Library API for Trivial Relocation

Minimal library interface for the core language feature

P3241R0 presented to LEWG
April 9, 2024

Alisdair Meredith
ameredith1@bloomberg.net

TechAtBloomberg.com

What is Trivial Relocation?

A solution to an old problem

- A popular “optimisation” is to move objects around by copying their representation to a new location, e.g., through `memcpy` or `memmove`
  - Assumes the moved objects do not have internal references
  - This is well-defined for trivial types
  - This is UB for non-trivial types, as there is no way to breathe life into the objects at their new location
- Trivial relocation is a new facility to inform the compiler about the object lifetimes
  - Call the *magic* function `trivially_relocate` to both move the bytes, and update lifetimes
  - Add syntax for the compiler to verify which types are eligible for trivial relocation

Design Principles for EWG

Concerns that guided our proposal

- Feature for users, not just the Standard Library
- A formal specification for object lifetimes
- Minimal design to enable library extension
- Predictable behavior; no freedom for QoI in core semantics
- Guard against accidental UB
- Trust, but verify: can explicitly mark types trivially relocatable *unless* they have non-trivially-relocatable member objects

Design Principles for LEWG

Concerns that guided our proposal

- Keep the API small, to leave space for pure library extensions
- Not a general purpose relocation facility
  - Build the larger API using trivial relocatability as a possible optimization
- Defer full library analysis until core feature accepted, as library is HUGE
- Be consistent with existing similar facilities
  - Type traits, magic functions, etc.
- Constrain and mandate to protect against UB

Core Language Changes

Trivially Relocatable Types

- A trivially relocatable type is:
  - A scalar type
  - A *trivially relocatable class* type
  - An array of trivially relocatable types
  - Cv-qualified version of any of the above

Trivially Relocatable Classes

Introducing syntax

- A class can be marked with the contextual keyword `trivially_relocatable`
  - Optional constant predicate: `trivially_relocatable(is_relocatable<T>())`
- A class is *ineligible for trivial relocatability* if it has
  - A virtual base class
  - Any base classes that are not trivially relocatable
  - Any non-static data members *of a non-reference type* that are not trivially relocatable
- It is ill-formed to mark a class as trivially relocatable if it is ineligible for trivial relocatability
  - i.e., `trivially_relocatable(`*bool-expression*`)` must evaluate to `false` if present

Trivially Relocatable Classes

Implicit without syntax

- A trivially relocatable class is:
  - Marked as trivially relocatable, or
  - All of the following:
    - Is not marked `trivially_relocatable(false)`
    - Is not ineligible for trivial relocatability
    - Does not have a user-provided or deleted move constructor
    - Does not have a user-provided or deleted destructor

Proposed Library Changes

The Whole Library API

```
// Type trait
template<class T>
struct is_trivially_relocatable;

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

// Magic function template with constraints clause for reference
template<class T>
  requires (is_trivially_relocatable_v<T> && !is_const_v<T>)
T* trivially_relocate(T* begin, T* end, T* new_location) noexcept;
```

Deferred LWG Proposals

Pure library extensions built on top of trivial relocatability

- Non-trivial relocatability
  - P2967 is a first draft of the larger interface
  - P1144 has a longer history with a more relaxed specification
- Optimising relocation within a container (P2959)
  - `std::vector`, `std::deque`, etc., are over-specified to use assignment
  - P2959 is a first draft address the larger semantic issues that go beyond just trivial relocation

API Design 1/3

Key questions we considered developing this proposal

- Type trait vs. Concept
  - Choosing type trait for consistency with every other trivial trait
  - Not opposed to a follow-up paper proposing trivial concepts for all trivialities, but not treating trivial relocation in isolation
- Range interface, not single object
  - Typically used with ranges rather than single objects
  - Minimises the number of *magic* functions
  - Can form a range of a single object, so not strictly needed
  - We are expecting follow-up papers in St Louis addressing higher level (non-trivial) library relocation facilities, e.g., P1144, P2967

API Design 2/3

Key questions we considered developing this proposal

- `!is_const` constraint to protect against UB
  - If you relocate a local variable, you cannot replace it without UB if that variable is const
  - If you know you are not falling into UB, can still `const_cast` for desired behavior — this is a guard rail, not a barrier
- Not a `constexpr` function
  - Implementation experience that this is hard
  - Not needed for a constexpr `vector`, but…
  - will need to guard use of the `trivial_relocate` function with `if consteval {`*compile-time*`} else {`*runtime*`}`

API Design 3/3

Key questions we considered developing this proposal

- function is not declared as `noexcept`
  - Preconditions on input and output ranges

Quick Examples

Example of `trivially_relocatable` class

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

struct MyRelocatableType trivially_relocatable : BaseType {
  // class definition details
  MyRelocatableType(MyRelocatableType&&); // user supplied
    // Having a user-provided move constructor `MyRelocatableType` would not
    // be trivially relocatable by default. The `trivially_relocatable`
    // annotation trusts the user's specification that this type can indeed
    // be trivially relocated.
};

struct MyNonRelocatableType : BaseType {
  // class definition details
  MyNonRelocatableType(MyNonRelocatableType&&); // user supplied
  // Having a user-provided move constructor `MyNonRelocatableType` is
  // not trivially relocatable.
};

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

Example using `trivally_relocate`

```
template <class T>
void MyVector::reserve(std::size_t new_capacity) {
    if (new_capacity <= d_capacity) return;

    T* new_buffer = d_alloc.allocate(new_capacity);
    if constexpr (std::is_trivially_relocatable_v<T>) {
        std::trivially_relocate(d_buffer, d_buffer + size, new_buffer);
        std::swap(buffer, d_buffer);
    }
    else if constexpr (std::is_nothrow_move_constructible_v<T>) {
        std::uninitialized_move(d_buffer, d_buffer + size, new_buffer);
        std::swap(buffer, d_buffer);
        std::destroy(buffer, buffer + size);
    }
    else if constexpr (std::is_copy_constructible_v<T>) {
        // exception safe copy code
    }
    else {
        // exception safe throwing-move code
    }
    d_alloc.deallocate(buffer, std::exchange(d_capacity, new_capacity);
}
```

Time for Feedback