Document Number: N4042
Date: 2014-05-23
Project: Programming Language C++, Library Working Group
Reply-to: Geoffrey Romer <gromer@google.com>

# Safe conversions in unique_ptr<T[]>

## Introduction

This paper proposes to resolve LWG 2118 by permitting conversions to unique_ptr<T[]> if they are known to be safe.

## Motivation and Scope

The array specialization of unique_ptr imposes several related restrictions that do not apply to the primary template:

- unique_ptr<T[], D> cannot be constructed from a plain pointer whose type is not exactly unique_ptr<T[], D>::pointer or nullptr_t.
- unique_ptr<T[], D> cannot be constructed from a unique_ptr<U[], E>&& unless U is exactly T and E is exactly D.
- unique_ptr<T[], D> cannot be move-assigned from a unique_ptr<U[], E>&& unless U is exactly T and E is exactly D.
- unique_ptr<T[], D>::reset cannot take an argument whose type is not exactly unique_ptr<T[], D>::pointer or nullptr_t.
- default_delete<T[]> cannot be constructed from a default_delete<U[]> unless U is exactly T.
- default_delete<T[]>::operator() cannot be called on a pointer whose type is not exactly T*.

The intent of these restrictions is to prevent conversions from pointer-to-derived to pointer-to-base, because it is unsafe to use the result of such a conversion in pointer arithmetic, or pass it to delete[]. However, LWG 2118 observes that this also has the effect of forbidding safe, useful conversions like qualification-conversion and user-defined conversions:

```
unique_ptr<Foo const[]> ptr1(new Foo[10]);  // ill-formed
unique_ptr<Foo[]> ptr2(new Foo[10]);
unique_ptr<Foo const[]> ptr3 = move(ptr2);  // ill-formed
unique_ptr<Foo const[]> ptr4;
ptr4.reset(new Foo[10]);  // ill-formed
```

I propose to make such demonstrably-safe constructs legal C++, while continuing to forbid conversions that are not known to be safe.

There are some cases in which these operations are demonstrably unsafe even in the primary template (e.g. conversion from unique_ptr<Derived> to unique_ptr<Base> when Base does not have a virtual destructor), and so perhaps ought to be disabled there as well as in the array specialization. This paper focuses solely on LWG 2118 and on broadening the API of the array specialization, so those problems are out of scope.

The wording changes in this paper highlight the difficulty of keeping [unique.ptr.single] consistent with [unique.ptr.runtime] while avoiding unnecessary duplication. I believe [unique.ptr] would be substantially improved by combining the two into a single section, with differences between them specified at the level of individual members (N3920's wording for `shared_ptr` shows the feasibility and desirability of this approach), and at the Issaquah meeting, there was tentative interest in making such a change as part of resolving LWG 2118. However, this paper does not propose that, in order to focus on the normative content of the change. Once we have consensus on that normative content, the restructuring of [unique.ptr] can be handled editorially, and/or with a followup paper.

## Design Decisions

### Mechanics

Implementations generally have three options for disabling a constructor or function for certain argument types:

1. Make it a template (if it isn't already), and render it ill-formed for those argument type (e.g. with a `static_assert` in the implementation)
2. Make it a template (if it isn't already), and use SFINAE to remove it from the overload set for those argument types.
3. Add a deleted template overload that can match the same arguments, and use SFINAE to remove it from the overload set for the argument types we wish to permit.

#1 has has the advantage that it can produce better error messages: a `static_assert` can be explicit and specific about why the call is disabled, whereas #2 will generally produce generic errors about no function matching the given call, and #3 will produce generic errors about ambiguous calls, or calling deleted functions. However, it causes `is_constructible`, `is_assignable`, and related traits to give incorrect answers and, more generally, prevents template metaprogramming constructs from inspecting the validity of expressions containing the operation. It can also create ambiguity between disabled and non-disabled overloads, when both match a given call.

#3 is largely equivalent to #2, but enables the operation to remain usable without an explicit template argument, in cases where the argument cannot be deduced. However, it doesn't apply to members that already have to be expressed as member templates.

The consensus of discussions in Issaquah was to use SFINAE for constructors and assignment operators, because the ability to condition on their availability is particularly important. There was no clear consensus on the other operations, `unique_ptr::reset()` and `default_delete::operator()()`. I propose to use approach #2 in all cases, for consistency and simplicity. The additional uses enabled by approach #3 (e.g. initializing a `unique_ptr<Foo[]>` from a proxy object) appear to be too marginal and speculative to justify the additional complexity.

## Multi-level qualification conversion

It seems clear that `unique_ptr` should allow usages like the above examples, which involve only top-level qualification conversions. However, the situation is more complex when there are multiple layers of qualifiers. Consider the following code, which is ill-formed under the current standard:

```
unique_ptr<Foo const * const []> ptr1(new Foo*[10]);
```

It is tempting to argue that conversions of this kind are obviously safe, and in practice this is probably true. However, the most closely analogous conversion for a plain pointer to an array, from `Foo*(*)[]` to `Foo const * const (*)[]`, is not a valid qualification-conversion (see CWG 330). Furthermore, since the resolution of CWG 1504, the behavior of accessing a `Foo*[]` through a `Foo const * const *` pointer has been technically undefined, which would make the behavior of the above code undefined even if it were well-formed. This seems to be a wording defect (see CWG 1865), and it's doubtful that implementations will give that code any meaning other than the obviously correct one, but it seems unwise for the library to venture too far out in front of the language in this respect.

Therefore, I propose to follow N3920 in appropriating the conversion rules for pointer-to-array types: assuming `unique_ptr<T[], D>::pointer` is a pointer type, then the following operations should be disabled if `U(*)[]` is not convertible to `T(*)[]`:

- `unique_ptr<T[], D>` construction from a `U*` argument
- `unique_ptr<T[], D>` construction from a `unique_ptr<U[], E>` argument
- `unique_ptr<T[], D>::reset` with a `U*` argument
- `default_delete<T[]>` construction with a `default_delete<U[]>` argument
- `default_delete<T[]>::operator()` with a `U*` argument

This effectively delegates the decision to the core language; qualification conversions below top-level would be forbidden under the current language rules, but would become legal when and if CWG 330 is resolved.

## 'Fancy' pointers

If `unique_ptr<T[], D>::pointer` is a class type (hereinafter referred to as a 'fancy' pointer), it is more difficult to impose conditions like those above, because we can no longer determine the input pointer's element type `U` as a byproduct of template parameter deduction; we must extract it using traits. However, the standard does not currently require `std::pointer_traits::element_type` or any comparable trait to be well-formed for fancy pointers, and still less for types that are implicitly convertible to fancy pointer types, so we would have to either begin requiring e.g. an `element_type` typedef (potentially invalidating some existing code) or have fallback logic for cases where the element type cannot be determined.

On the other hand, in the fancy pointer case it's less clear that these protections are needed in the first place. In principle, fancy pointers can (and arguably should) be designed to disable unsafe conversions themselves. In most cases, the deleter/fancy pointer pair already have to be designed specifically for arrays, in order to support indexing and cleanup correctly, so the necessary information is already available; the fancy pointer just has to take advantage of it. Of course, that's no guarantee that it *will* do so, so we face a tradeoff between empowering the fancy pointer and protecting its users.

Existing practice could give some indication of how to weight these priorities, but existing practice is hard to come by. Boost.Interprocess's `managed_unique_ptr`, arguably the canonical use case for fancy pointers, does not appear to support arrays, and I am not aware of any other uses of fancy pointers with arrays that would be suitable reference material. By the same token, though, the existing restrictions are unlikely to do much harm in the fancy pointer case, at least at present. I therefore propose the conservative option of retaining the status quo (all conversions disabled) for fancy pointers.

## Proposed Wording

Changes are relative to N3936.

### Revise [unique.ptr.dltr.dflt1] as follows:

```
namespace std {
  template <class T> struct default_delete<T[]> {
    constexpr default_delete() noexcept = default;
    template <class U> default_delete(const default_delete<U[]>&) noexcept;
    ~~void operator()(T*) const;~~
    template <class U> void operator()(U* ptr) const~~ = delete~~;
  };
}
```

`template <class U> default_delete(const default_delete<U[]>& other) noexcept;`
*Effects:* constructs a `default_delete` object from another `default_delete<U[]>` object.
*Remarks:* This constructor shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

~~`void operator()(T* ptr) const;`~~
`template <class U> void operator()(U* ptr) const;`
*Effects:* calls `delete[]` on ptr.
*Remarks:* ~~If T is an incomplete type, the program is ill-formed.~~ U shall be a complete type. This function shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

## Revise [unique.ptr.single]/3 as follows:

If the type `remove_reference<D>::type::pointer` exists, then `unique_ptr<T, D>::pointer` shall be a synonym for `remove_reference<D>::type::pointer`. Otherwise `unique_ptr<T, D>::pointer` shall be a synonym for ~~`T*`~~ `element_type*`. The type `unique_ptr<T, D>::pointer` shall satisfy the requirements of `NullablePointer` (17.6.3.3).

## Revise [unique.ptr.runtime] as follows:

```
namespace std {
  template <class T, class D> class unique_ptr<T[], D> {
  public:
    typedef see below pointer;
    typedef T element_type;
    typedef D deleter_type;

    // 20.7.1.3.1, constructors
    constexpr unique_ptr() noexcept;
    template <class U> explicit unique_ptr(~~pointer~~U p) noexcept;
    template <class U> unique_ptr(~~pointer~~U p, see below d) noexcept;
    template <class U> unique_ptr(~~pointer~~U p, see below d) 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;

    // destructor
    ~unique_ptr();

    // assignment
    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;

    // 20.7.1.3.2, observers
    T& operator[](size_t i) const;
    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;
    // 20.7.1.3.3 modifiers
    pointer release() noexcept;
    ~~void reset(pointer p = pointer()) noexcept;~~
    void reset(nullptr_t = nullptr) noexcept;
    template <class U> void reset(U p)~~ = delete~~;
    void swap(unique_ptr& u) noexcept;

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

A specialization for array types is provided with a slightly altered interface.
— Conversions between different types of `unique_ptr<T[], D>` that would be disallowed for the corresponding pointer-to-array types~~ or~~, and conversions to or from the non-array forms of `unique_ptr`, produce an ill-formed program.
— Pointers to types derived from `T` are rejected by the constructors, and by reset.
— The observers `operator*` and `operator->` are not provided.
— The indexing observer `operator[]` is provided.
— The default deleter will call `delete[]`.
Descriptions are provided below only for ~~member functions that have behavior different~~ members that differ from the primary template.
The template argument `T` shall be a complete type.

**unique_ptr constructors [unique.ptr.runtime.ctor]**

```
template <class U> explicit unique_ptr(~~pointer~~U p) noexcept;
template <class U> unique_ptr(~~pointer~~U p, see below d) noexcept;
template <class U> unique_ptr(~~pointer~~U p, see below d) noexcept;
```

These constructors behave the same as the constructors that take a pointer parameter in the primary template except that they ~~do not accept pointer types which are convertible to `pointer`~~shall not participate in overload resolution unless either
— `U` is the same type as `pointer`, or
— `pointer` is the same type as `element_type*`, `U` is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.~~ [ Note: One implementation technique is to create private templated overloads of these members. — end note ]~~

```
template <class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;
```

This constructor behaves the same as in the primary template, except that it shall not participate in overload resolution unless all of the following conditions hold:
— `U` is an array type `V[]`, and
— `pointer` is the same type as `element_type*`, and

— unique_ptr<U, E>::pointer is the same type as V*, and
— V(*)[] is convertible to element_type(*)[], and
— either D is a reference type and E is the same type as D, or D is not a reference type and E is implicitly convertible to D.

**unique_ptr assignment [unique.ptr.runtime.asgn]**

```
template <class U, class E>
  unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

This operator behaves the same as in the primary template, except that it shall not participate in overload resolution unless all of the following conditions hold:
— U is an array type V[], and
— pointer is the same type as element_type*, and
— unique_ptr<U, E>::pointer is the same type as V*, and
— V(*)[] is convertible to element_type(*)[], and
— either D is a reference type and E is the same type as D, or D is not a reference type and E is implicitly convertible to D.

**unique_ptr observers [unique.ptr.runtime.observers]**

```
T& operator[](size_t i) const;
```

*Requires:* i < the number of elements in the array to which the stored pointer points.
*Returns:* get()[i].

**unique_ptr modifiers [unique.ptr.runtime.modifiers]**

~~void reset(pointer p = pointer()) noexcept;~~
void reset(nullptr_t p = nullptr) noexcept;
Effects: ~~If get() == nullptr there are no effects. Otherwise get_deleter()(get()).~~Equivalent to reset(pointer()).
~~Postcondition: get() == p.~~

```
template <class U> void reset(U p);
```

This function behaves the same as the reset member of the primary template, except that it shall not participate in overload resolution unless either
— U is the same type as pointer, or
— pointer is the same type as element_type*, U is a pointer type V*, and V(*)[] is convertible to element_type(*)[].