N3974 - Polymorphic Deleter for Unique Pointers

Peter Sommerlad

Motivation

- shared_ptr<Base> is safe for storing pointers to a derived type, because of its type-erasing deleter
    - but it incurs synchronization overhead
- unique_ptr<Base> is safe only for Base with a virtual destructor
    - and doesn’t check so
- a user-defined destructor invalidates move operations and copy operations (in the future)
    - inheriting from a Base class with virtual dtor requires explicit resurrection of move/copy operations, if needed
- That makes OO-ish interface mix-in a burden and only allows the “Rule of Zero” with shared_ptr<Base> incurring overhead

alternatives

- DIY type-erasing deleter for unique_ptr<Base,my_deleter> pb;
  - breaks with pb.reset(otherderivedptr)
    - thus needs unique_ptr specialization
- std::function<void(void*)> as deleter
  - might always have allocator overhead and thus lead to two allocations on make_uniqe_xxx();

Prop2: checked_delete

- not necessarily required to be part of std, because it doesn't require unique_ptr specialization
- convenient, zero-overhead alternative to unique_ptr's default_delete
- checks if Base has a virtual destructor
- compile-error if not when assigned from a unique_ptr<Derived>

Prop 1: safe_delete

- Idea: provide the type erasure like shared_ptr's default_delete without the need for a (potential) additional allocation, plus the safe-guard of a working reset() member function
  - can't do that outside the standard, because a specialization of unique_ptr is required for behavior
- Price: unique_ptr<T,safe_delete> is bigger than a plain unique_ptr<T>, but no additional synchronization required (neither through allocation) as with shared_ptr<T>

Advantages of safe_delete

- no synchronization overhead
- no need for base-class virtual destructor
    - no need for resurrecting copy/move operations in derived classes -> Rule of Zero continues to work
- works with OO-interface mix-ins
- works with polymorphic vector<unique_ptr<Base>> without Base having a virtual dtor

Cost of safe_delete

- Yet another library type and factory function
- Does it hit the sweet spot? IMHO, yes.
- Should we just tell users who want OO-ish code to rely on shared_ptr<Base> for it? And don't care about synchronization overhead.
- Should we tell users to live with their DIY safe_delete and tell them to sidestep up.reset(other) or get burned.

Objections/Questions

- * What's so bad about defining a virtual destructor for a class that already has virtual functions? (The virtual functions are what"polymorphic" means.)
    - see above: adding it might require resurrection of copy/moved in derived classes that you want to. (can no longer live with Rule of Zero -> teachability overhead)
- You talk about overhead a lot. Can you measure the reduced overhead of this class vs shared_ptr?
    - We didn’t measure, but shared_ptr will definitely be more expensive (but smaller)
- The need to define a whole new specialization of `unique_ptr<T,safe_delete<T>>` implies to me that this shouldn't be spelled "unique_ptr", or possibly that something else in the main template should change.
- Should `default_delete` grow the same conversion checks as`checked_delete`? What correct code would we break? IMHO none.
- Is unique_ptr<T,function<void(void*)> sufficient? -> NO, because of reset()

TBD

- Naming
- Need
- Typos/Mistakes in paper…….