D3XXX - Polymorphic Deleter for Unique Pointers

Marco Arena and Peter Sommerlad

2014-05-28

Document Number:	D3XXX
Date:	2014-05-28
Project:	Programming Language C++

1 Introduction

Something on Peter Sommerlad's *Rule of Zero*...

Something on deleted copy/move operations when a destructor is defined. Makes writing polymorphic base classes tricky, even when copying is not really useful of such. But will prohibit easy implementation of Template Method design pattern, where subclasses only need to fill in some algorithm steps with member functions and no data.

Something on shared_ptr vs. unique_ptr

2 Acknowledgements

- We need to thank Marco Arena for writing a blog article on how to enable Peter Sommerlad's *Rule of Zero* for unique_ptr. [1]
- Thanks also to members of the mailing lists who gave feedback.

3 Motivation and Scope

While `std::unique_ptr` can be tweaked by using a custom deleter type to a handler for polymorphic types, it is awkward to use as such, because such a custom deleter is missing from the standard library. API's would need to provide such a handler and different libraries will definitely have different such implementations. In addition to a standardized alias template for unique_ptr with a different deleter, a corresponding

[1] http://marcoarena.wordpress.com/2014/04/12/ponder-the-use-of-unique_ptr-to-enforce-the-rule-of-zero/

factory function for polymorphic types, remembering the created object type in the deleter is required.

This proposal introduces `unique_poly_ptr<T>` as a template alias for `uniqe_ptr<T,polymorphic_-deleter<T>>` and `make_unique_poly<T>(...)` as a factory function for it. The polymorphic_deleter is not specified as such, to enable implementors creative and more efficient implementations, i.e., storing the deleter object in the allocated memory instead of the handle object, like shared_ptr implementations can do, when allocated with make_-shared.

4 Impact on the Standard

This proposal is a pure library extension to header ¡memory¿ or its corresponding header for an upcoming library TS. It does not require any changes in the core language, and it has been implemented in standard C++ conforming to C++14. Depending on the timing of the acceptance of this proposal, it might go into the library fundamentals TS under the namespace std::experimental, a follow up library TS or directly in the working paper of the standard, once it is open again for future additions.

5 Design Decisions

5.1 Open Issues to be Discussed

- Are the names chosen appropriate. Potential alternative candidates are: unique_-object, unique_polymorphic_ptr, unique_object_ptr
- Is it useful or even desirable to have array support for unique_poly_ptr. Peter doesn't think so, but we might need to specify this limitation explicitly.

6 Technical Specifications

The following formulation is based on inclusion to the draft of the C++ standard. However, if it is decided to go into the Library Fundamentals TS, the position of the texts and the namespaces will have to be adapted accordingly, i.e., instead of namespace `std::` we suppose namespace `std::experimental::`.

6.1 Changes to [unique.ptr]

In section [unique.ptr] add the following to the `uniqe_ptr` synopsis in corresponding places.

```
namespace std{
```

```
struct polymorphic_delete;

template<typename T>
unique_poly_ptr=unique_ptr<T,polymorphic_delete>;

template<typename T, typename... Args>
unique_poly_ptr<T> make_unique_poly(Args&&... args);

}
```

In section [unique.ptr.dltr] add a subsection [unique.ptr.dltr.poly] for polymorphic_-delete.

6.2 `polymorphic_delete` [unique.ptr.dltr.poly]

1 This subclause contains infrastructure for a polymorphic deleter.

2 [*Note:* `polymorphic_delete` is meant to be a deleter for safe conversion of `unique_-ptr<Derived¿` to `unique_ptr<Base>` even when the Base class doesn't define a virtual destructor. *— end note*]

```
namespace std{

class polymorphic_delete{
   using del_t = void(*)(void*); // exposition only
   del_t del_; // exposition only

   template <typename T>
   static void delete_it(void *p) // exposition only
   {
      delete static_cast<T*>(p);
   }
public:
   template<typename T>
   polymorphic_delete(T*) noexcept
     : del_(&delete_it<T>)
   {}

   void operator()(void* ptr) const
   {
     (*del_)(ptr);
   }

};
}
```

7 Appendix: Example Implementations

7.1 TBD