internal pointer.

5.3 Implicit trivial relocatability

Both proposals specify a mechanism for types to be considered implicitly **trivially relocatable**.

Both papers explicitly state that a **trivially copyable** type would be **trivially relocatable** by default. However, the inverse is specifically *not* true; i.e., a **trivially relocatable** type is not *necessarily* **trivially copyable**.

However, both proposals state that a non-**trivially copyable** type *can* be implicitly **trivially relocatable** based on a number of requirements.

Most of the requirements are similar between the two proposals but with subtle yet important differences.

[**P1144R7**]	[**P2786R0**]
There are no virtual base classes.	There are no virtual base classes.
All base classes are of **trivially relocatable** type.	All base classes are of **trivially relocatable** type.
All members are either of reference type or of **trivially relocatable** type.	All *non-static data* members are either of reference type or of **trivially relocatable** type.
There are *no* user-provided destructors.	*The selected* destructor is neither user-provided *nor deleted.*
There are *no* user-provided move constructors or copy constructors.	When an instance of the type is direct-initialized from an rvalue of the same type, *the selected* constructor is neither user-provided *nor deleted.*
There are *no user-provided move assignment operators or move assignment operators.*	
There are *no virtual member functions.*	

The following are noteworthy points when comparing the two sets of conditions.

— [P1144R7] requires that the copy constructor not be *user-provided*, whereas [P2786R0] does not care about the copy constructor unless it also serves as the move constructor.

— [P1144R7] requires that there be no *user-provided* move or copy assignment operators, whereas [P2786R0] does not care about assignment operators. This turns out to be more significant than the authors of both [P1144R7] and [P2786R0] first thought; see std::swap below.

— In [P1144R7], deleted special member functions do not affect implicit **trivial relocatability**, whereas [P2786R0] requires a user to explicitly declare their type as **trivially relocatable** if the relevant members are deleted. This difference comes from [P2786R0] trying to follow the users' intent declared by their syntax: if the user intends their type to be non-relocatable, then we should not bypass that intent by means of **trivial relocation** without their express permission to do so. [P1144R7], on the other hand, sees **trivial relocation** as merely an optimization opportunity, which means it matters only for types that are movable to begin with: as long as the compiler respects the user's intent to make their type non-relocatable, it does not matter whether we consider the impossible operation 'trivial' or not.

— [P1144R7] specifies that `trivially_relocatable` attributes are ignored if the compiler infers that type is **trivially relocatable**; i.e., the attribute is only opt-in and never opt-out. [P2786R0] requires the compiler to respect a `trivially_relocatable(false)` specification, regardless of any compiler-inferred implicit **trivial relocatability**; i.e., the user specification takes priority over the default semantics.

  — In particular, [P1144R7] lets the user wrap a type not known to be trivially relocatable (e.g. `boost::shared_ptr`) in a Rule-of-Zero wrapper marked `trivially_relocatable(true)`; [P2786R0] would consider such a marking to be a diagnosable error. Vice versa, [P2786R0] lets the user wrap a type which the compiler incorrectly believes to be trivially relocatable in a wrapper marked `trivially_relocatable(false)`; [P1144R7] ignores such a marking.

  — This explains why [P1144R7] is less willing to infer **trivial relocatability** on certain types (e.g. polymorphic types, types with user-provided assignment operators): [P1144R7] provides no *escape hatch*, when the compiler infers that a type is **trivially relocatable**, to undo that inference.

5.4 Explicit trivial relocatability and its safety

Both proposals provide a mechanism to explicitly declare that a type is **trivially relocatable**.

[**P1144R7**]	[**P2786R0**]

`struct [[trivially_relocatable(true)]] C`
`{`
`    C(C&&);`
`    ~C();`
`};`
	`struct C trivially_relocatable(true)`
	`{`
	`    C(C&&);`
	`    ~C();`
	`};`

Although we consider the keyword-vs.-attribute issue to be essentially cosmetic, two other differences are extremely important.

— In [P1144R7], **trivial relocatability** is only opt-in, never opt-out. Only [P2786R0] provides a means to mark an implicitly **trivially copyable** type as not **trivially relocatable** by means of `trivially_relocatable(false)`.

— [P2786R0] explicitly states that it would be a diagnosable error to explicitly declare a type to be **trivially relocatable** where any non-static data members or base classes are not **trivially relocatable**. This approach is intended to avoid silently introducing undefined behavior where the compiler cannot see that all moving parts are **trivially relocatable** types. On the other hand, [P1144R7] allows users to mark any class as **trivially relocatable**, as long as it supports public move semantics, even if it comprises bases or data members that are not themselves **trivially relocatable**. This approach is intended to support integration with third-party libraries, where the user believes that moving the bytes of all the relevant third-party data types would be safe. P1144R8 section 3 will include further details.

5.5 New traits

Both proposals include an `is_trivially_relocatable` trait, defined almost identically. (Note that P1144R8 will update the precondition, so the definitions will be identical, although they both depend on **trivially relocatable**, and [P1144R7]'s definition of **trivially relocatable** differs from [P2786R0]'s definition of **trivially relocatable**.)

Paper	**[P1144R7]**	**[P2786R0]**
Template	`template<class T> struct is_trivially_relocatable;`	`template<class T> struct is_trivially_relocatable;`
Condition	`T` is a **trivially relocatable** type	`T` is a **trivially relocatable** type
Preconditions	`T` shall be a complete type, *cv* `void`, or an array of unknown bound	`remove_all_extents_t<T>` shall be a complete type or *cv*-`void`

[P1144R7] also defines two more traits, `is_relocatable` and `is_nothrow_relocatable` but these will be removed in P1144R8.

Paper	**[P1144R7]**	**[P2786R0]**
Template	`template<class T> struct is_relocatable;`	
Condition	`is_move_constructible_v<T>` is `true` and `is_destructible_v<T>` is `true`	
Preconditions	`T` shall be a complete type, *cv* `void`, or an array of unknown bound	

Paper	**[P1144R7]**	**[P2786R0]**
Template	`template<class T> struct is_nothrow_relocatable;`	
Condition	`is_relocatable_v<T>` is `true` and both the indicated move-constructor and the destructor are known not to throw any exceptions	
Preconditions	`T` shall be a complete type, *cv* `void`, or an array of unknown bound	

[P1144R7] also defines a concept `relocatable`.

[P1144R7]	[P2786R0]

`template<class T>`
`concept relocatable = move_constructible<T>;`
 	
If `T` is an object type, then let `rv` be an *rvalue* of type `T`, `lv` an *lvalue* of type `T` equal to `rv`, and `u2` a distinct object of type `T` equal to `rv`. `T` models `relocatable` only if
 	
* After the definition `T u = rv;`, `u` is equal to `u2`.
 	
* `T(rv)` is equal to `u2`.
 	
* If the expression `u2 = rv` is well-formed, then the expression has the same semantics as `u2.~T(); ::new ((void*)std::addressof(u2)) T(rv);`
 	
* If the definition `T u = lv;` is well-formed, then after the definition `u` is equal to `u2`.
 	
* If the expression `T(lv)` is well-formed, then the expression's result is equal to `u2`.
 	
* If the expression `u2 = lv` is well-formed, then the expression has the same semantics as `u2.~T(); ::new ((void*)std::addressof(u2)) T(lv);`	

5.6 New relocation functions

The proposed interface is very different between the [P1144R7] and [P2786R0] proposals.

— [P1144R7] proposes two key functions, `relocate_at` and `relocate`, which can, internally, make use of trivial relocatability for optimization. It also proposes convenience wrapper functions `uninitialized_relocate`, `uninitialized_relocate_n`, and `uninitialized_relocate_backward`.

— [P2786R0] proposes a key function, `trivially_relocate`, suitable only for **trivially relocatable** objects, suitable for the common use case of relocating a contiguous range of objects. and a convenience wrapper function, `relocate`, also suitable for *nothrow-move-constructible* objects.

[P1144R7]	[P2786R0]

```template<class T>```
```T *relocate_at(T* source, T* dest);```
 	
```Which is equivalent to```
 	
```struct guard {```
```   T *t;```
```   ~guard() {```
```   destroy_at(t);```
```   }```
```} g(source);```
 	
```return ::new (voidify(*dest))```
```   T(std::move(*source));```
 	
```// except that if T is trivially relocatable```
```// [basic.types], side effects associated```
```// with the relocation of the value of```
```// *source might not happen```
	```template <class T>```
	``` requires is_trivially_relocatable_v<T>```
	```constexpr```
	```T* trivially_relocate(```
	``` T* begin, T* end, T* new_location)```
	```noexcept;```
	
	```Which is equivalent to```
	
	```memmove(new_location,```
	``` begin,```
	``` sizeof(T) * (end - begin));```
	
	```// except the object lifetimes are```
	```// managed correctly; see D2786```

```template<class T>```
```T relocate(T* source);```
`````
// not to be confused with P2786 `relocate````
`````
// function, which is the same only in name
 	
Which is equivalent to
 	
T t = std::move(source);
 	
destroy_at(source);
 	
return t;
 	
// except that if T is trivially relocatable
 	
// [basic.types], side effects associated
 	
// with the relocation of the object's value
 	
// might not happen```	

The following usage examples illustrate the use of these functions.

[P1144R7]

```
{
  TriviallyRelocatable    *tr =
       new TriviallyRelocatable;
  NonTriviallyRelocatable *ntr =
       new NonTriviallyRelocatable;

  TriviallyRelocatable    *trd =
       malloc(sizeof TriviallyRelocatable);
  NonTriviallyRelocatable *ntrd =
       malloc(sizeof NonTriviallyRelocatable);

  // The following is valid and MAY OR
  // MAY NOT use memmove "under the hood":
  static_assert(
     std::is_trivially_relocatable_v<
         TriviallyRelocatable>);
  std::relocate_at(tr,  trd);

  // The following is valid and will
  // use move+destroy:
  static_assert(
     ! std::is_trivially_relocatable_v<
        NotTriviallyRelocatable>);
  std::relocate_at(ntr, ntrd);
}
```

[P2786R0]

```
{
  TriviallyRelocatable    *tr =
       new TriviallyRelocatable;
  NonTriviallyRelocatable *ntr =
       new NonTriviallyRelocatable;

  TriviallyRelocatable    *trd =
       malloc(sizeof TriviallyRelocatable);
  NonTriviallyRelocatable *ntrd =
       malloc(sizeof NonTriviallyRelocatable);

  // The following is valid and WILL ALWAYS
  // use memmove:
  static_assert(
     std::is_trivially_relocatable_v<
         TriviallyRelocatable>);
  std::trivially_relocate(tr, tr+1, trd);

  // The following assert will pass;
  // the function call is ill formed:
  static_assert(
     ! std::is_trivially_relocatable_v<
        NotTriviallyRelocatable>);
  std::trivially_relocate(ntr, ntr+1, ntrd);
}
```

In addition, both proposals suggest a number of utility functions.

— [P1144R7] proposes a family of generic algorithms to relocate a range which can be forward or bidirectional (even though the `memcpy` optimization will be practically useful only when the range is contiguous). This genericity is consistent with the existing memory algorithms such as `std::uninitialized_move`.

— [P2786R0] also proposes a wrapper function, `relocate`, implemented using `trivially_relocate` for the convenience of users wanting a single function to cater for both **trivially relocatable** and non-**trivially relocatable** types, and to support simple and practical implementations of `vector`-like types that wish to optimize on the availability of **trivial relocation**

[P1144R7]

```
template<class InputIterator,
         class NoThrowForwardIterator>
NoThrowForwardIterator
uninitialized_relocate(
    InputIterator          first,
    InputIterator          last,
    NoThrowForwardIterator result);

template<class InputIterator,
         class Size,
         class NoThrowForwardIterator>
pair<InputIterator, NoThrowForwardIterator>
uninitialized_relocate_n(
    InputIterator          first,
    Size                   n,
    NoThrowForwardIterator result);

template<class BidirectionalIterator,
         class NoThrowBidirectionalIterator>
NoThrowBidirectionalIterator
uninitialized_relocate_backward(
    BidirectionalIterator        first,
    BidirectionalIterator        last,
    NoThrowBidirectionalIterator  result);
```

[P2786R0]

```
template<class T>
requires ( (is_trivially_relocatable_v<T> &&
            !is_const_v<T>) ||
          is_nothrow_move_constructible_v<T>)
T* relocate(T* begin,
            T* end,
            T* new_location)
// not to be confused with P1144 `relocate`
// function which is the same only in name
```

5.7 Moved object lifetime

Both papers agree that `memmove` alone is insufficient for performing an in-memory relocation, as the C++ abstract machine tracks object lifetimes independently of the object representation in memory. Both papers propose Standard Library APIs to perform a bitwise relocation in a way that is exposed to the abstract machine, and both papers expect that function to be implemented simply as a bitwise copy at run time.

[P1144R7] describes its proposed `relocate` functions as "equivalent to a move and a destroy", with *permission* to elide any side effect of move construction and the destructor. Object lifetimes are not explicitly addressed by [P1144R7], but it uses the phrase "equivalent to a move and destroy" to perform the heavy lifting for unspecified compiler magic to manage the object lifetimes.

[P2786R0] proposes explicit changes to the lifetime model.

— The Standard Library `trivially_relocate` function is explicitly stated to copy the bytes of the object representation.

— Use of the `trivially_relocate` function is mandated, not optional.

— The lifetime of an object ends once it is **trivially relocated** from.

— The compiler magic to imbue life into the relocated objects is limited to the single function `trivially_relocate`.

— The user can explicitly request a **trivial relocation** as the "magic" function is specified and publicly available.

— No claim is made as to any equivalence between **trivial relocation** and "move and destroy".

— It is the responsibility of the user to ensure that the destructor is not called on the **trivially relocated-**from object; to do so would result in undefined behaviour.

6 General observations regarding the two proposals

6.1 Scope and Standard Library changes

Although both proposals have considered how the respective changes can enable further optimization by library implementation, neither proposal requires or relies upon any changes to the existing Standard Library containers and algorithms.

6.2 Difference in level of implementation

[P1144R7] has a focus on delivering a high-level library interface to users of the language and proceeds with the minimal level of detail with regard to core specification to make those library APIs implementable. This is evidenced by language talking about (1) the absence of side effects to infer that certain code transformations are possible, spread across a number of functions, and (2) a library API that works with iterator ranges as the most useful approach that fits within the standard library design.

[P2786R0] starts from the principle that a change in the abstract machine is necessary; thus it begins with a core specification that explicitly handles changes to the rules of object lifetimes and then puts all of that abstract machine magic in exactly one function that is available to library implementers and users of the library alike. The broader library support is then built on top of this function, which may be reflected in the design of the library APIs themselves. [P2786R0] aims to provide a lower-level facility, which can be used as a foundation for further work.

6.3 Interface vs. semantics

Originally, [P1144R6] was based upon relocatability semantics, which it defined to be equivalent to a move followed by destruction. That design led to the deduced relocatable property being framed in terms of the *public interface* of its bases and non-static data members, namely a publicly accessible move constructor and destructor, and the **trivially relocatable** requirements are based on those same syntactic requirements.

However, as can be seen in the more recent version, [P1144R7], that design is moving toward the idea that a type can be **trivially relocatable** without being **move constructible** and **destructible** (just as it can by **trivially copyable** without being **copy assignable**), although the **relocation** operation itself is still defined in terms of move-and-destroy.

Although in [P1144R6] explicitly marking a type as **trivially relocatable** if it does not have a public interface that supports regular move semantics was an error (albeit one for which no diagnostic is required), that restriction has been removed in the latest draft, (P1144R8).

[P2786R0] leads with the idea that triviality is key and that all other trivial semantics in the language are based upon the trivial semantics of bases and non-static data members, not syntax. This is a similar approach to **trivial copyability**. Hence, public access to move functions does not matter, as relocation of such types uses the trivial semantic instead for such types.

[P2786R0] explicitly allows the user to specify that a type is **trivial relocatable**, overriding the default, as long as all of its bases and members are **trivially relocatable** in turn.

An alternative way to present this is that [P1144R7] provides **trivial relocation** as an optimization enabled only for movable types, and [P2786R0] proposes a low-level language primitive to relocate objects in memory, regardless of their movability.

6.4 Predictable specification without deference to QoI

[P1144R7] permits the library to use bitwise copies to perform relocation operations but does not mandate it. That optimization is left as a QoI feature of the relocating library functions. Misuse by annotating types that are not suitable for relocation often leads to UB or to programs that are ill-formed, no diagnostic required.

[P2786R0] leaves no room for QoI, fully specifying observable behavior and requiring (diagnosable) ill-formed programs when the facility is misused.

An alternative way to consider this is that, in [P1144R7], `relocate_at` will use either move-and-destroy or `memmove`. (Although `memmove` will not be used for types that are not **trivially relocatable**, it is not guaranteed to be used for types that are.)

Conversely, in [P2786R0], `trivially relocate` can only be used for **trivially relocatable** types and is guaranteed to use `memmove`. (Attempting to use `trivially_relocate` for a non-**trivially relocatable** type is a diagnosable error.)

6.5 Revocation of trivial relocatability

Both proposals provide two ways to observe trivial relocatability: directly via `is_trivially_relocatable`, and indirectly via the library functions that do relocation operations.

[P1144R7] doesn't permit compiler-inferred implicit **trivial relocatability** to be revoked. A type annotated `[[trivially_relocatable(false)]]` will still be visibly `is_trivially_relocatable`. The compiler's decision does not affect whether `std::relocate_at` is well-formed or not, although it may affect the runtime performance of `std::relocate_at` (which is affected by QoI anyway).

[P2786R0] does permit compiler-inferred implicit **trivial relocatability** to be revoked. A type annotated `trivially_relocatable(false)` will not be `is_trivially_relocatable` (even if it remains `is_trivially_copyable`!). Since it will not is_trivially_relocatable, it will not a valid argument for `std::trivial_relocate`.

6.6 Respecting encapsulation of bases and members

[P1144R7] adopts a "trust the user" approach, even when that leads to undefined behavior. [P1144R7] allows a type to break the encapsulation of its members and **trivially relocate** them even when they do not have that property (thus allowing a type that is neither `T` nor a friend of `T` to skip user-defined special member functions of `T` when `T` is not **trivially relocatable**).

[P2786R0] considers that marking as **trivially relocatable** a type whose bases and non-static data members are not, themselves, **trivially relocatable** should be a diagnosable error.

This is perhaps best illustrated via an example. Suppose a user has examined the code of `thirdparty::shared_ptr` and concluded that **trivially relocating** `thirdparty::shared_ptr` would be safe, but the providers of that library have not annotated it as such.

[P1144R7]

```
// in <thirdparty.h>
namespace thirdparty {
    class shared_ptr {
        ...
    };
}

// The following assert will pass:
static_assert(
    ! std::is_trivially_relocatable_v<
         thirdparty::shared_ptr<OtherClass>>);

// P1144 would consider the following valid
// even though thirdparty::shared_ptr is not
// marked as trivially relocatable.
class [[trivially_relocatable]] MyClass {
  private:
    thirdparty::shared_ptr<OtherClass> data_p;
  public:
    ...
};
```

[P2786R0]

```
// in <thirdparty.h>
namespace thirdparty {
    class shared_ptr {
        ...
    };
}

// The following assert will pass:
static_assert(
    ! std::is_trivially_relocatable_v<
         thirdparty::shared_ptr<OtherClass>>);

// P2786 would consider the following
// ill formed because thirdparty::shared_ptr
// is not marked as trivially relocatable.
class MyClass trivially_relocatable {
  private:
    thirdparty::shared_ptr<OtherClass> data_p;
  public:
    ...
};
```

See also P1144R8 section 3 example 4.

6.7 Immovable but relocatable types

[P1144R7] treats **relocation** as "move plus destroy," and **trivial relocation** as an optimization of **relocation**; so it does not admit the possibility of a type that is **relocatable** without being **movable**. Indeed, [P1144R7]'s `concept relocatable` is syntactically equivalent to `concept move_constructible`, and [P1144R7]'s `relocate_at` is defined as "Equivalent to" a move-construction followed by a destruction. It is ill-formed to call `relocate_at` on a non-**movable** type (e.g. one whose move constructor is private).

[P2786R0] treats **trivial relocation** as a separate primitive operation, and permits calling `trivially_relocate` on any **trivially relocatable** type, regardless of whether that type is **movable** (e.g. even if its move constructor is private). This allows creating a type which is relocatable only by `trivially_relocate` and not by any other means.

6.8 Assignment operators

Although both proposals agree that **trivial relocation** can be used in place of move construction followed by destruction, a fundamental area of disagreement is the relationship between move assignment and **relocation/trivial relocation**.

[P1144R7] proposes that **relocation** can be used in place of an appropriately structured series of move assignments. This widens the scope of what can be optimized by means of trivial relocation (e.g., `std::swap`) but narrows the set of types for which trivial relocation can be used as ending an object's lifetime and creating a new object with a new value in its place is, for many types with non-salient properties such as the `pmr` types, not equivalent to assigning a value to the original object.

[P2786R0] proposes that **trivial relocation** *cannot* be used in place of move assignment plus destruction. This significantly widens the scope in terms of allowing `pmr` types to be **trivially relocatable** but does restrict the places where such trivial relocation can be used.

Note for context that the `bsl::vector` implementation of Bloomberg's open-source BDE library has an optimization that goes beyond [P2786R0] and will, *only* when moving objects around within its own internal storage, use `memmove` for **trivially relocatable** types in places where assignment would otherwise be used, such as in the implementations of `insert` and `erase`. However, BDE will never, for allocator-aware objects, use `memmove` in place of any assignment operation moving an object into, out of, between, or outside containers, as guaranteeing that the source and destination objects will have the same allocator is not possible.

6.8.1 `pmr` types

A notable area of disagreement between the authors of both [P1144R7] and [P2786R0] arises with `pmr` types. The scoped allocator model, typified by `std::pmr` types, is a strong motivator behind [P2786R0], notably for examples like `std::pmr::vector<std::pmr::string>`. As such, the authors are careful to specify the feature to support their primary use case.

Under [P2786R0], `pmr` types can be considered **trivially relocatable**, as the assignment operator will still be used for all move assignments; thus `propagate_on_container_move_assignment` and `propagate_on_container_swap` will be respected.

On the other hand, [P1144R7] explicitly requires certain behavior out of relocatable types' assignment operators, which means either that `pmr` types cannot be considered **trivially relocatable** or that `pmr` types can be considered **trivially relocatable** only under certain runtime conditions. This is difficult to square with the fact that `is_trivially_relocatable_v<pmr::string>` needs to be a compile-time constant. It is unsatisfactory to have `pmr::string` never advertise itself as **trivially relocatable**, but it is also unsatisfactory for it to always appear **trivially relocatable** but sometimes violate the type's semantic requirements (note in particular `propagate_on_container_move_assignment`).

6.8.2 `std::swap`

Part of the motivation for [P1144R7] including move assignment within the umbrella of trivial relocation is to enable support for making `std::swap` more efficient. [P2786R0] does not permit `std::swap` to use `memmove`. The most significant difference between [P1144R7] and [P2786R0] is whether trivial swappability implies **trivial relocatability** or whether **trivial relocatability** implies trivial swappability.

[P1144R7] reports significant benefits optimizing `std::swap` as three `relocate` operations, which is why it requires **trivially relocatable** types to have appropriate assignment operators as well. The underlying assumption is that destroy-then-move-construct has the same behavior as move-assign for supported types. This restriction immediately excludes a set of types very important to the authors of [P2786R0], namely types using scoped allocators (or any other allocator that does not propagate on `swap`). Prominent examples of types that do not satisfy this second constraint include class types with reference data members, all polymorphic types, and types with non-propagating allocators (such as all of `pmr`).

[P2786R0] asserts that all trivially swappable types are **trivially relocatable**, as one can effectively perform a trivial `swap` and then end the lifetime of the original object without running its destructor, per the **trivial relocate** semantics where destructors do not run. The converse is not true though, with a variety of types that would satisfy a primitive **trivial relocation** specification but not a primitive trivially swappable specification.

The focus of [P2786R0] is purely on whether moving a sequence of bytes (once) produces a valid object representation, modelling move-then-destroy for movable types but also allowing relocation of immovable types with the right properties. One of the underlying assumptions of [P2786R0] is that **trivial relocation** is a fundamental operation in the abstract machine, and trivial swap would be a similar fundamental primitive, much as the library continually runs into the design constraint that `swap` is essentially a primitive operation, which the Standard Library expresses as a compound operation through multiple moves.

7 Summary of key differences

P1144R7	**P2786R0**
Considers relocation as equivalent to move + destroy.	Considers trivial relocation separate from move + destroy.
• Defines **relocation** as equivalent to move + destroy, and **trivial relocation** as an optimization of that. • **Relocatability** is considered a feature of a type's public interface. • The "equivalence" wording is used to address lifetime issues. • It is ill-formed to mark a non-movable type as **trivially relocatable** (removed in R8).	• Defines only **trivial relocation**, and does not consider the non-trivial case. • **Trivial relocatability** is based on the semantics of bases and members. • Enhancements are required to the lifetime model wording. • Non-movable non-copyable types can be made **trivially relocatable**. • This is similar approach to **trivial copyability**.
Considers move assignment to be in scope.	Considers move assignment to be out of scope.
• **Relocation** (**trivial** or not) can replace certain move assignments. • `pmr` and allocator aware types will behave in surprising ways when used in containers and algorithms. • Fewer types can be considered **trivially relocatable** and **trivial relocation** can be used in more places.	• **Trivial relocation** cannot be used in place of move assignment. • `pmr` and allocator aware types work as-is. • More types can be considered **trivially relocatable** and **trivial relocation** can be used in fewer places.
Adopts a "trust the user" approach.	Makes many potential misuses into compilation errors.
• There is greater opportunity for mis-annotation resulting in Undefined Behaviour. • The user has greater flexibility.	• Many potential mis-annotations are strictly ill-formed. • The user has less flexibility.
Provides a suite of user-focused generic algorithms.	Is focused purely on the core language changes required.
• Two "core" functions, `relocate` and `relocate_at`, are provided to support **relocation**. • Library functions may perform either **trivial relocation** or move+destroy depending on type and QoI. • A family of utility functions to support iterator-based relocations is provided.	• A single "core" function `trivially_relocate` constrained for **trivially relocatable** types is provided. • The `trivially_relocate` function will only ever perform **trivial relocation**. • No support for iterator-based relocation is provided.
Provides no opt-out.	Provides an opt-out.
• Compiler-inferred **trivial relocatability** cannot be removed from a type.	• `trivially_relocatability(false)` will remove **trivial relocatability** from a type (even if that type is **trivially copyable**).

8 Acknowledgements

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

The author of [P1144R7] has also created three blog posts with additional background of the motivations behind some of the design decisions in that paper:

[trivial_swap_x_prize] [relocate_algorithm_design] [sharp_knife_dull_knife]

9 References

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

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

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

[P2786R0] Mungo Gill, Alisdair Meredith. 2023-02-11. Trivial relocatability options.
https://wg21.link/p2786r0

[relocate_algorithm_design] Arthur O'Dwyer. 2023. STL algorithms for trivial relocation.
https://quuxplusone.github.io/blog/2023/03/03/relocate-algorithm-design/

[sharp_knife_dull_knife] Arthur O'Dwyer. 2023. Should the compiler sometimes reject a [[trivially_relocatable]] warrant?
https://quuxplusone.github.io/blog/2023/03/10/sharp-knife-dull-knife/

[trivial_swap_x_prize] Arthur O'Dwyer. 2023. Trivial relocation, std::swap, and a $2000 prize.
https://quuxplusone.github.io/blog/2023/02/24/trivial-swap-x-prize/