Document Number: N4089
Date: 2014-06-25
Project: Programming Language C++, Library Working Group
Revises: N4042
Reply-to: Geoffrey Romer <gromer@google.com>

Safe conversions in `unique_ptr<T[]>`, revision 2

Introduction

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

Changes since N4042

This paper corrects a minor factual error in N4042, and makes the following changes to the proposed wording:

- Restored guarantee that `default_delete::operator()` is ill-formed when called on an incomplete type.
- Added noexcept to `reset()`.
- Improved parallelism of wording for SFINAE conditions.
- Added note clarifying relationship between SFINAE rules for the primary template and the specialization.

Correction on multi-level qualification conversions

Consider the following code:

```
unique_ptr<Foo const * const []> ptr1(new Foo*[10]);
Foo const * ptr = ptr1[9];
```

Under this proposed resolution, the declaration of `ptr1` would be ill-formed due to the issue reported in CWG 330, but would become well-formed if that issue is resolved. N4042 stated that even if it were well-formed, the declaration of `ptr1` would have undefined behavior due to the issue reported in CWG 1865, but this is not precisely correct. The undefined behavior results not from the pointer conversion itself, but from performing arithmetic on the result of the conversion, so it is the second line that results in undefined behavior, not the first.

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~~U is an incomplete type, the program is ill-formed. 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(pointerU p) noexcept;
    template <class U> unique_ptr(pointerU p, see below d) noexcept;
    template <class U> unique_ptr(pointerU 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) noexcept = 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, where *UP* is `unique_ptr<U, E>`:
— `U` is an array type, and
— `pointer` is the same type as `element_type*`, and
— *UP*`::pointer` is the same type as *UP*`::element_type*`, and
— *UP*`::element_type(*)[]` 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`.
[*Note:* this replaces the overload-resolution specification of the primary template — *end note*]

`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, where *UP* is `unique_ptr<U, E>`:
— `U` is an array type, and
— `pointer` is the same type as `element_type*`, and
— *UP*`::pointer` is the same type as *UP*`::element_type*`, and
— *UP*`::element_type(*)[]` 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`.

[*Note:* this replaces the overload-resolution specification of the primary template — *end note*]

`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) noexcept;`
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(*)[]`.