N3974 - Polymorphic Deleter for Unique Pointers

Marco Arena, Davide di Gennaro and Peter Sommerlad

2014-05-28

Document Number:	N3974
Date:	2014-05-28
Project:	Programming Language C++

1 Introduction and Motivation

Special member functions, i.e., move/copy constructors and assignment operators will not/no longer be compiler provided if a destructor is defined. However, currently all text books and compiler warnings propose to define a virtual destructor when one defined a polymorphic base class with other virtual functions. Some IDEs even automatically generate class frames consisting only of a default constructor and a virtual destructor.

In C++98 the *"Rule of Three"* was the best practice to get consistent behavior from a class that either required a destructor or a copy constructor or copy assignment. Beginning with C++11 move semantics complicated the situation. Peter Sommerlad therefore promotes a *Rule of Zero* that tells "normal" classes to be written in a way that neither a destructor nor a copy or move operation needs to be user-defined. That means, classes need to be written in a way that compiler-provided defaults just work™.

However, with heap-allocated polymorphic types in C++11 code this means one needs to use `shared_ptr<Base>` and `make_shared<Derived>` to avoid the need to define a virtual destructor for `Base`. There is no standard deleter for unique_ptr that will allow to safely use `unique_ptr<Base>` if Base doesn't define a virtual destructor. Such a mis-use is not even detectable easily.

This proposal tries to ease the burden for programmers of heap allocated polymorphic classes and gives them the option to use unique_ptr with a standard provided deleter classes that either check correct provisioning of a virtual destructor in the base class or provide a slight overhead infrastructure for save deletes (1 extra function pointer).

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]

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

- Thanks for Davide di Gennaro for proposing the deleter with safeguard against missing virtual destructors in bases. Special thanks for teaching me the intricate issues of providing a polymorphic_delete that can (almost) actually work.

- Thanks also to members of the mailing lists who gave feedback.

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

For promoting the *Rule of Zero*, this proposal introduces `unique_poly_ptr<T>` as a template alias for `uniqe_ptr<T,polymorphic_delete>` and `make_unique_poly<T>(...)` as a factory function for it. The polymorphic_delete deleter is not specified in detail, 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. However, this only moves the memory overhead of 1 extra pointer from the handle object to heap memory.

For more classic code with Base classes with a virtual destructor, this proposal introduces safe_delete deleter, that is limiting a `unique_ptr<Base,safe_delete<Base>>` move of a `unique_ptr<Derived,safe_delete<Derived>>` if `Base` has a virtual destructor.

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.
- It seems impossible to make reset(pointer) work, if pointer is not equal to nullptr. This might call for a partial specialization of unique_ptr¡T,polymorphic_delete¿ which we do not (yet) specify. Other operations seem not to be as critical. A partial specialization would even need to be a template member function to allow pointer to be passed in from any subclass and replace the infrastructure in the deleter object as well, or, it might prohibit reset with a non-nullptr completely for simplicity. This type is only meant to be instantiated by its factory.

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);

template <class T>
struct  safe_polymorphic_delete;

template<typename T>
using unique_safe_ptr=std::unique_ptr<T,safe_polymorphic_delete<T>>;

template<typename T,typename ...ARGS>
unique_safe_ptr<T> make_unique_safe(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. It will incur one function pointer overhead. — *end note*]

```
namespace std{
struct polymorphic_delete{
        void *memory; // exposition only
        void (*del)(void *) noexcept; // exposition only
        polymorphic_delete();
        template<typename T>
        polymorphic_delete(T *tp);
        void operator()(void *p) noexcept;
};
}
```

```
polymorphic_delete()
```

3 *Effects:* creates an empty polymorphic_delete object, that can not delete anything.

```
template<typename T>
polymorphic_delete(T *tp);
```

4 *Effects:* initializes

- `memory` with `tp` and
- `del` with `[](void *p) noexcept {delete static_cast<T*>( p );}`

```
void operator()(void *p) noexcept;
```

5 *Effects:* if neither `p`, `memory`, or `del` are equal to `nullptr` calls `del(memory)` and resets `memory` to `nullptr`.

6 [*Note:* Calling `reset( p )` on a `unique_ptr` with a `polymorphic_delete` deleter where p is non-null incurs undefined behavior. — *end note*]

In section [unique.ptr] append a subsection [unique.ptr.poly] for the unique pointers for polymorphic types.

6.3 unique_ptr for polymorphic types [unique.ptr.poly]

1 This subclause contains infrastructure for a creating unique pointers for polymorphic types without the need to define a base class virtual destructor.

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

2 *Effects:* works like make_unique but will store a deleter function `polymorphic_delete` that deletes a `T*`.

3 *Returns:* `unique_ptr<T, polymorphic_delete>(new T(forward<Args>(args)...), static_cast<T*>(nullptr))`.

4 [*Note:* A `unique_poly_ptr<Derived>` created with make_unique_poly can be assigned safely to a `unique_poly_ptr<Base>`, even when Base doesn't have a virtual destructor. This allows for example to have an efficient container with `uniqe_poly_ptr<Base>` without the overhead of `shared_ptr<Base>`. — *end note*]

In section [unique.ptr.dltr] add a subsection [unique.ptr.dltr.safe] for safe_polymorphic_delete.

6.4 `safe_polymorphic_delete` [unique.ptr.dltr.safe]

1 This subclause contains infrastructure for a deleter for polymorphic types that ensures a base class defines a virtual destructor.

2 [*Note:* `safe_polymorphic_delete` is meant to be a deleter for safe conversion of `unique_ptr<Derived>` to `unique_ptr<Base>`. Such a conversion will not compile, if `Base` does not have a virtual destructor. — *end note*]

```
namespace std{
template <class T>
struct  safe_polymorphic_delete
{
        typedef T*pointer;
    constexpr safe_polymorphic_delete() noexcept = default;
    template <class U>
         safe_polymorphic_delete(const safe_polymorphic_delete<U>&
             ,std::enable_if_t<
                 std::is_convertible<U*, T*>{}()
                 && (std::is_same<std::remove_cv_t<U>,std::remove_cv_t<T>>{}()
                     || std::has_virtual_destructor<T>{}()
                  )>* = 0
                  ) noexcept {}
     void operator() (T* p) const noexcept;
};
}
```

```
template <class U>
safe_polymorphic_delete(const safe_polymorphic_delete<U>&) noexcept
```

3 *Effects:* This constructor is only available, when `U*` is convertible to `T*` and `T` provides a virtual destructor or `T` and `U` are the same except for any cv-qualifiers.

4 [*Note:* That constructor will be applied by unique_ptr's move-construction/assignment operations and thus prohibits such a move, when the base class doesn't provide a virtual

destructor if required. A mismatch in cv-qualifiers is handled by `is_convertible<U*,T*>`. — *end note*]

```
void operator() (T* p) const noexcept;
```

5 *Effects:* deletes `p`.

In section [unique.ptr] append a subsection [unique.ptr.safe] for the safe unique pointers for polymorphic types.

6.5 Safe `unique_ptr` for polymorphic types [unique.ptr.safe]

1 This subclause contains infrastructure for a creating unique pointers for polymorphic types that only work if a base class provides a virtual destructor.

```
template<typename T,typename ...ARGS>
unique_safe_ptr<T> make_unique_safe(ARGS&&...args);
```

2 *Effects:* works like make_unique but will keep safe_polymorphic_deleter¡T¿.

3 *Returns:* `unique_ptr<T, safe_polymorphic_delete<T>>(new T(forward<Args>(args)...))`.

4 [*Note:* A `unique_safe_ptr<Derived>` created with make_unique_safe can only be assigned to a `unique_safe_ptr<Base>` when Base has a virtual destructor. There is no run-time overhead. — *end note*]

7 Appendix: Example Implementations

Note the polymorphic_delete implementation uses a naive approach making the unique_ptr bigger, twice as big. An more sophisticated implementation can follow make_shared and keep the deleter function pointer on the heap with the allocated object.

```
struct polymorphic_delete {
        void *memory; // exposition only
        void (*del)(void *)noexcept; // exposition only
        polymorphic_delete()
        :memory{nullptr},del{[](void*){}}{}
        template<typename T>
        polymorphic_delete(T *tp)
                : memory(tp)
                , del { [](void *p) noexcept {delete static_cast<T*>(p);} } {}
        void operator()(void *p) noexcept
        {
                if (p) {
```

```
                        if (memory && del) {
                                del(memory);
                                memory = nullptr; // need to prohibit double deletes
                        } else {
                                assert(false); // if reset(pointer) is used
                        }
                }
        } // p is ignored, because it might be mutated through upcasts
private:
        void reset(){ if (memory && del){del(memory); memory=nullptr;}}
};
template<typename T>
using unique_poly_ptr=std::unique_ptr<T,polymorphic_delete>;
template <typename T, typename ...ARGS>
unique_poly_ptr<T> make_unique_poly(ARGS&&...args){
        std::unique_ptr<T> unclever{new T(std::forward<ARGS>(args)...)};
        T *memory=unclever.get();
        return unique_poly_ptr<T>{unclever.release(),memory};
}

// an implementation that keeps one pointer co-allocated with the memory itself
// sizeof(unique_ptr¡T,clever_polymorphic_delete¿ == 2* sizeof(void*)

using void_deleter=void(*)(void*);
struct clever_polymorphic_delete{ // allocates function pointer next to object
        void* control_block{nullptr};
        clever_polymorphic_delete()=default;
        template<typename T>
        clever_polymorphic_delete(std::pair<void_deleter,T> *cb,T* t)
        :control_block{cb}{
                using control_block_t=std::pair<void_deleter,T>;
                assert(*static_cast<void_deleter*>(control_block)==cb->first);
        }
        clever_polymorphic_delete(clever_polymorphic_delete &&other)
        :control_block{other.control_block}{
                other.control_block=nullptr;
        }
        clever_polymorphic_delete& operator=(clever_polymorphic_delete &&other){
                reset();
                std::swap(control_block,other.control_block);
                return *this;
        }
        void operator()(void *p){
                if (p){
                        if (control_block){
                                void_deleter del=
                                        *static_cast<void_deleter*>(control_block);
                                         // assume .first is really first!
                                del(control_block);
```

```
                                control_block=nullptr;
                        } else {
                                assert(false);// if reset(ptr) is used
                        }
                }
        }
private:
        void reset(){
                if (control_block){
                        void_deleter del=
                                *static_cast<void_deleter*>(control_block);
                                // assume .first is really first!
                        del(control_block);
                        control_block=nullptr;
                }
        }
};

template <typename T>
using unique_clever_ptr=std::unique_ptr<T,clever_polymorphic_delete>;

template <typename T,typename ...ARGS>
unique_clever_ptr<T> make_clever_unique(ARGS&&...args){
        using control_block_t=std::pair<void_deleter,T>;
        void_deleter deleter=[](void *cp)noexcept{
                control_block_t* ptr=static_cast<control_block_t*>(cp);
                delete ptr;
        };
        std::unique_ptr<control_block_t>
        up=std::make_unique<control_block_t>(std::piecewise_construct,
                        std::forward_as_tuple(deleter),
                        std::forward_as_tuple(std::forward<ARGS>(args)...));
        T* tp=&up->second;
        std::unique_ptr<T,clever_polymorphic_delete>
        res{tp,{up.release(),tp}};
        return res;
}

// a checking deleter
template <class T>
struct  safe_polymorphic_delete
{
    constexpr safe_polymorphic_delete() noexcept = default;
    template <class U>
         safe_polymorphic_delete(const safe_polymorphic_delete<U>&
             , std::enable_if_t<
                 std::is_same<U,T>{}()||
                 (std::is_convertible<U*, T*>{}()
                  &&  std::has_virtual_destructor<T>{}()
```

```
                  )>* = 0
                  ) noexcept {}
     void operator() (T* __ptr) const noexcept
        {
            static_assert(sizeof(T) > 0, "safe_delete can not delete incomplete type");
            static_assert(!std::is_void<T>::value, "safe_delete can not delete incomplete type");
            delete __ptr;
        }
};
template<typename T>
using unique_safe_ptr=std::unique_ptr<T,safe_polymorphic_delete<T>>;

template<typename T,typename ...ARGS>
unique_safe_ptr<T> make_unique_safe(ARGS&&...args){
        return unique_safe_ptr<T>{new T(std::forward<ARGS>(args)...)};
}
```