A Lifetime-Management Primitive for Trivially Relocatable Types

P3858R1

David Sankel, Jon Bauman, Pablo Halpern

2025-11-05 LEWG Presentation

Trivial Relocation (TR) basis operation

```
template<class T>
T* trivially_relocate(T* first, T* last, T* result);
```

- *Mandates* T is trivially relocatable
- *Postconditions* `result` contains objects whose lifetime has begun and whose object representations are the original object representations of `[first,last)`. Lifetime of objects at `[first,last)` have ended.

Works well for the vector resize use-case

Do we need additional basis operations?

Unaddressed use-cases

Most of these use-cases are addressed by existing (non-portable) library-based trivial relocation facilities.

realloc

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

// Create a foo sequence with a single element using a specialized 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_buffer, sizeof(Foo)*2, alignof(Foo));
foo_sequence = (Foo*)foo_sequence_buffer;
::new (&foo_sequence[1]) Foo();

foo_sequence[0].bar(); // UNDEFINED BEHAVIOR
```

`serialization`

In-memory databases/tiered caching systems

- Frequently relocate data structures from memory to disk and back again
- Possible only for *trivially copyable* types due to the lack of sufficient library primitives for *trivially relocatable* types

Specialized `memcpy` use case

- Tuned memory copy operation can produce a 10% speedup over `std::memcpy`
- Heterogeneous memory systems (e.g. Cuda)

Rust-interop use case

- C++ types that aren't *trivially copyable* have poor ergonomics (via pinning) or incur performance penalties (heap allocation)
- There's no way to extend the fast, ergonomics to *trivially relocatable* types with the existing primitives

Type erasure use case

Create a type erased wrapper that is trivially relocatable

- Use small buffer optimization when its wrapped type is trivially relocatable
- Use heap allocation otherwise

Currently not possible

`restart_lifetime`

- An additional TR primitive
- Decouples memory copying from restarting lifetimes
- General usecase:
  - `memcpy` object representation to a new location
  - use `restart_lifetime` to restart the object's lifetime a the new location

trivially_relocate implemented with restart_liftime

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

Revisit `realloc`

```
1  // A trivially relocatable, but not trivially copyable, type.
2  class Foo { /*...*/ };
3
4  // Create a foo sequence with a single element using a specialized allocator.
5  void* foo_sequence_buffer = mi_malloc_aligned(sizeof(Foo), alignof(Foo));
6  Foo* foo_sequence = ::new (foo_sequence_buffer) Foo();
7
8  // Extend the sequence reusing the same memory if possible
9  foo_sequence_buffer = mi_realloc_aligned(foo_sequence_buffer, sizeof(Foo)*2, alignof(Foo));
10 foo_sequence = (Foo*)foo_sequence_buffer;
11 ::new (&foo_sequence[1]) Foo();
12
13 // Restart lifetime of relocated elements
14 std::restart_lifetime<Foo>(foo_sequence[0]);
15 foo_sequence[0].bar(); // Okay
```

Other use-cases are handled in a similar way

From Arthur O'Dwyer's Thoughts on P3858R0:

```
template<class T>
  requires std::is_trivially_relocatable_v<T>
void my_sort(T *first, T *last) {
  auto *cmp = +[](const void *va, const void *vb) {
    const T& a = *std::restart_lifetime<T>(static_cast<void*>(va));
    const T& b = *std::restart_lifetime<T>(static_cast<void*>(vb));
    return a < b;
  };
  std::qsort(first, last - first, sizeof(T), cmp);
}
// and fall back to a type-specific template for the general case
```

	C++26 draft	**+ `restart_lifetime`**	**bitwise relocation**
Object lifetime based	X	X	
Polymorphic type support	X	X	
Vec resize use case	X	X	X
`realloc` use case		X	X
Custom `memcpy` use case		X	X
Rust interop use case		X	X
Type erasure use case		X	X
TR defn. consistent w/ existing practice			X
Defines previously undefined behavior			X

`restart_lifetime` issue

- It might be possible to abuse in ARM64e to work around security guarantees
  - ARM64e is an important Apple platform
  - Future architectures may move in the same direction
- Because `restart_lifetime` can modify "non-value" bytes of target object, its use in a multi-threaded context is limited

Possible solutions

- Accept the gadget trade off
- Restrict trivial relocatability to non-polymorphic types
- Restrict `restart_lifetime` to non-polymorphic types
- Have `restart_lifetime` take an additional origin pointer parameter

For C++26?

- **Pro**
  - Complete basis operation set for TR
  - Address many more use-cases
  - Advances memory safety plans of several organizations
- **Con**
  - Risky given the short time frame
  - Path to handle ARM64e-like platforms is unclear