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 misuse is not even detectable easily. However, always using `shared_ptr` to get its desired deleter magic would also incur the overhead of the atomic reference counter and the overhead of full type erasure.

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 safe deletes through base type pointers, even if the base class doesn't define a virtual destructor.

A lot of discussion on the mailing list and the usefulness of additional smart pointer variations, we still want to pursue standardizing these utilities, because at least the safe

deleter version of `unique_ptr` can not be implemented by a user, because it requires a specialization of `unique_ptr` to ensure its correct usage with the fewest possible means to shoot yourself in the foot.

The two proposed solutions to the problems of using `unique_ptr<Base>` can be voted on independently.

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 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 safe_delete that can (almost) actually work.

- Thanks also to members of the mailing lists who gave feedback, encouragement and discussed it and apologies, if not everybody actually is mentioned.

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_safe_ptr<T>` as a template alias for `uniqe_ptr<T,safe_delete>` and `make_unique_safe<T>(...)` as a factory function for it. The `safe_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 two extra pointers from the handle object to heap memory.

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

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

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_safe_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::`.[2]

6.1 Changes to [unique.ptr]

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

```
namespace std{

struct safe_delete;

template <class _Tp >
class unique_ptr<_Tp,safe_delete>;

template<typename T>
unique_safe_ptr=unique_ptr<T,safe_delete>;
```

[2]This depends on how library extensions with specializations for a template class in the `std` namespace is handled.

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

template <class T>
struct  checked_delete;

template<typename T>
using unique_checked_ptr=std::unique_ptr<T,checked_delete<T>>;

template<typename T,typename ...ARGS>
unique_checked_ptr<T> make_unique_checked(ARGS&&...args);

}
```

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

6.2 `safe_delete` [unique.ptr.dltr.safe]

1 This subclause contains infrastructure for a safe deleter for types where unique_ptr stores a potentially compatible casted pointer.

2 [*Note:* `safe_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 is meant to provide only little overhead. It should work like `shared_ptr` without the overhead introduced by reference counting. — *end note*]

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

`safe_delete()`

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

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

4 *Effects:* initializes

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

```
template<typename T>
void operator()(T *p) noexcept;
```

5 *Effects:* if neither `p` or `del` are equal to `nullptr` calls `del(memory)`. [*Note:* It does not call `del(p)`. — *end note*]

6

In section [unique.ptr] append a subsection [unique.ptr.safe] for the safe unique pointer.

6.3 Polymorphic `unique_ptr` with safe deleter [unique.ptr.safe]

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

[*Note:* This even allows a `unique_safe_ptr<void>` initialized or reset with any non-array new expression. — *end note*]

```
namespace std{
template <class _Tp >
class unique_ptr<_Tp,safe_delete>
{
public:
    typedef _Tp element_type;
    typedef safe_delete deleter_type;
    typedef _Tp* pointer;
        constexpr unique_ptr() noexcept;
        template<typename U>
        explicit unique_ptr(U* p) noexcept;
        unique_ptr(unique_ptr&& u) noexcept;
        constexpr unique_ptr(nullptr_t) noexcept
        : unique_ptr() { }
    template <class U, class E>
        unique_ptr(unique_ptr<U, E>&& u) noexcept;
    template <class U>
        unique_ptr(auto_ptr<U>&& u) noexcept;
        ~unique_ptr();
        unique_ptr& operator=(unique_ptr&& u) noexcept;
        template <class U, class E>
        unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
        unique_ptr& operator=(nullptr_t) noexcept;
        add_lvalue_reference_t<T> operator*() const;
        pointer operator->() const noexcept;
        pointer get() const noexcept;
        deleter_type& get_deleter() noexcept;
        const deleter_type& get_deleter() const noexcept;
        explicit operator bool() const noexcept;
        pointer release() noexcept;
```

```
    template <typename U>
    void reset(U *p) noexcept;
    void reset(nullptr_t) noexcept;
    void reset() noexcept {
                this->reset(nullptr);
    }
        void swap(unique_ptr& u) noexcept;
};

template<typename T>
unique_safe_ptr=unique_ptr<T,safe_delete>;

template<typename T, typename... Args>
unique_safe_ptr<T> make_unique_safe(Args&&... args){
        return unique_safe_ptr<T>{new T(forward<Args>(args)...)};
}

}
```

If not mentioned explicitly the semantics of `unique_safe_ptr<T>` functions are identical to the corresponding `std::unique_ptr<T>` functions.

```
template<typename U>
explicit unique_ptr(U* p) noexcept;
```

2 *Requires:* `is_convertible<U*,pointer>`

3 *Effects:* constructs a `unique_ptr<T*,safe_delete>` with a deleter function `safe_delete` that deletes a `U*`.

```
template<typename U>
void reset(U *p) noexcept;
```

4 *Requires:* `is_convertible<U*,pointer>` and the expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions.

5 *Effects:* assigns `p` to the stored pointer, and then if the old value of the stored pointer, `old_p` was not equal to nullptr, calls `get_deleter()(old_p)`. Replaces the `safe_delete` deleter by one that deletes a `U*`.

```
void reset(nullptr_t) noexcept;
```

6 *Requires:* The expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions.

7 *Effects:* assigns `nullptr` to the stored pointer, and then if the old value of the stored pointer, `old_p` was not equal to `nullptr`, calls `get_deleter()(old_p)`.

In section [unique.ptr.dltr] add a subsection [unique.ptr.dltr.checked] for checked_delete.

6.4 `checked_delete` [unique.ptr.dltr.checked]

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

2 [*Note:* `checked_delete` is meant to be a deleter for safe conversion of `unique_ptr<Derived,checked_delete<Derived>>` to `unique_ptr<Base,checked_delete<Base>>`. In contrast to a `unique_ptr<Base>` such a conversion will not compile, if `Base` does not have a virtual destructor, otherwise the behavior of `checked_delete` is the same as `default_delete`. — *end note*]

```
namespace std{
template <class T>
struct  checked_delete
{
        typedef T*pointer;
    constexpr checked_delete() noexcept = default;
    template <class U>
         checked_delete(const checked_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>
checked_delete(const checked_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_checked_ptr<T> make_unique_checked(ARGS&&...args);
```

2 *Effects:* works like make_unique but will use `checked_deleter<T>`.

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

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

7 Appendix: Example Implementations

The following implementation is derived from libc++ and uses its macros partially in the unique_ptr specialization. Test cases can be provided by the one of the authors (Peter).

```
namespace std{
struct safe_delete {

        void *memory; // exposition only
        void (*del)(void *)noexcept; // exposition only
        safe_delete()
        :memory{nullptr},del{[](void*){}}{}

        template <typename T>
        safe_delete(T *tp)
            : memory((void*)tp) // ugly, seems to need c-style cast when used with volatile:-
(
        , del { [](void *p) noexcept {delete static_cast<T*>(p);} } {}

    template <typename T>
    void operator()(T *p) const noexcept
    // must be template to avoid nasty compile errors from cv qualified types
    {
        if (p) {
            if (this->del && this->memory) {
                this->del(this->memory);
            } else {
                assert(false);  // debug support for me
            }
        }
```

```
    } // p is ignored, because it might be mutated through upcasts
};
template <class _Tp >
class _LIBCPP_TYPE_VIS_ONLY unique_ptr<_Tp,safe_delete>
{
public:
    typedef _Tp element_type;
    typedef safe_delete deleter_type;
    typedef _Tp* pointer;
private:
    __compressed_pair<pointer, deleter_type> __ptr_;

    struct __nat {int __for_bool_;};

    typedef       typename remove_reference<deleter_type>::type& _Dp_reference;
    typedef const typename remove_reference<deleter_type>::type& _Dp_const_reference;
public:
     _LIBCPP_CONSTEXPR unique_ptr() noexcept
    : __ptr_ {pointer(),safe_delete{pointer()}}
        {
            static_assert(!is_pointer<deleter_type>::value,
                "unique_ptr constructed with null function pointer deleter");
        }
     _LIBCPP_CONSTEXPR unique_ptr(nullptr_t) noexcept
    : __ptr_ {pointer(),safe_delete{pointer()}}
        {
            static_assert(!is_pointer<deleter_type>::value,
                "unique_ptr constructed with null function pointer deleter");
        }
    // cross-init with adjusted deleter object
    template<typename U>
     explicit unique_ptr(U * __p,
            enable_if_t<is_convertible<U*,pointer>{}()
            ,__nat> =__nat()) noexcept
            : __ptr_ {__p
            , safe_delete {__p}}
        {
        }

    unique_ptr(pointer __p,
    typename conditional<is_reference<deleter_type>::value,
                         deleter_type,
                         typename add_lvalue_reference<const deleter_type>::type>::type __d)
             noexcept
    : __ptr_ {__p, __d} {}

     unique_ptr(pointer __p, typename remove_reference<deleter_type>::type&& __d)
             noexcept
    : __ptr_ {__p, std::__1::move(__d)}
```

```
    {
        static_assert(!is_reference<deleter_type>::value, "rvalue deleter bound to refer
    }
 unique_ptr(unique_ptr&& __u) noexcept
: __ptr_ {__u.release(), std::__1::forward<deleter_type>(__u.get_deleter())} {}

template <class _Up, class _Ep>
unique_ptr(unique_ptr<_Up, _Ep>&& __u,
               typename enable_if<
                 !is_array<_Up>::value &&
                  is_convertible<typename unique_ptr<_Up, _Ep>::pointer, pointer>::value
                  is_convertible<_Ep, deleter_type>::value &&
                  (
                   !is_reference<deleter_type>::value ||
                   is_same<deleter_type, _Ep>::value
                  ),
                  __nat
                  >::type = __nat()) noexcept
: __ptr_ {__u.release(), std::__1::forward<_Ep>(__u.get_deleter())} {}

template <class _Up>
     unique_ptr(auto_ptr<_Up>&& __p,
            typename enable_if<
                        is_convertible<_Up*, _Tp*>::value &&
                        is_same<deleter_type, default_delete<_Tp> >::value,
                       __nat
                     >::type = __nat()) noexcept
: __ptr_ {__p.get(),safe_delete{__p.get()}}
        {__p.release();
        }

     unique_ptr& operator=(unique_ptr&& __u) noexcept
        {
            reset(__u.release());
            __ptr_.second() = _VSTD::forward<deleter_type>(__u.get_deleter());
            return *this;
        }

    template <class _Up, class _Ep>

        typename enable_if
        <
            !is_array<_Up>::value &&
            is_convertible<typename unique_ptr<_Up, _Ep>::pointer, pointer>::value &&
            is_assignable<deleter_type&, _Ep&&>::value,
            unique_ptr&
        >::type
        operator=(unique_ptr<_Up, _Ep>&& __u) noexcept
        {
```

```
            reset(__u.release());
            __ptr_.second() = _VSTD::forward<_Ep>(__u.get_deleter());
            return *this;
        }
 ~unique_ptr() {reset();}

 unique_ptr& operator=(nullptr_t) noexcept
{
    reset();
    return *this;
}

 typename add_lvalue_reference<_Tp>::type operator*() const
    {return *__ptr_.first();}
 pointer operator->() const noexcept {return __ptr_.first();}
 pointer get() const noexcept {return __ptr_.first();}
       _Dp_reference get_deleter() noexcept
    {return __ptr_.second();}
 _Dp_const_reference get_deleter() const noexcept
    {return __ptr_.second();}

    _LIBCPP_EXPLICIT operator bool() const noexcept
    {return __ptr_.first() != nullptr;}

 pointer release() noexcept
{
    pointer __t = __ptr_.first();
    __ptr_.first() = pointer();
    return __t;
}
// provide templatized reset overload to adjust deleter as well
template <typename U>
 void reset(U *__p) noexcept
{
    pointer __tmp = __ptr_.first();
    __ptr_.first() = __p;
    if (__tmp)
        __ptr_.second()(__tmp);
    __ptr_.second() = safe_delete{__p};
}
// and special case for nullptr_t
 void reset(nullptr_t) noexcept
{
    pointer __tmp = __ptr_.first();
    __ptr_.first() = nullptr;
    if (__tmp)
        __ptr_.second()(__tmp);
}
// and default argument version of original unique_ptr::reset
```

```
     void reset() noexcept
    {
            this->reset(nullptr);
    }

     void swap(unique_ptr& __u) noexcept
        {__ptr_.swap(__u.__ptr_);}
};
_LIBCPP_END_NAMESPACE_STD
```

Here comes a checked_delete implementation.

```
// a checking deleter
template <class T>
struct  checked_delete
{
    constexpr checked_delete() noexcept = default;
    template <class U>
         checked_delete(const checked_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, "checked_delete can not delete incomplete type");
            static_assert(!std::is_void<T>::value, "checked_delete can not delete incomplete
            delete __ptr;
        }
};
template<typename T>
using unique_checked_ptr=std::unique_ptr<T,checked_delete<T>>;

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