ISO/IEC JTC1/SC22/WG21

P3858R0

Audience: LEWG

A Lifetime-Management Primitive for Trivially Relocatable Types

David Sankel [1], Jon Bauman [2], Pablo Halpern [3]

[1] Adobe dsankel@adobe.com

[2] Rust Foundation jonbauman@rustfoundation.org

[3] Halpern-Wight, Inc. phalpern@halpernwightsoftware.com

October 6, 2025

Abstract

"Trivial Relocatability For C++ 26"[1] introduced mechanisms for the identification and tagging of types whose objects can be "trivially" relocated from one memory address to another, as well as standard library functions that perform this relocation. A call to `std::trivial_relocate` performs a logically atomic operation whereby an object's representation is copied, its lifetime is ended at the original location, and its lifetime is restarted at the target location, without invoking any constructors or destructors. Useful as they are, these standard library functions are insufficient for important use cases where the three component operations must be separated by intervening code, such as `realloc` support, value representation serialization, and cross language interoperability. We propose to complete the trivial relocation function set with the addition of a single function template, `std::restart_lifetime`, that addresses these unsupported use cases.

Context

```
// A trivially relocatable, but not trivially copyable, type.
class Foo { /*...*/ };

// Create a foo sequence with a single element using Microsoft's specialized mimalloc allocator.
void* foo_sequence_buffer = mi_malloc_aligned(sizeof(Foo), alignof(Foo));
Foo* foo_sequence = new (foo_sequence_buffer) Foo();

// Extend the sequence reusing the same memory if possible
foo_sequence_buffer = mi_realloc_aligned(foo_sequence, sizeof(Foo)*2, alignof(Foo));
new (foo_sequence_buffer+sizeof(Foo)) Foo();
foo_sequence = (Foo*)foo_sequence_buffer;
```

Before

```
foo_sequence[0].bar(); // Undefined behavior
```

After

```
// Restart lifetime of relocated elements
std::restart_lifetime(foo_sequence[0]);

foo_sequence[0].bar(); // Okay
```

1. Introduction

It is a common yet unspecified property that for many types, an object can be relocated with a `memcpy` of its underlying bytes. Although the standard guarantees this only for the small number of *trivially copyable* types, virtually all C++ compilers support `memcpy`-relocation of non-self-referential types. Many applications have taken advantage of this property for performance optimizations and a number of libraries have emerged that attempt to surface this functionality in a generic way.[1]

After much debate, the committee added a form of this functionality to the working draft[3] with "Trivial Relocatability For C++ 26"[1][2]. An important trade-off in this design is that qualifying types may be "trivially" relocated using only the `trivially_relocate` function; `memcpy` will not suffice.

```
// Foo is a trivially relocatable, but not
// trivially copyable, type.
class Foo { /*...*/ };
static_assert(
     std::is_trivially_relocatable_v<Foo>()
 && !std::is_trivially_copyable_v<Foo>());

void f() {
  alignas(Foo) char x1_buffer[sizeof(Foo)],
                    x2_buffer[sizeof(Foo)],
                    y1_buffer[sizeof(Foo)],
                    y2_buffer[sizeof(Foo)];

  // Relocating using std::memcpy results in
  // undefined behavior.
  Foo* x1 = new (x1_buffer) Foo();
  std::memcpy(&y1_buffer, x1, sizeof(Foo));
  Foo* y1 = reinterpret_cast<Foo*>(y1_buffer);
  y1->bar(); // Undefined behavior

  // Relocating using std::trivially_relocate
  // works as expected.
  Foo* x2 = new (x2_buffer) Foo();
  Foo* y2 = std::trivially_relocate(
    x2,
    x2+1,
    reinterpret_cast<Foo*>(&y2_buffer));
  y2->bar(); // Okay
}
```

Among other benefits, this type-aware design enables the ARM64e ABI which encodes an object's address in its virtual table (vtable) pointer making `memcpy`-relocation impossible for polymorphic types on this platform[3]. The requirement to call `std::trivially_relocate` provides an opportunity for the standard library to perform "fixups" on these vtable pointers.

While `std::trivially_relocate` suffices for many use cases and neatly handles the ARM64e platform, other important use cases remain unaddressed. We propose to complement `std::trivially_relocate` with another function, `std::restart_lifetime`, that addresses these use cases.

2. Key `std::trivially_relocate` limitations

2.1. `realloc` use case

Allocation libraries often feature a reallocation function (such as C's `realloc`) that attempts to resize a given memory block[4]. It either extends the block in-place or moves its contents to a new, larger allocation, freeing the original block in the process.

Reallocation serves as an important performance optimization for high-performance, low-level code that dynamically resizes arrays. By taking advantage of the allocation library's knowledge of available space after the originally allocated block, expensive copy operations and fragmentation can be avoided.

However, because these reallocation functions potentially `memcpy`-relocate objects, they may only portably be used with *trivially copyable* types and `std::trivially_relocate` will not help.

2.2. Serialization use case

In-memory databases[9] and tiered caching systems frequently relocate data structures from memory to disk and back again. Unfortunately, this operation is possible only for *trivially copyable* types due to the lack of sufficient library primitives for *trivially relocatable* types.

2.3. Specialized `memcpy` use case

A tuned memory copy operation can produce a 10% speedup over `std::memcpy`, and heterogenious memory systems require an alternative[5]. `std::trivially_relocate`'s coupling of the physical moving of an object with restarting its lifetime makes it

[1] See [1] and [2] for a survey of such libraries.

[2] See [2] for a notable design alternative that was considered, but ultimately rejected.

[3] This is a memory safety vulnerability mitigation. See [4] and [5] for details.

[4] See mimalloc[6], umm_malloc[7], and tcmalloc[8] for some examples.

[5] See "Going faster than memcpy"[10] and CUDA's `cudaMemcpy`[11] for some notable examples.

is impossible to portably take advantage of these mechanisms with trivially relocatable types.

2.4. Rust-interop use case

One of the major challenges for high-performance interop is language differences in how memory for object storage is handled. For Rust and C++ to use the same memory for an object that either language can access, we must account for the differing models of ownership and relocation. While current practice tends to use indirection so that the underlying storage is opaque across the language boundary, this has a cost both in performance and ergonomics.

Moves in C++ are non-destructive, whereas Rust's ownership model is based on destructive moves and it is a compile-time error to refer to a moved-from location. This facilitates Rust's value-oriented semantics where all assignments (including parameters and return values) transfer ownership[6]. This is a fundamental piece of the memory-safe model of default Rust. To facilitate efficient moves, Rust defines their semantics as a bitwise copy[7]. In other words, all Rust objects are *trivially copyable* in the C++ sense. The fact that Rust objects cannot be self-referential[8] facilitates this. Rust has no analog to a C++ move constructor, meaning there is no opportunity for additional code that may be added to `trivially_relocate` in C++ to run following a Rust move. Without the addition of `std::restart_lifetime`, only *trivially copyable* C++ types could be passed to Rust by value. Other types must be allocated on the heap, which is a significant performance penalty, or be pinned[9], which has a significant ergonomics penalty.

3. `restart_lifetime`

We propose a `restart_lifetime` function that fits within the `start_lifetime_as` series of functions. It allows us to separate the "memory copying" aspect of relocation from restarting the object's lifetime at the new memory address.

Here is an implementation of `std::trivially_relocate` using `restart_lifetime` as a lower-level primitive.

```
template<class T>
requires /* ... */
T* trivially_relocate(T* first, T* last, T* result)
{
  std::memcpy( result,
               first,
               (last-first)*sizeof(T));
  for(size_t i = 0; i < (last-first); ++i)
    std::restart_lifetime(result[i]);
}
```

This separation of concerns allows developers to copy an object's value representation to a new location by any means and then use it from the new location after a call to `std::restart_lifetime`. This enables all the usecases highlighted in Section 2.

Here is an example of using `std::restart_lifetime` to roundtrip a `Foo` object from main memory to GPU memory.

```
void * host_buffer = /*...*/
void * device_buffer = /*...*/

// Create a `Foo` object in host memory
Foo* x = new (host_buffer)[sizeof(Foo)];

// Move it to CUDA memory
cudaMemcpy( device_buffer,
            host_buffer,
            sizeof(Foo),
            cudaMemcpyHostToDevice );

// ... reuse host_buffer for other purposes

// Move it back to host memory
cudaMemcpy( host_buffer,
            device_buffer,
            sizeof(Foo),
            cudaMemcpyDevicetoHost );

// Restart the object's lifetime on the host
x = std::restart_lifetime<Foo>(host_buffer);

// ... continue using *x
```

3.1. Addressing Rust-interop

Since `std::trivially_relocate` can be decomposed into a bitwise copy followed by `std::restart_lifetime`, and it's only necessary to make sure that the latter occurs before accessing the potentially authenticated C++ vtable pointers, there is an opportunity to lazily perform fixups on the C++ side. For example, say we have a polymorphic class hierarchy implemented in C++:

```
class Shape {
public:
  virtual float area() const = 0;
```

[6] Rust can also use references for "borrows", which provide either shared immutable access or exclusive mutable access to a value with a compiler-checked lifetime

[7] See https://doc.rust-lang.org/stable/std/marker/trait.Copy.html#whats-the-difference-between-copy-and-clone and https://doc.rust-lang.org/stable/std/ptr/fn.read.html#ownership-of-the-returned-value

[8] Raw, unsafe pointers and `Pinned` data are two ways Rust can express self-referential types

[9] See https://doc.rust-lang.org/stable/std/pin/index.html

```
  virtual ~Shape() = default;
};

class Circle final : public Shape {
public:
  Circle(float radius);
  float area() const override;
private:
  float m_area;
};
```

We'd like to interact with this API idiomatically within Rust:

```
let a = Circle::new(1.0);
let b = Circle::new(2.0);
a = b;
println("a's area: {}", a.area());
```

To do so, we first observe that `Shape` and `Circle` are trivially relocatable and replaceable types. We denote the alignment of Circle as `CIRCLE_ALIGNMENT` and its size as `CIRCLE_SIZE`. Now we can define the Rust-side `Circle` type:

```
#[repr(C)]
#[repr(align(CIRCLE_ALIGNMENT))]
struct Circle {
    data: Cell<[MaybeUninit<u8>; CIRCLE_SIZE]>,
}
```

where `data` holds the bit representation of the object[10]. Let's now turn to `Circle`'s methods. Their implementations are essentially boilerplate that delegates to corresponding C functions prefixed with `c_`:

```
impl Circle {
    fn new(radius: f32) -> Circle {
        let mut c = MaybeUninit::uninit();
        unsafe { c_create(
            c.as_mut_ptr() as *mut c_void,
            &radius as *const f32
                    as *mut c_void)};
        unsafe { c.assume_init() }
    }
    fn area(&self) -> f32 {
        let mut result = MaybeUninit::uninit();
        unsafe { c_area(
            result.as_mut_ptr() as *mut c_void,
            self as *const Circle
                 as *mut c_void)};
        unsafe { result.assume_init() }
    }
}
```

These C functions are implemented as follows:

```
void c_create(void* result, void* radius) {
    new (result) Circle(
      *static_cast<float*>(radius));
}

void c_area(void* result, void* circle) {
    Circle* data =
      std::restart_lifetime<Circle>(circle);
    *static_cast<float*>(result) = data->area();
}
```

With the addition of such easily generated wrapper code, efficient and ergonomic access from Rust can be achieved. Furthermore, on platforms where `std::restart_lifetime` is a no-op, there is no performance penalty.

4. Implementation

On most platforms, the implementation of `std::restart_lifetime` is a no-op. The exception is ARM64e where polymorphic types and types with polymorphic data members require special handling as follows:

1. If the object is polymorphic, set its vtable pointers and cryptographically sign them according to the object's new address.
2. Recursively do the same for the object's fields.

Implementation experience is in progress but we do not foresee difficulties with either the no-op implementation on most platforms or the more sophisticated ARM64e implementation[11].

5. Other considerations

5.1. Will this undermine ARM64e security guarantees?

Prior drafts of this proposal suggested implementations that indiscriminately sign existing vtable pointers without a priori verifying their validity. This resulted in a Return-Oriented Programming (ROP) Gadget exploitable by hackers using buffer overruns.

The current proposal avoids this attack by overwriting the vtable pointers to their correct values, thus eliminating the possibility that an attacker could set them to arbitrary memory locations.

[10] See https://doc.rust-lang.org/stable/std/cell/struct.Cell.html, https://doc.rust-lang.org/stable/std/mem/union.MaybeUninit.html

[11] Note that at the time of this writing, there are no implementations of `std::trivially_relocate` for ARM64e. See https://github.com/llvm/llvm-project/pull/144420 for a work in progress

5.2. Why is this being brought up now (and not earlier)?

Although trivial relocatability has been discussed for many years, issues related to ARM64e were brought forward only recently[12]. It took us the time since then to understand the issues and formulate a suitable solution.

5.3. Is this a bug fix or a feature?

An important aspect of the trivial relocatability's feature design is its basis operations. The basis operations provided in the C++26 working draft did not satisfy important use cases and that was discovered only recently. Consequently, it can be argued that this contribution is a bug fix as the intention is to ship a *complete* trivial relocatability solution in C++26.

5.4. Is this critical for C++26?

Whether or not this feature is considered a bug fix, it can be argued that it is critical this functionality be shipped in C++26 due to the urgency of memory safety initiatives. The ability to call existing C++ code ergonomically from Rust is a critical to the memory safety roadmaps of many major corporations. Delaying this functionality by three years may force undesirable choices like depending on non-portable undefined behavior for interop or a strong push to rewrite existing C++ code that works just fine.

5.5. Should this *replace* `trivially_relocate` instead of complement it?

Some have argued that `std::trivially_relocate` is unnecessary in the standard library because it can be implemented in terms of `std::restart_lifetime`, while `std::relocate` covers the most common use cases.

We disagree with this assertion. `std::trivially_relocate` has legitimate use cases, one being a replacement for similar operations provided by existing library relocation solutions.

6. Alternatives considered

6.1. Pass in origin pointer

We also considered an alternative formulation of `restart_lifetime` that accepted the object's original location in addition to the new location. Its definition is provided below.

```
// Not proposed
template<class T>
T* restart_lifetime(uintptr_t origin,
                    void* p) noexcept;
```

Mandates: `is_trivially_relocatable_v<T>` && ! `is_const_v<T>` is `true`.

Preconditions:

- [`p, (char*)p + sizeof(T)`) denotes a region of allocated storage that is a subset of the region of storage reachable through [basic.compound] `p` and suitably aligned for the type `T`.
- The contents of [`p, (char*)p + sizeof(T)`) is the value representation of an object `a` that was stored at `origin`.

Effects: Implicitly creates an object *b* within the denoted region of type `T` whose address is `p`, whose lifetime has begun, and whose object representation is the same as that of *a*.

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

The rationale was that on ARM64e, the `origin` pointer could be used to validate the vtable pointers in the new location before re-signing them. This design, however, is significantly more complicated than our proposal and is considered overly tailored to ARM64e.

6.2. Extension for inter-process communication

Another alternative we considered was to generalize `restart_lifetime` to support reconstituting an object from its value representation across different processes. This would involve completely restoring the invisible parts of an object (such as vtable pointers) in a new context based solely on its byte representation. However, we concluded that this approach is fraught with difficulty for several reasons.

First, such a function would be very difficult, if not impossible, for a compiler to optimize into a no-op, as it cannot assume that the source and destination contexts are identical. Second, C++ currently lacks a mechanism to formally describe or mandate a "same layout" guarantee for non-standard-layout types across different compiler versions, platforms, or even separate compilations of the same program; future reflection capabilities might provide a path toward establishing such guarantees, but this is beyond the scope of this paper. Finally, describing a mandate for a "valid member state" is problematic. The language does not provide a standardized way to describe or check class invariants, making it difficult to specify

[12]Their first mention was in May of 2025 with [2]

preconditions for a function that must reconstitute an object from a potentially untrusted byte stream without invoking a constructor.

While a more powerful cross-process variation of `restart_lifetime` could be considered in the future, we wanted this to be a minimalist proposal. Our goal is to address the immediate and well-understood use cases of in-process relocation without venturing into the more complex domain of general-purpose serialization.

6.3. Names

Another name we considered was `start_lifetime_at` due to its similarty to the `start_lifetime_as` function group. The authors do not have strong preferences between `start_lifetime_at` and `restart_lifetime`.

7. Wording

Add to end of [obj.lifetime].

```
template<class T>
T* restart_lifetime(void* p) noexcept;
template<class T>
volatile T* restart_lifetime(volatile void* p)
                                    noexcept;
```

Mandates: `is_trivially_relocatable_v<T> && ! is_const_v<T>` is `true`.

Preconditions:

- [`p, (char*)p + sizeof(T)`) denotes a region of allocated storage that is a subset of the region of storage reachable through [basic.compound] `p` and suitably aligned for the type `T`.
- The contents of [`p, (char*)p + sizeof(T)`) is the value representation of an object *a* that was stored at another address.

Effects: Implicitly creates an object *b* within the denoted region of type `T` whose address is `p`, whose lifetime has begun, and whose object representation is the same as that of *a*. If *a* was still within its lifetime, its lifetime is ended.

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

8. Conclusion

The `std::trivially_relocate` primitive, while valuable, is insufficient for a number of important, real-world use cases involving `realloc`-driven optimizations, serialization, and cross-language interoperability. We propose a minimal, orthogonal primitive, `std::restart_lifetime`, which decomposes relocation into its constituent parts: a byte-wise copy and a subsequent lifetime restart. This separation of concerns directly enables the aforementioned use cases. Crucially, it provides a portable and ergonomic pathway for interoperability with other languages, such as Rust, supporting critical industry-wide memory safety initiatives. Given its importance as a completion of the trivial relocatability feature set and its low implementation cost, we believe this proposal should be considered for C++26.

9. Acknowledgments

We would like to thank Oliver Hunt for his review from an ARM64e security perspective, Jens Maurer for wording assistance, and the P2786 authors for valuable feedback.

Bibliography

[1] Alisdair Meredit, Mungo Gill, Joshua Berne, Corentin Jabot, Pablo Halpern, and Lori Hughes, "Trivial Relocatability for C++26," Feb. 2025. Accessed: Jul. 15, 2025. [Online]. Available: https://wg21.link/p2786r13

[2] Arthur O'Dwyer *et al.*, "`std::is_trivially_relocatable`," May 2025. Accessed: May 13, 2025. [Online]. Available: https://wg21.link/p1144R13

[3] ISO/IEC JTC 1/SC22/WG21, "Working Draft, Programming Languages - C++ (N5014)," Aug. 2025. [Online]. Available: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2025/n5014.pdf

[4] Oliver Hunt and John McCall, "A gentle introduction to pointer authentication," Jun. 2018. Accessed: Jul. 15, 2025. [Online]. Available: https://waffles.dog/d3751r0.html

[5] John McCall and Ahmed Bougacha, "arm64e: An ABI for Pointer Authentication." Oct. 22, 2019. Accessed: Jul. 15, 2025. [Online]. Available: https://llvm.org/devmtg/2019-10/slides/McCall-Bougacha-arm64e.pdf

[6] Microsoft, "mimalloc." [Online]. Available: https://github.com/microsoft/mimalloc

[7] Ralph Hempel, "umm_malloc." [Online]. Available: https://github.com/rhempel/umm_malloc

[8] Google, “tcmaloc.” [Online]. Available: https://github.com/google/tcmalloc

[9] Wikipedia contributors, “In-memory database.” [Online]. Available: https://en.wikipedia.org/wiki/In-memory_database

[10] Dheeraj Rajaram Reddy, “Going faster than memcpy.” [Online]. Available: https://squadrick.dev/journal/going-faster-than-memcpy

[11] NVidia, “`cudaMemcpy`.” [Online]. Available: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/html/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html