P3233R0
Issues with P2786

Giuseppe D'Angelo, KDAB
<giuseppe.dangelo@kdab.com>

WG21 St. Louis, MO, USA
June 2024

Introduction

- Two competing proposals for (trivial) relocation

- P2786R4 (Mungo Gill, Alisdair Meredith)
  - Approved by EWG for C++26 in Tokyo
  - Lots of follow-ups
- P1144R10 (Arthur O'Dwyer)
  - Not seen in Tokyo?

- Other papers:
  - P2814R0: comparison paper between P2786/P1144
  - P3236R1: “Please reject P2786 and adopt P1144” (myself + many authors)
  - P3278R0: “Analysis of interaction between relocation, assignment, and swap” (Nina Ranns)

- Some other proposals, not relevant right now

What is P3233 about?

- P3233 is not a counter-proposal
- My take on the issue
  - Personal / as a contributor to the Qt Project
- Criticism, first and foremost to some design decisions of P2786
  - Alas, not always constructive
- Not a blanket endorsement of P1144 either
  - Some aspects of P1144 could also be refined

I’m seeking direction

- Many design decisions around trivial relocatability are subtle and complex
- I’m really not sure if this room will have the necessary data for a well-informed decision right here, right now
  - Or if instead the discussion will get stuck in some of these subtle issues, and we’ll lack consensus to move in any specific direction
  - Some of the issue I raise need further exploration

- I don’t want to just “rant” about things:
  - I’ll end with a bunch of ideas/“action points”
  - Possible candidates for polls

The status quo: P2786 approved in Tokyo

- C++26 will have a new type property called “trivially relocatable” (TR)
  - TR types: scalars, TR classes, arrays of TR, cv-TR types
- A class can be manually marked as TR by using the **`trivially_relocatable`** contextual keyword
  - Optionally with a bool argument
- An *unmarked* class is automatically TR under certain conditions:
  - All subobjects are TR, or of reference type
  - No virtual bases
  - Non-deleted, non-user-provided destructor
  - Move constructible via a non-deleted non-user-provided constructor
- Enforcement: if a class is manually marked as TR, and has a non-TR subobject or a virtual base, the program is ill-formed.

The status quo (cont.)

- A new library function:
  `std::trivially_relocate(T *begin, T *end, T *out)`
  that perform trivial relocation

- “Compiler magic”:
  - End lifetime of objects in the source range
  - Start lifetime of objects in the destination
  - Copy their representations (i.e. memcpy)

- Constrained to work only on TR types

The main use case for P2786 trivial relocation

- The #1 use case for P2786’s definition of TR is to be able to optimize vector reallocation.

The main use case for P2786 trivial relocation

- Vector reallocation is implemented like this:
    a. new storage is acquired;
    b. existing objects are move-(if-noexcept) constructed into the new storage;
    c. old objects are destroyed;
    d. old storage is deallocated;
    e. bookkeeping is updated.

- With TR we can turn b. + c. into "one call to memcpy" → huge speedup

- Wide applicability: `std::vector<unique_ptr<T>>`, `std::vector<std::vector<X>>`, etc.

- The current requirements for TR types directly support this use case.

Agenda

Issues with P2786: agenda

1. Lack of/questionable relocation semantics
2. Lack of Library API
3. Missed optimizations
4. Enforcement model
5. Conclusions

Several of these issues are intertwined, which further complicates discussion and analysis.

Unclear Relocation Semantics

What is relocation?

- P2786 does not define what relocation is; only <u>trivial</u> relocation

- This is an asymmetry with existing properties which also exist in a trivial-less version
  - E.g. copy constructible ⇔ trivially copy constructible
  - Destructible ⇔ trivially destructible
  - Granted, “trivial” semantics need fixing (CWG2463, P3279)

- Only exception: trivially copyable, “umbrella” property

What is relocation?

- P2786's model **does not state** that a relocation is "move construction + destruction of the source"

- This makes it hard to reason about the impact of this type property with class design (RO5)
    - It surely has **some** interaction with RO5, given that the presence of user-defined moves or destruction disables automatic TR

- This is at odds with existing practice (Qt, Folly, BSL, …), cf. P1144

- This is at odds with providing higher-level library features like `std::uninitialized_relocate(begin, end, out)`
    - Which trivially relocates TR types, and does "something else" for non-TR types

Non-movable types can be trivially relocated

- It is possible to create TR types which are not movable:

```
struct S trivially_relocatable {
  S();
  S(const S &) = delete;
  S(S &&) = delete;
  ~S();
};
```

- Why is this allowed? What does it mean? Is it a “destructive move” for immovable objects? (With dynamic storage duration)?
  - Is this type an “abomination”?
  - Is TR a brand new primitive operation?
  - Is TR going to interfere with future work on destructive moves?

Class authors can “lie”

- Why is it allowed for a user to lie?

```
struct S trivially_relocatable(false) {
  int i;
};
```

- This class would be “naturally” TR, but the user is allowed to say it isn’t. Why is that a good thing?
    - At odds with any other similar type property

Trivially copyable isn’t a subset of trivially relocatable

- This has “interesting” consequences. For instance, is possible to create Trivially Copyable (TC) types which are not TR:

```
struct TC trivially_relocatable(false) {
  int a, b;
};
```

- In the adopted model the sets of TC and TR types are merely intersecting
  - Conflicts with existing practice
  - In P1144, TC is a subset of TR
- Unclear why this is allowed, instead of being ill-formed or ignored
  - No need of perpetuating broken precedents
- This results in vexing / duplicated code (see next slide)

Trivially copyable isn’t a subset of trivially relocatable

Example:

```
template <typename T>
vector<T>::reallocate_impl(size_t new_capacity)
{
    assert(m_size <= new_capacity);
    T *new_storage = allocate(new_capacity);

    // Need to handle TR and TC separately, because it's
    // not allowed to call trivially_relocate on a non-TR type,
    // even if it's TC!
    if constexpr (std::is_trivially_relocatable_v<T>) {
        std::trivially_relocate(m_begin, m_begin + m_size, new_storage);
    } else if constexpr (std::is_trivially_copyable_v<T>) {
        std::memcpy(new_storage, m_begin, m_size * sizeof(T));
    } else if constexpr (std::is_nothrow_move_constructible_v<T>) {
        std::uninitialized_move(m_begin, m_begin + m_size, new_storage);
        std::destroy(m_begin, m_begin + m_size);
    } else {
        // …
    }

    deallocate(m_begin);
    m_begin = new_storage;
    m_capacity = new_capacity;
}
```

Unclear behavior for slicing / polymorphic classes

Polymorphic classes can be implicitly TR:

```
struct Base {
  virtual void f();
  int a;
};

struct Derived : Base {
  void f() override;
  int b;
};

static_assert(std::is_trivially_relocatable_v<Base>);    // OK
static_assert(std::is_trivially_relocatable_v<Derived>); // OK
```

Unclear behavior for slicing / polymorphic classes

- While it makes sense to want to use TR to reallocate a `std::vector<Derived>`, the semantics break down for single-object operations
    - When these operations involve static/apparent types

- This is a “known” problem for these kinds of operations/optimizations:
    - E.g.: given a type T which is trivially copyable, and contiguous input/output ranges of T, one *cannot* use `memcpy` to implement a `std::copy` of 1 object because of potentially overlapping subobjects
    - https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108846

Unclear behavior for slicing / polymorphic classes

Example:

```
struct Base {
  virtual void f();
  int a;
};

struct Derived : Base {
  void f() override;
  int b;
};

Base *source = new Derived;
Base *target = allocate(sizeof(Derived));

std::trivially_relocate(source, source + 1, target);
```

Unclear behavior for slicing / polymorphic classes

```
Base *source = new Derived;
Base *target = allocate(sizeof(Derived));

// What is the behavior here?
std::trivially_relocate(source, source + 1, target);
```

- If relocation were defined in terms of moves and destructions, we could claim this is UB because it's "destroying" a `Derived` object through a `Base` pointer, and `Base` does not have a virtual destructor…
- Again: the lack of a precise specification of what "relocation" is makes it hard to reason about this.

Unclear behavior for slicing / polymorphic classes

Let's add the missing virtual destructor:

```
struct Base {
  virtual ~Base() = default; // still TR: defaulted dtor
  virtual void f();
  int a;
};

struct Derived : Base {
  void f() override;
  int b;
};
```

Unclear behavior for slicing / polymorphic classes

```
Base *source = new Derived;
Base *target = allocate(sizeof(Derived));

std::trivially_relocate(source, source + 1, target);
```

- This code is still extremely problematic: the TR operation is copying `Derived`’s vtable pointer into a `Base` object!
  - If someone calls `target->f()`, this will be dispatched through `Derived::f()`, with `this` pointing to a `Base` object!
- This should still be UB!
  - The enforcement model is not preventing this!

Slicing in P1144

- In P1144 polymorphic classes are not TR
- Slicing (via relocation) a class without a virtual destructor is UB
  - Polymorphic or non-polymorphic, TR or non-TR
  - This is just matching core language
- Slicing (via relocation) a class with a virtual destructor … just slices™
  - “Falls back” to move construction and destruction, well-defined behavior

Lack of Standard Library APIs

Procedural precedent

- Is it sound to split a feature in language and library, and merge them separately, before a consistent design is achieved?

- Is this encouraging authors of changes affecting language *and* library changes to split their papers?
  - … I have P2509 on the table …

- Is this why P1144 failed to gain consensus?

Lack of Standard Library API

- P2786 only added a minimal library API:
  - A type trait
  - A trivial relocation function
- Further library work has been delegated to other papers:
  - P2959, “Container Relocation”
  - P2967, “Relocation Has A Library Interface”
  - P3239, “Relocating Swap”

Have we got the design right?

- Trivial Relocation is a feature that first and foremost is going to be used to optimize library facilities

- The library additions should have been thoroughly analyzed in order to validate the language changes
  - P2786 has no field experience
  - P1144’s design has widespread implementation experience: Qt, Folly, BSL, others

Leaving the status of Standard Library types as QoI

- Types in the Standard Library may or may not be TR
  - Their status is left unspecified
- Since TR has enforcement semantics, **this *completely reasonable code* is not portable:**

```
struct S trivially_relocatable {
    S();
    S(S &&);
    ~S();

    Private *data;
    std::unique_ptr<int> ptr; // ERROR if not TR
};
```

Missing: std::uninitialized_relocate algorithm

- This is what end-users need, as a useful building block

- Sure, they can implement their own
  - But so they can reimplement any std::uninitialized_* algorithm…?

- As noted before: this kind of algorithms create relationships between RO5 and trivial relocability, which in the adopted model are not clear

Missing: std::realloc

- We'd like to use `std::realloc` to reallocate an array of TR elements
  - Standard Library containers can't use realloc yet… ☹️

- In one call we allocate new memory, memcpy the elements there, deallocate the old memory
  - Hopefully, it's actually even cheaper: the allocator grows in-place

- `std::realloc` (and any other similar function) needs to have granted the same special handling that only `std::trivially_relocate` offers at the moment
  - Also an issue with P1144

Trivial relocation for assignments

Optimizing assignments

- As already noted, P2786's TR model can optimize vector reallocation
  - During reallocation, we move-construct elements in the newly allocated buffer
  - Destroy the original objects

- **It does not allow many other related optimizations**: vector erasure, insertion, swap, swap-based algorithms, etc.
  - They are based on *move assignments*, not *constructions*
  - Whether TR can be used for move assignments is a *different* type property

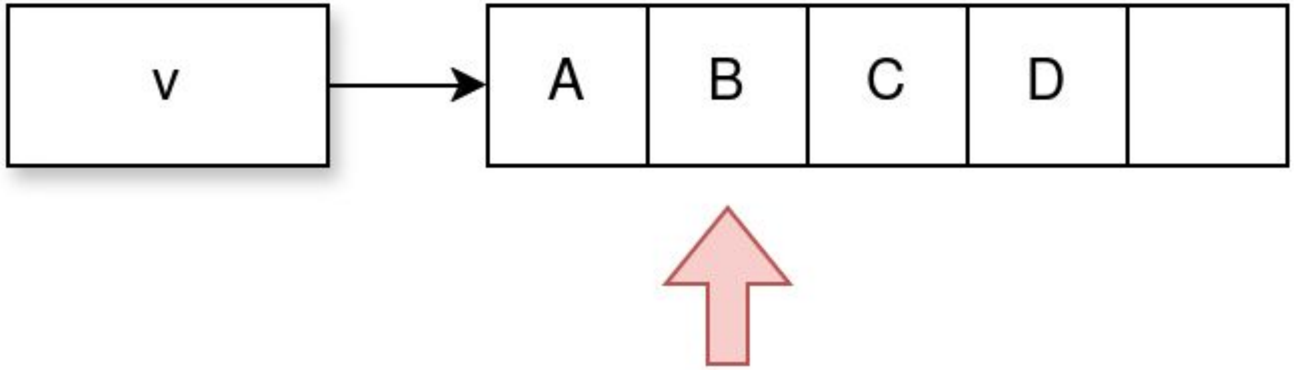
- Meant to be tackled by follow-up papers

Example: vector erasure

- Vector erasure is specified (in [vector.modifiers]) in terms to work in terms of move assignments
- For instance, to erase one element:
  - Move-assign each element after the to-be-erased one to the left
  - Destroy the last element

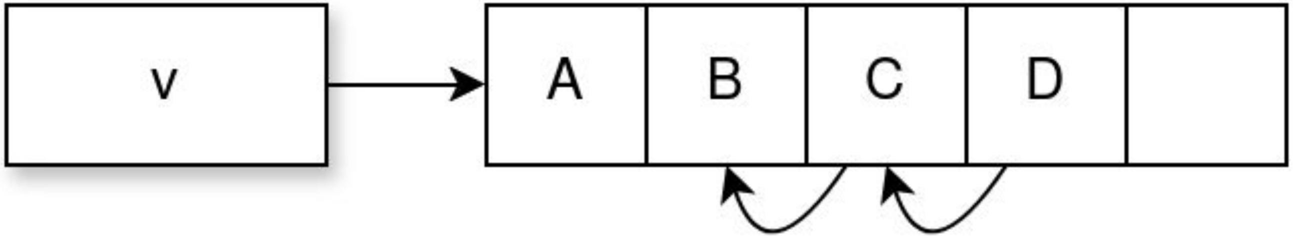
Example: vector erasure

- Move-assign each element after the to-be-erased one to the left
- Destroy the last element



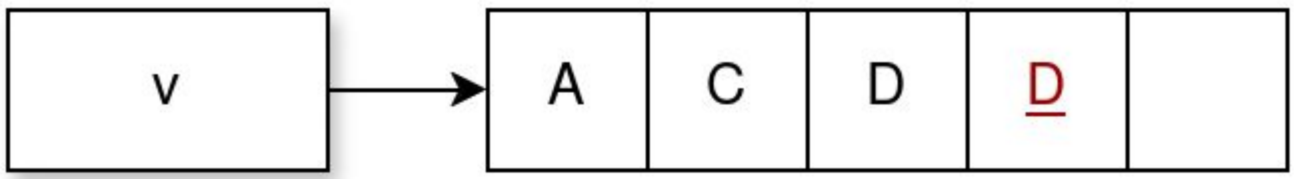
Example: vector erasure

- Move-assign each element after the to-be-erased one to the left
- Destroy the last element



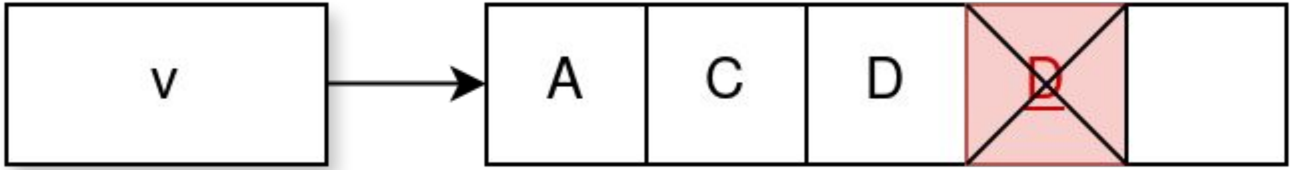
Example: vector erasure

- Move-assign each element after the to-be-erased one to the left
- Destroy the last element



Example: vector erasure

- Move-assign each element after the to-be-erased one to the left
- Destroy the last element

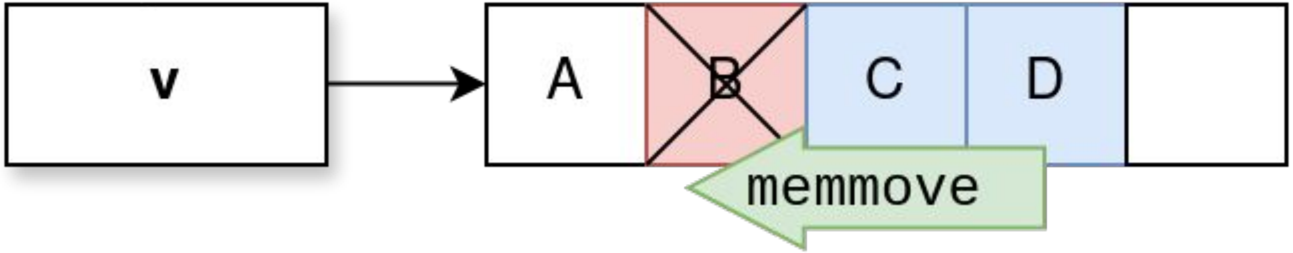


Vector erasure for TR types

- Given a vector of TR types, couldn't we use TR to erase?
- In principle, yes:
  - Destroy the element(s) to be erased
  - Compact the tail to the left by trivially relocating it

Vector erasure for TR types

- Given a vector of TR types, couldn't we use TR to erase?
- In principle, yes:
  - Destroy the element(s) to be erased
  - Compact the tail to the left by trivially relocating it



TR and move assignments

- However, in practice, **no**

- We can't "just" change semantics for vector operations, swaps, etc.
  - Changes the requirements on the operations
  - Hyrum's law: people are depending on the current semantics…

- For instance, can't change vector::erase to do something else:
  - Destroy the element to be deleted;
  - And then move construct+destroy elements from the tail.

  - Although this would unlock relocation semantics…

TR and move assignments

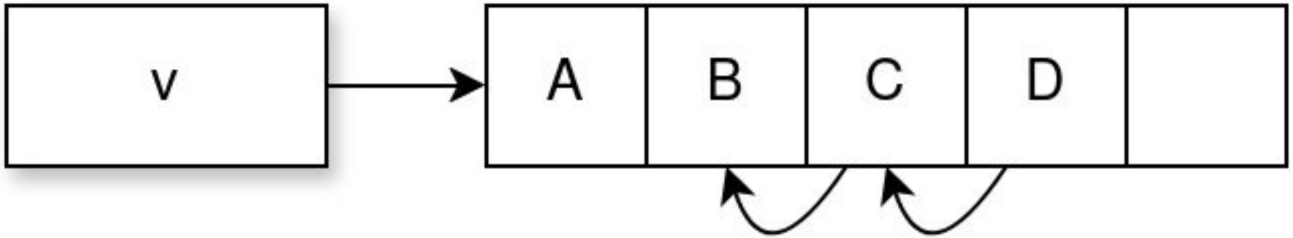
- Relocation semantics and move assignments semantics are actually tied

- We could keep the existing semantics for vector erase <u>and</u> use TR if we had an extra guarantee from a type:

  that its move assignment is "equivalent" to destruction of the target followed by move construction from the source.

- There are TR types for which this holds, and TR types for which does not.

How TR can optimize erasure (given a suitable type)



<table>
<tr><th>Erase as currently specified:</th><th>Possibly “equivalent” to:</th><th>Which is then equivalent to:</th></tr>
<tr><td rowspan="2">Move assign C onto B</td><td>Destroy B</td><td>Destroy B</td></tr>
<tr><td>Move construct C over B</td><td rowspan="2">Relocate C over B</td></tr>
<tr><td rowspan="2">Move assign D onto C</td><td>Destroy C</td></tr>
<tr><td>Move construct D over C</td><td rowspan="2">Relocate D over C</td></tr>
<tr><td>Destroy D</td><td>Destroy D</td></tr>
</table>

TR and move assignments

- Consider `std::unique_ptr<int>` and `std::tuple<int &>`
  - Both are TR types in P2786
- Very different behaviors on move assignment:
  - **`std::unique_ptr<int>`**: destroys the object owned by the target, transfers ownership from source, source is left empty.
    - *Equivalent to destroying the target unique_ptr, and move constructing from the source*
  - **`std::tuple<int &>`**: writes through the reference
    - *Not equivalent*

- Erasing an element out of a vector<unique_ptr<int>> could use TR
- Erasing an element out of a vector<tuple<int &>> cannot use TR
  - The side-effects in the referenced objects need to be visible

tuple<int &>? Really?

- OK: a better example is `std::pmr::string`
  - Polymorphic allocators don't propagate on assignment
  - OK: std::basic_string may not be TR at all (SSO, self-referential)

- In other words, assuming a TR basic_string, then

  ```
  std::vector<std::pmr::string>
  ```

  has the same "issue" of

  ```
  std::vector<std::tuple<int &>>
  ```

  - One can't use TR to optimize erasure

How to model this type property?

- This is a different type property than trivially relocatable: **how to model it?**

- In P1144's model, TR covers *both* construction and assignment
    - And therefore swaps, algorithms, etc.
        - We want them! Sorting an array of unique_ptr generates terrible code today.
    - In P1144's model, `tuple<int &>` is not TR
        - We can still optimize vector reallocation for it if `tuple<int &>` is trivially move constructible + trivially destructible
        - But yes, reallocating a vector of a std::pmr type won't get automatically optimized

- In P2786's model, TR only covers construction
    - Assignment, swaps, etc. are left to follow-up papers

How to model this type property?

- P2959 (follow-up of P2786) proposes a *library* trait/customization point
  - The idea is that it doesn't affect the core language

- RO5 types can opt-in by specializing a trait:

```
template <>
inline constexpr bool container_replace_with_assignment_v<RO5Type> = true;
```

- Otherwise: calculate the value of the trait
  - Using compiler magic, reflection, … (but it should *not* have to wait for reflection!)

```
// S should get “TR for assignments” automatically,
// (assuming unique_ptr<int> has it)
struct S {
     std::unique_ptr<int> ptr;
};
```

A separate enabler?

- Having **a separate enabler is vexing for TR RO5 types**
  - Most of them can have this optimization, but need to *remember* to opt-in into another enabler (in addition to the explicit `trivially_relocatable` mark)!
  - … RO7?

- It should have the same enforcement policy as TR for construction!
  - If a subobject doesn't qualify for TR for assignments, marking a class should make the program ill-formed!

- Therefore, it should be a language feature, not a library one

Another keyword perhaps?

- Should we have another keyword for this?

- Maybe, but IMO: `trivially_relocatable` should enable TR for construction *and* assignment
  - Precedent of *trivially copyable*, umbrella property
  - Widespread implementation precedent of these semantics

  - … find another keyword for “TR for construction only”

Summary: lack of TR for assignments

- It’s a huge optimization opportunity left on the plate (erase, insert, swap, algorithms)
  - *most* TR types have “value semantics” for move assignments and would benefit;
  - should’ve been included from the get-go

- At odds with existing practice
  - Qt, Folly, BSL: their definitions of TR types always encompass construction and assignment

- Squatting the term “trivially relocatable”
  - I’d prefer it to be akin to “trivially copyable”: an *umbrella* property for both TR for move construction and move assignment

- Should be a language feature
  - (Absolutely not have to wait for reflection)
  - Use a contextual keyword just like for TR for construction?
  - Have the same enforcement semantics as the main feature

Enforcement semantics

Enforcement semantics

- It is ill-formed to mark as TR a class that has non-TR bases or members:

```
struct S trivially_relocatable {
    S();
    ~S();

    NonTRClass m_data; // ERROR
};
```

Enforcement semantics

- On one side, this is going to prevent mistakes
- Many vocabulary types may not be TR:

```
struct S trivially_relocatable {
  S();
  ~S();

  std::string m_data; // ERROR! Likely not TR, SSO may
                      // require a pointer into self
};
```

Enforcement semantics: adoption issues

- On one other side, this may be annoying to deploy in practice
  - The bar for adoption is set extremely high given **no** existing code uses TR

```
struct S /* trivially_relocatable? */
{
    S();
    ~S();

    Lib1::Class1 m_foo;
    Lib2::Class2 m_bar;
    Lib3::Class3 m_baz;
};
```

Enforcement semantics: Standard Library

- One of the offenders will be the Standard Library itself, due to its unspecified status:

```
struct S /* trivially_relocatable? */
{
    S();
    ~S();

    std::shared_ptr<int> m_foo;
    std::vector<int> m_bar;
};
```

Enforcement semantics: unclear implementation costs

- It is very common to implement vocabulary types via extensive (private) inheritance and composition. E.g. `std::variant` from libc++:

```
template<typename... _Types>
  class variant
  : private __detail::__variant::_Variant_base<_Types...>,
    private _Enable_default_constructor<
      __detail::__variant::_Traits<_Types...>::_S_default_ctor,
        variant<_Types...>>,
    private _Enable_copy_move<
      __detail::__variant::_Traits<_Types...>::_S_copy_ctor,
      __detail::__variant::_Traits<_Types...>::_S_copy_assign,
      __detail::__variant::_Traits<_Types...>::_S_move_ctor,
      __detail::__variant::_Traits<_Types...>::_S_move_assign,
      variant<_Types...>>
  {
```

Enforcement semantics: unclear implementation costs

- All these base classes must be TR for the final type to be TR
  - A few will certain require explicit marking

- How vexing is this going to be, compared to just marking the “leaf” class?
  - Experience needed!

Enforcement semantics: UB still possible

- Even for RO0 types, UB may still be possible
  - Should `std::trivially_relocate` have preconditions?
  - Cf. the discussion on slicing / polymorphic types

- What does SG12 think about this?

Conclusions

Conclusions

- Reconsider the adoption of P2786 as the relocation model for C++
- Ideally, P1144 and P2786 should be “merged”
  - But given the status quo, P1144 provides the semantics that match existing experience
- One proposal that covers language + library

Conclusions

- Give proper name and semantics to what “relocation” means
- Give proper name and semantics to the type property “move assignment = destruction + move construction”
- Have two language enablers
  - One for trivial relocation only for destructive moving
  - One for trivial relocation “everywhere” (construction / assignments / swaps / …)
- The latter should have the simpler, more generic name!
  - Most TR types will want to use that one
  - “Trivially copyable” as precedent
- This should be part of the same package

Conclusions

- The costs of enforcement semantics are unknown
  - Ask for express vote, SG12 opinion?
- If enforcement is wanted, then the proposal **must** mark all the RO5 Standard Library types that are TR
  - TR shouldn't ship in C++26 if the library isn't also ready
  - Leaving it to QoI is a usability nightmare / poorly cooked feature

Thank you!