Constraining unique_ptr

Document: N2853=09-0043
Date: 2009-03-20
Reply to: Alisdair Meredith public@alisdairm.net

Introduction

Unique_ptr is a new class template introduced in C++0x to simplify the use of move semantics and pointer-ownership. It is an extremely flexible template, supporting an extremely high degree of customizability with essentially zero overhead. By default there is no overhead in time or space compared to a 'raw' pointer.

It is clearly desirable to be able to use this new smart pointer in constrained code, and a challenge to constrain it in such a way that it loses none of the flexibility without adding any overhead.

This paper forms a partial response to the following National Body comments on the CD1 ballot for C++0x:

CA-2, US-61, US-62, UK-209, UK-211

It indirectly contributes to resolving some additional comments:

CA-1, CH-2, US-2

Note: This paper is not complete, and clearly marks out as yet unreviewed clauses. It is intended to generate feedback so that a complete paper will be available for the pre-Frankfurt mailing.

Notable Use Cases

The following use cases highlight some tricky design choices supported by the original unconstrained template that must be preserved for the constrained template.

- Support incomplete types (e.g. for use with pImpl idiom)
- Support custom pointer types (e.g. shared memory)
- No space overhead compared to raw pointer by default

Design Issues

Empty pointers and nullptr equivalence

Another problem that comes up is the nature of a 'null' unique_ptr. Essentially, we would like the following expressions to hold, given a variable `ptr` of type `unique_ptr<T,D>`.

```
static_cast<bool>(ptr) ⇔ ptr == unique_ptr<T,D>{} ⇔ ptr == nullptr
```

Likewise, we would like to see

```
unique_ptr<T,D>{} ⇔ unique_ptr<T,D>{ nullptr }
```

This seems like a good place to introduce axioms, but finding the best minimum formulation of the concept to hold these axioms is probably the biggest design challenge.

Re-usability of concepts

As we design concepts and constraints for unique_ptr, we must keep one eye to the future and the next task, which will be constraining shared_ptr. The plan is that the concepts for unique_ptr should be usable without changes for shared_ptr, otherwise they have not delivered on a true abstraction, but merely an implementation detail.

The success of this effort will not be known until that follow-up paper.

Likewise, at the heart of any smart pointer type is the notion of a dereferenceable thing. This might imply some kind of consistency between operator* and operator->, callability if the pointed-to object is callable, and handling the case that the pointed-to type is a void type. This set of properties is common to all smart-pointers, and that might include iterators which are essentially a smart-pointer that abstracts iteration rather than object lifetime.

Should we support function pointers

Technically there is nothing in `unique_ptr` that prevents use with function pointers. The user will be responsible for providing an appropriate deleter policy as `DefaultDelete` will not do the right thing, but otherwise everything should work (at least with the unconstrained unique_ptr).

The more interesting question is that if we choose to support function pointers, should we provide a function call operator for pointers to `Callable` types? This would be a clear extension of the existing `unique_ptr` interface, and start encroaching on the design space of `std::function`. However, it might still be useful in specific contexts. E.g. a function pointer with a deleter that includes a spin-lock on the shared library the function pointer was retrieved from.

This paper does not extend in this direction, although the question itself seems worth of some discussion when in LWG review. The end result is that unique_ptr is constrained to point to object types, which is a mild loss of functionality compared to the unconstrained template.

Solutions

New Concepts

A couple of new concepts are proposed to support smart pointers with deleters. The same set of concepts should be useful constraining shared_ptr as well. It is recommended that any additional requirements emerging from the process of constraining shared_ptr be tackled as an issue for synchronizing the two smart pointer types, than maintain a parallel set of smart pointer custom deleter concepts.

NullablePointerLike

For the purposes of this concept, a pointer is something that is copyable, movable and contextually convertible to bool. Likewise, it is convertible to/from nullptr literals, and a value that compares equal to the nullptr literal is equivalent to one that yields false when converted to bool.

While many pointers are dereferencable, that is not a universal trait (e.g. void*) and not a requirement of this concept.

The key ideas it is factoring out are

- interaction with nullptr
- boolean conversions
- relationship between bool and nullptr

The value-like semantics are added as a convenient bundling that simplify the writing of unique_ptr constraints, but might be pulled out separately if more fine-grained concepts are desired.

Note that the proposed specification does **not** require that a `NullablePointerLike` type is `DefaultConstructible`, although it does require the ability to construct from null pointer constants. This has the effect of replacing occurrences `pointer()`, typically used as a default, with a `nullptr` literal.

Likewise, while `EqualityComparable` would be a useful trait, it is not absolutely required and the whole `unique_ptr` template can be specified without it. However, there are additional axioms that make should be asserted if this extra constraint holds.

One design considered making `NullablePointerLike` a refinement of `SemiRegular` or `Regular`, which would have brought along default construction and comparison operators. However, this seems to be over-constraining, so was ultimately rejected. In particular, it would disallow use of move only types as models of `NullablePointerLike`, including `unique_ptr` itself.

SmartPointerDeleter

The SmartPointerDeleter concept describes types that may be used as a custom deleter with shared_ptr and to supply the deleter policy to unique_ptr. The requirements for unique_ptr are that it provides an associated pointer type, which defaults to T*, and that it is Callable with a single pointer argument. It is expected that calling the SmartPointerDeleter type with a pointer value will destroy the pointed-to object and release other resources associated with that ownership, although it is not clear how to write the semantic requirements in terms of a set of axioms.

SmartPointerDeleters are NOT required to handle null pointer values. However, this is currently a requirement for custom deleters used with shared_ptr.

PointerType and MemberPointerType

The concepts PointerType and MemberPointerType are required to constrain the acceptable values for constructors that do not supply a value for the deleter. If the deleter type is a function pointer or member-pointer type, this would produce a null pointer which is always an error, yet easily diagnosed by the type system. While these concepts are used in this paper, they are not defined. This was initially an oversight assuming they existed in the support concepts clause, 14.9.4 [concept.support]. However, it now relates directly to National Body comment US-70/LWG #1018.

One easy way to write these concepts would to make them depend on the type traits:

```
concept PointerType< typename T > {
   requires True< is_pointer<T>::value >;
};
```

To make this work, we would need to constrain the type traits templates, although they can all safely be constrained with an arbitrary requires True<true> constraint so there is no real difficulty here.

Alternatively, we can introduce new fundamental concepts PointerType etc. and define the type traits in terms of these new concepts:

```
template< typename T >
   requires True<true>
   struct is_pointer : false_type {};

template< typename T >
   requires PointerType<T>
   struct is_pointer<T> : true_type {};
```

This variation assumes that any constraint on T makes the partial specialization more-specialized than the primary, which needs to be confirmed with Core.

Resolving this should be part of a larger paper dedicated to the topic, which this paper will depend on.

Rationales

Some decisions taken below may not seem obvious at first sight. Some of the more interesting decisions are spelled out below.

Constraints on return type of operator->

`operator->` returns a value of type `pointer`. We could accept the implicitly deduced `Returnable<pointer>` constraint and leave it at that. However, this paper proposes strengthening that constraint to `CopyConstructible<pointer>` because `Returnable` would also support move-only pointer types, and we don't want `operator->` to implicitly move out the contents of the owned `pointer`.

To support move-only `pointer` types, and additional constrained overload is supplied when `pointer` is not `CopyConstructible`. In this case it returns by (const) reference, and should allow for the usual `operator->` chaining rules. This might well be deemed an extension rather than a strictly required feature. It is supplied now as it is easier to remove, than to add it later.

Constraints on relational operators

This part of the design took a number of evolutions. The goal is to focus on supporting the minimal set of constraints, and therefore operators that must be implemented by the wrapped pointer types. It is well understood that for comparisons of the same type, the full set of 6 relational operators can be built from the pair of < and ==. An early evolution insisted that the pointer typedef of both `unique_ptr` types be the same, allowing us to rely on the `LessThanComparable` and `EqualityComaparable` concepts, and `std::relops` for an implementation. This approach was ultimately rejected when Howard Hinnant presented the following motivating example:

```
struct base {...};
struct derived : base {...};

vector<unique_ptr<base>> sorted_vec;
...
unique_ptr<derived> d(...);
vector<unique_ptr<base>>::iterator i =
     lower_bound(sorted_vec.begin(), sorted_vec.end(), d);
```

This desire to support heterogeneous comparison leads to the proposed formulation, which requires `HasLessThan` be supported in both directions. This does allow for occasionally strange types that will support the operators ==, !=, < and >=; but not >= and <. A third option would simply require `HasLessThan` in one direction, and use `HasEqualTo` to support the missing cases. This was rejected as being inefficient in some common cases, in order to support very unusual types. For reference, the library already takes the constrain-both-ways approach in several algorithms (equal_range, binary_search and the set operations). It is also the approach taken by `std::move_iterator` and consistency with this was the deciding factor.

A further research project might want to investigate a concept-enabled equivalent for a heterogeneous relops, which would be re-used here. Be advised though, that early exploration suggests that complexity may outweigh any benefit though.

Acknowledgements

I would particularly like to thank Howard Hinnant, the father of unique_ptr , for help and support preparing this paper. Likewise, Daniel Krugler, Peter Dimov and Walter Brown gave invaluable feedback in early reviews of the paper.

LWG Issues

The proposed wording also provides a solution for the following LWG issues, which should be considered resolved if this paper is adopted:

834, 854, 932, 938, 950, 1021

Checklist of issues not yet addressed:

933, 978, 983, 998

Proposed Wording

Note that this wording is draughted against N2800, and might need some tweaks to correspond to later working papers. It also applies the resolution of LWG issues 821 and paper N2844 adopted at the Summit meeting, and highlights a couple of editorial changes..

Additions and ~~deletions~~ are marked with the appropriate highlights.

Sections with a grey background have not been reviewed yet; this remains a work in progress.

20.7 Memory [memory]

1 **Header** <memory_concepts> **synopsis**

```
namespace std {

    // 20.7.12 Smart Pointer concepts
    auto concept< typename Ptr > NullablePointerLike {
        requires MoveConstructible< Ptr >
              && MoveAssignable< Ptr >
              && Convertible< nullptr_t, Ptr >
              && ExplicitlyConvertible< Ptr, bool >;

        requires EqualityComparable<Ptr>
        axiom NullptrEmptyEquivalence( Ptr & p ) {
            static_cast<bool>(p) == ( p == nullptr );
        }
    };

    concept< typename Deleter, typename T > SmartPointerDeleter {
        NullablePointerLike pointer = T*;

        requires Callable< Deleter, pointer >;
    };

    template< PointeeType T, typename D >
        requires Callable< D, T* >
        concept_map SmartPointerDeleter< D, T >;
}
```

2 **Header** <memory> **synopsis**

```
namespace std {
    // 20.7.12 Class unique_ptr:
    template <class T>
        requires True<true>
            class~~struct~~ default_delete;

    template <~~class~~PointeeType ~~X~~T
              , ~~class~~SmartPointerDeleter<auto, T> D = default_delete<T>>
        class unique_ptr;

    // 20.7.12.4 unique_ptr specialized algorithms:
    template <PointeeType T, SmartPointerDeleter<auto, T> D>
        requires Swappable<D> && Swappable<D::pointer>
        void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
    template< PointeeType T1, SmartPointerDeleter<auto, T1> D1
              , PointeeType T2, SmartPointerDeleter<auto, T2> D2 >
        requires HasEqualTo<D1::pointer, D2::pointer>
        bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< PointeeType T1, SmartPointerDeleter<auto, T1> D1
              , PointeeType T2, SmartPointerDeleter<auto, T2> D2 >
        requires HasEqualTo<D1::pointer, D2::pointer>
        bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< PointeeType T1, SmartPointerDeleter<auto, T1> D1
              , PointeeType T2, SmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D1::pointer, D2::pointer>
        bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< PointeeType T1, SmartPointerDeleter<auto, T1> D1
              , PointeeType T2, SmartPointerDeleter<auto, T2> D2 >
```

```
        requires HasLessThan<D2::pointer, D1::pointer>
        bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< PointeeType T1, SmartPointerDeleter<auto, T1> D1
            , PointeeType T2, SmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D2::pointer, D1::pointer>
        bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< PointeeType T1, SmartPointerDeleter<auto, T1> D1
            , PointeeType T2, SmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D1::pointer, D2::pointer>
        bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

}
```

20.7.12 Class template unique_ptr [unique.ptr]

1 Template unique_ptr stores a pointer to an object and deletes that object using the associated delete when it is itself destroyed (such as when leaving block scope (6.7)).

2 The unique_ptr provides a semantics of strict ownership. A unique_ptr owns the object it holds a pointer to. A unique_ptr is not CopyConstructible, nor CopyAssignable, however it is MoveConstructible and MoveAssignable. The template parameter T of unique_ptr may be an incomplete type. [*Note:* The uses of unique_ptr include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. —*end note*]

```
namespace std {
    template <class T>
        requires True<true>
        struct default_delete;
    template <class T>
        requires True<true>
        struct default_delete<T[]>;

    template <classPointeeType T
            , classSmartPointerDeleter<auto, T> D = default_delete<T>>
        class unique_ptr;

    template < classValueType T
             , classSmartPointerDeleter<auto, T> D = default_delete<T[]>
             >
         class unique_ptr<T[], D>;

    template <classPointeeType T, classSmartPointerDeleter<auto, T> D>
        requires Swappable<D> && Swappable<D::pointer>
        void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
    template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
            , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
        requires HasEqualTo<D1::pointer, D2::pointer>
        bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
            , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
        requires HasEqualTo<D2::pointer, D1::pointer>
        bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
            , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D1::pointer, D2::pointer>
        bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
            , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D2::pointer, D1::pointer>
        bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
            , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D2::pointer, D1::pointer>
        bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
    template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
            , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
        requires HasLessThan<D1::pointer, D2::pointer>
        bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
}
```

20.7.12.1 Default deleters [unique.ptr.dltr]

20.7.12.1.1 default_delete [unique.ptr.dltr.dflt]

```
namespace std {
    template <class T>
       requires True<true>
       struct default_delete {
          default_delete() = default;

          template <class U>
             requires Convertible<U*, T*>
                default_delete(const default_delete<U>&);

          requires FreeStoreAllocatable<T>
             void operator()(T*) const;
       };
    }
```

~~default_delete();~~

Effects: Default constructs a default_delete.

default_delete and all its specializations shall be empty POD types.

```
template <class U>
   requires Convertible<U*, T*>
      default_delete(const default_delete<U>& other);
```

1 *Effects:* Constructs a default_delete from a default_delete<U>.

```
requires FreeStoreAllocatable<T>
   void operator()(T *ptr) const;
```

2 *Effects:* calls delete on ptr. A diagnostic is required if T is an incomplete type.

20.7.12.1.2 default_delete<T[]> **[unique.ptr.dltr.dflt1]**

```
namespace std {
    template <class T>
       requires True<true>
       struct default_delete<T[]> {
          requires FreeStoreAllocatable<T>
             void operator()(T*) const;

          template <class U> void operator()(U*) const = delete;
       };
    }
```

```
requires FreeStoreAllocatable<T>
   void operator()(T* ptr) const;
```

1 operator() calls delete[] on ptr. A diagnostic is required if T is an incomplete type.

20.7.12.2 unique_ptr **for single objects** **[unique.ptr.single]**

```
namespace std {
    template <~~class~~PointeeType T
              , ~~class~~SmartPointerDeleter<auto, T> D = default_delete<T>>
    class unique_ptr {
    public:
       typedef ~~implementation-defined~~D::pointer  pointer;
       typedef T element_type;
       typedef D deleter_type;

       // constructors
       requires DefaultConstructible<D>
             && !PointerType<D>
             && !MemberPointerType<D>
          constexpr unique_ptr()~~;~~ : unique_ptr{nullptr} {}
       requires DefaultConstructible<D>
             && !PointerType<D>
             && !MemberPointerType<D>
          constexpr unique_ptr(nullptr_t) ~~: unique_ptr() { }~~;
       requires DefaultConstructible<D>
             && !PointerType<D>
             && !MemberPointerType<D>
          explicit unique_ptr(pointer p);
       requires MoveConstructible<D>
          unique_ptr(unique~~P~~_ptr&& u);
       unique_ptr(pointer p, implementation-defined d);
```

```
        unique_ptr(pointer p, implementation-defined d);
        template <classPointeeType U, classSmartPointerDeleter<auto, U> E>
           requires Constructible< pointer, E::pointer&& >
                 && Constructible< D, E&& >
              unique_ptr(unique_ptr<U, E>&& u);

        // destructor
        ~unique_ptr();

        // assignment
        requires MoveAssignable<D>
           unique_ptr& operator=(unique_ptr&& u);
        template <classPointeeType U, classSmartPointerDeleter<auto, U> E>
           requires Convertible< E::pointer &&, pointer >
                 && Convertible< E&&, D >
              unique_ptr& operator=(unique_ptr<U, E>&& u);
        unique_ptr& operator=(unspecified-pointer-typenullptr_t);

        // observers
        requires HasDereference<pointer>
           typename add_lvalue_reference<T>::type
            HasDereference<pointer>::result_type
           operator*() const;
        requires CopyConstructible<pointer>
           pointer operator->() const;
        requires !CopyConstructible<pointer>
           const pointer & operator->() const;

        requires CopyConstructible<pointer>
           pointer get() const;
        requires !CopyConstructible<pointer>
           const pointer & get() const;
        deleter_type& get_deleter();
        const deleter_type& get_deleter() const;
        explicit operator bool() const;

        // modifiers
        pointer release();
        void reset(pointer p = pointer()nullptr);
        requires Swappable<pointer>
              && Swappable<D>
           void swap(unique_ptr& u);

        // disable copy from lvalue
        unique_ptr(const unique_ptr&) = delete;
        template <class U, class E> unique_ptr(const unique_ptr<U, E>&) = delete;
        unique_ptr& operator=(const unique_ptr&) = delete;
        template <class U, class E> unique_ptr& operator=(const unique_ptr<U, E>&) =
            delete;
    };
}
```

1 The default type for the template parameter `D` is `default_delete`. A client-supplied template argument `D` shall be a ~~function pointer or functor~~`Callable` object type for which, given a value `d` of type `D` and a ~~pointer~~value `ptr` of type ~~`T*`~~`D::pointer`, the expression `d(ptr)` is valid and has the effect of deallocating the pointer as appropriate for that deleter. `D` may also be an lvalue-reference to a deleter.

2 If the deleter D maintains state, it is intended that this state stay with the associated pointer as ownership is transferred from `unique_ptr` to `unique_ptr`. The deleter state need never be copied, only moved or swapped as pointer ownership is moved around. ~~That is, the deleter need only be `MoveConstructible`, `MoveAssignable`, and `Swappable`, and need not be `CopyConstructible` (unless copied into the `unique_ptr`) nor `CopyAssignable`.~~

3 ~~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*`. The type `unique_ptr<T, D>::pointer` shall be CopyConstructible (Table 20.1.8) and CopyAssignable (Table 20.1.8).~~

20.7.12.2.1 unique_ptr **constructors** **[unique.ptr.single.ctor]**

```
requires DefaultConstructible<D>
      && !PointerType<D>
      && !MemberPointerType<D>
```

```
constexpr unique_ptr(nullptr_t);
```

1 *Requires:* ~~D shall be default constructible, and that construction~~ Default constructor for D shall not throw an exception. ~~D shall not be a reference type or pointer type (diagnostic required).~~

2 *Effects:* Constructs a `unique_ptr` which owns nothing.

3 *Postconditions:* ~~`get() == 0`~~`!*this`. `get_deleter()` returns a reference to a ~~default constructed~~ value initialized deleter D.

4 *Throws:* nothing.

```
requires DefaultConstructible<D>
      && !PointerType<D>
      && !MemberPointerType<D>
unique_ptr(pointer p);
```

5 *Requires:* ~~D shall be default constructible, and that construction~~ Neither the default constructor for D nor the move constructor for `pointer` shall ~~not~~ throw an exception.

6 *Effects:* Constructs a `unique_ptr` which owns p.

7 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to a ~~default constructed~~ value initialized deleter D.

8 *Throws:* nothing.

```
unique_ptr(pointer p, implementation-defined d);
unique_ptr(pointer p, implementation-defined d);
```

9 The signature of these constructors depends upon whether D is a reference type or not. If D is non-reference type A, then the signatures are:

```
unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, A&& d);
```

10 If D is an lvalue-reference type A&, then the signatures are:

```
unique_ptr(pointer p, A& d);
unique_ptr(pointer p, A&& d);
```

11 If D is an lvalue-reference type const A&, then the signatures are:

```
unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, const A&& d);
```

12 *Requires:* If D is not an lvalue-reference type then

— If d is an lvalue or const rvalue then the first constructor of this pair will be selected. D must be CopyConstructible (Table 20.1.8), and this unique_ptr will hold a copy of d. The copy constructor of D shall not throw an exception.

— Otherwise d is a non-const rvalue and the second constructor of this pair will be selected. D need only be MoveConstructible (Table 20.1.8), and this unique_ptr will hold a value *move constructed* from d. The move constructor of D shall not throw an exception.

13 Otherwise D is an lvalue-reference type. d shall be reference-compatible with one of the constructors. If d is an rvalue, it will bind to the second constructor of this pair. That constructor shall emit a diagnostic. [*Note:* The diagnostic could be implemented using a static_assert which assures that D is not a reference type. —*end note*] Else d is an lvalue and will bind to the first constructor of this pair. The type which D references need not be CopyConstructible nor MoveConstructible. This unique_ptr will hold a D which refers to the lvalue d. [*Note:* D may not be an rvalue-reference type. —*end note*]

14 *Postconditions:* get() == p. get_deleter() returns a reference to the internally stored deleter. If D is a reference type then get_deleter() returns a reference to the lvalue d.

15 *Throws:* nothing.

[*Example:*

```
D d;
unique_ptr<int, D> p1(new int, D());          // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d);            // D must be Copyconstructible
unique_ptr<int, D&> p3(new int, d);           // p3 holds a reference to d
```

unique_ptr<int, const D&> p4(new int, D()); *// error: rvalue deleter object combined*
// with reference deleter type
—end example]

```
requires MoveConstructible<D>
  unique_ptr(unique_ptr&& u);
```

Requires: If the deleter is not a reference type, construction of the deleter D from an rvalue D shall not throw an exception.

Effects: Constructs a `unique_ptr` which owns the pointer which u owns (if any). If the deleter is not a reference type, it is move constructed from u's deleter, otherwise the reference is copy constructed from u's deleter. After the construction, u no longer owns a pointer. [*Note:* The deleter constructor can be implemented with `std::forward<D>`. *—end note*]

Postconditions: `get() == value u.get()` had before the construction. `get_deleter()` returns a reference to the internally stored deleter which was constructed from `u.get_deleter()`. If D is a reference type then `get_deleter()` and `u.get_deleter()` both reference the same lvalue deleter.

Throws: nothing.

```
template <classPointeeType U, classSmartPointerDeleter<auto, U> E>
   requires Constructible< pointer, E::pointer&& >
         && Constructible< D, E&& >
     unique_ptr(unique_ptr<U, E>&& u);
```

Requires: If D is not a reference type, construction of the deleter D from an rvalue of type E shall be well formed and shall not throw an exception. If D is a reference type, then E shall be the same type as D (diagnostic required). unique_ptr<U, E>::pointer shall be implicitly convertible to pointer. [*Note:* These requirements imply that T and U are complete types. *—end note*]

Effects: Constructs a unique_ptr which owns the pointer which u owns (if any). If the deleter is not a reference type, it is move constructed from u's deleter, otherwise the reference is copy constructed from u's deleter. After the construction, u no longer owns a pointer. [*Note:* The deleter constructor can be implemented with std::forward<D>. *—end note*]

Postconditions: get() == value u.get() had before the construction, modulo any required offset adjustments resulting from the cast from unique_ptr<U, E>::pointer to pointer. get_deleter() returns a reference to the internally stored deleter which was constructed from u.get_deleter().

Throws: nothing.

20.7.12.2.2 unique_ptr **destructor** **[unique.ptr.single.dtor]**

```
~unique_ptr();
```

Requires: The expression get_deleter()(get()) shall be well formed, shall have well defined behavior, and shall not throw exceptions. [*Note:* The use of default_delete requires T to be a complete type. *—end note*]

Effects: If `get() == 0` `!*this` there are no effects. Otherwise `get_deleter()(get())`. [*Note:* The use of default_delete requires T to be a complete type. *—end note*]

Throws: nothing.

20.7.12.2.3 unique_ptr **assignment** **[unique.ptr.single.asgn]**

```
requires MoveAssignable<D>
  unique_ptr& operator=(unique_ptr&& u);
```

Requires: Assignment of the deleter D from an rvalue D shall not throw an exception.

Effects: `reset(u.release())` followed by a move assignment from u's deleter to this deleter.

Postconditions: This unique_ptr now owns the pointer which u owned, and u no longer owns it. [*Note:* If D is a reference type, then the referenced lvalue deleters are move assigned. *—end note*]

Returns: *this.

Throws: nothing.

```
template <classPointeeType U, classSmartPointerDeleter<auto, U> E>
    requires Convertible< E::pointer &&, pointer >
          && Convertible< E&&, D >
      unique_ptr& operator=(unique_ptr<U, E>&& u);
```

Requires: Assignment of the deleter D from an rvalue D shall not throw an exception. ~~unique_ptr<U, E>::pointer shall be implicitly convertible to pointer. [*Note:* These requirements imply that T and U are complete types. — *end note*]~~

Effects: `reset(u.release())` followed by a move assignment from u's deleter to this deleter. If either D or E is a reference type, then the referenced lvalue deleter participates in the move assignment.

Postconditions: This unique_ptr now owns the pointer which u owned, and u no longer owns it.

Returns: *this.

Throws: nothing.

```
unique_ptr& operator=(~~unspecified-pointer-type~~nullptr_t);
```

~~Assigns from the literal 0 or NULL. [*Note:* The *unspecified-pointer-type* is often implemented as a pointer to a private data member, avoiding many of the implicit conversion pitfalls. — *end note*]~~

Effects: reset().

Postcondition: ~~`get() == 0`~~`!*this`.

Returns: *this.

Throws: nothing.

20.7.12.2.4 unique_ptr **observers** **[unique.ptr.single.observers]**

```
requires HasDereference<pointer>
    ~~typename add_lvalue_reference<T>::type~~HasDereference<pointer>::result_type
    operator*() const;
```

Requires: ~~`get() == 0`~~`!*this`.

Returns: *get().

Throws: nothing.

```
requires CopyConstructible<pointer>
   pointer operator->() const;
requires !CopyConstructible<pointer>
   const pointer & operator->() const;
```

Requires: ~~`get() == 0`~~`!*this`.

Returns: get().

Throws: nothing.

Note: use typically requires that T be a complete type.

```
requires CopyConstructible<pointer>
   pointer get() const;
requires !CopyConstructible<pointer>
   const pointer & get() const;
```

Returns: The stored pointer.

Throws: nothing.

```
deleter_type& get_deleter();
const deleter_type& get_deleter() const;
```

Returns: A reference to the stored deleter.

Throws: nothing.

```
explicit operator bool() const;
```

12 *Returns:* ~~get() != 0~~static_cast<bool>(this->get()).

13 *Throws:* nothing.

20.7.12.2.5 unique_ptr **modifiers** **[unique.ptr.single.modifiers]**

```
pointer release();
```

1 *Postcondition:* ~~get() == 0~~!*this.

2 *Returns:* The value get() had at the start of the call to release.

3 *Throws:* nothing.

```
void reset(pointer p = ~~pointer()~~nullptr);
```

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

5 *Effects:* If ~~get() == 0~~!*this there are no effects. Otherwise `get_deleter()(get())`.

6 *Postconditions:* get() == p.

7 *Throws:* nothing.

```
requires Swappable<pointer> && Swappable<D>
   void swap(unique_ptr& u);
```

8 *Requires:* The deleter D ~~shall be Swappable~~ and pointer type `pointer` shall not throw an exception under `swap`.

9 *Effects:* The stored pointers of `*this` and `u` are exchanged. The stored deleters are `swap`'d (unqualified).

10 *Throws:* nothing.

20.7.12.3 unique_ptr **for array objects with a runtime length** **[unique.ptr.runtime]**

```
namespace std {
    template < ~~class~~ValueType T
             , ~~class~~SmartPointerDeleter<auto, T> D = default_delete<T[]>
             >
    class unique_ptr<T[], D> {
    public:
       typedef ~~implementation-defined~~D::pointer pointer;
       typedef T element_type;
       typedef D deleter_type;

       // constructors
       requires DefaultConstructible<D>
             && !PointerType<D>
             && !MemberPointerType<D>
          constexpr unique_ptr()~~;~~ : unique_ptr{nullptr} {}
       requires DefaultConstructible<D>
             && !PointerType<D>
             && !MemberPointerType<D>
          constexpr unique_ptr(nullptr_t) ~~: unique_ptr() { }~~;
       requires DefaultConstructible<D>
             && !PointerType<D>
             && !MemberPointerType<D>
       explicit unique_ptr(pointer p);
       unique_ptr(pointer p, implementation-defined d);
       unique_ptr(pointer p, implementation-defined d);
       requires MoveConstructible<D>
          unique_ptr(uniquePtr&& u);

       // destructor
       ~unique_ptr();

       // assignment
       unique_ptr& operator=(unique_ptr&& u);
       unique_ptr& operator=(~~unspecified-pointer-type~~nullptr_t);

       // observers
       requires HasSubscript<pointer, size_t>
```

```
        T&HasSubscript<pointer, size_t>::result_type operator[](size_t i) const;
    requires CopyConstructible<pointer>
        pointer get() const;
    requires !CopyConstructible<pointer>
        const pointer & get() const;
    deleter_type& get_deleter();
    const deleter_type& get_deleter() const;
    explicit operator bool() const;

    // modifiers
    pointer release();
    void reset(pointer p = pointer()nullptr);
    void reset( std::nullptr_t );
    template< typename U >
        requires Convertible< U, pointer >
            void reset( U ) = delete;
    requires Swappable<pointer>
            && Swappable<D>
        void swap(unique_ptr& u);

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

    template< typename U >
        requires Convertible< U, pointer >
            unique_ptr( U ) = delete;
};
}
```

1 A specialization for array types is provided with a slightly altered interface.

— Conversions among different types of unique_ptr<T[], D> or to or from the non-array forms of unique_ptr are disallowed (diagnostic required).
— 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[].

2 Descriptions are provided below only for member functions that have behavior different from the primary template.

3 The template argument T shall be a complete type.

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

```
requires DefaultConstructible<D>
      && !PointerType<D>
      && !MemberPointerType<D>
unique_ptr(pointer p);
unique_ptr(pointer p, implementation-defined d);
unique_ptr(pointer p, implementation-defined d);
```

These constructors behave the same as in the primary template except that they do not accept pointer types which are convertible to pointer. [*Note:* One implementation technique is to create private templated overloads of these members. — *end note*]

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

```
requires HasSubscript<pointer, size_t>
    T&HasSubscript<pointer, size_t>::result_type operator[](size_t i) const;
```

1 *Requires:* i < the size of the array to which the stored pointer points.

2 *Returns:* get()[i].

3 *Throws:* nothing.

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

```
void reset(pointer p = pointer()nullptr);
void reset(std::nullptr_t );
```

2 *Effects:* If get() == 0!*this there are no effects. Otherwise `get_deleter()(get())`.

3 *Postcondition:* `get() == p`.

4 *Throws:* nothing.

20.7.12.4 unique_ptr **specialized algorithms** **[unique.ptr.special]**

```
template <classPointeeType T, classSmartPointerDeleter<auto, T> D>
   requires Swappable<D> && Swappable<D::pointer>
      void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
```

1 *Effects:* Calls x.swap(y).

```
template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
        , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
   requires HasEqualTo<D1::pointer, D2::pointer>
      bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

2 *Returns:* `x.get() == y.get()`.

```
template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
        , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
   requires HasEqualTo <D1::pointer, D2::pointer>
      bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

3 *Returns:* ~~`x.get() != y.get()`~~`!(x==y)`.

```
template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
        , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
   requires HasLessThan<D1::pointer, D2::pointer>
      bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

4 *Returns:* `x.get() < y.get()`.

```
template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
        , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
   requires HasLessThan<D2::pointer, D1::pointer>
      bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

5 *Returns:* ~~`x.get() <= y.get()`~~`!(y < x)`.

```
template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
        , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
   requires HasLessThan<D2::pointer, D1::pointer>
   bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

6 *Returns:* ~~`x.get() > y.get()`~~`y < x`.

```
template< classPointeeType T1, classSmartPointerDeleter<auto, T1> D1
        , classPointeeType T2, classSmartPointerDeleter<auto, T2> D2 >
   requires HasLessThan<D1::pointer, D2::pointer>
   bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

7 *Returns:* ~~`x.get() >= y.get()`~~`!(x < y)`.

Preview of constraints for shared_ptr

As this paper was going to press, Peter Dimov supplied the following class template definition for a constrained shared_ptr template. This will be the basis for including full wording for shared_ptr constraints in the next paper, and feedback is invited.

New Concepts

Shared_ptr relies on a small number of compiler-supported concepts that might be added to the earlier concepts headers:

```
// Suggested new concepts

concept Deletable<typename P>;
// The expression delete p; is well-formed and its behavior is defined
// In particular, p shall not be a pointer to an incomplete type
// FreeStoreAllocatable might refine Deletable

concept HasStaticCast<typename From, typename To>;
// static_cast<To>( From ) is well-formed and well-defined

concept HasDynamicCast<typename From, typename To>;
// dynamic_cast<To>( From ) is well-formed and well-defined

concept HasConstCast<typename From, typename To>;
// const_cast<To>( From ) is well-formed and well-defined
```

Shared_ptr template

```
// shared_ptr

template<class PointeeType T>
class shared_ptr
{
public:

    typedef T element_type;

    // 20.7.13.2.1, constructors:
    shared_ptr();

    template<class PointeeType Y>
        requires Convertible<Y*, T*> && Deletable<Y*>
        explicit shared_ptr(Y* p);

    template<class PointeeType Y, class MoveConstructible D>
        requires Convertible<Y*, T*> && Callable<D, Y*>
        shared_ptr(Y* p, D d);

    template<class PointeeType Y, class MoveConstructible D, class Allocator A>
        requires Convertible<Y*, T*> && Callable<D, Y*>
        shared_ptr(Y* p, D d, A a);

    template<class PointeeType Y>
        requires Convertible<Y*, T*>
        shared_ptr(const shared_ptr<Y>& r, T *p);

    shared_ptr(const shared_ptr& r);

    template<class PointeeType Y>
        requires Convertible<Y*, T*>
        shared_ptr(const shared_ptr<Y>& r);

    shared_ptr(shared_ptr&& r);

    template<class PointeeType Y>
        requires Convertible<Y*, T*>
        shared_ptr(shared_ptr<Y>&& r);

    template<class PointeeType Y>
        requires Convertible<Y*, T*>
        explicit shared_ptr(const weak_ptr<Y>& r);
```

```
    // Note: suggesting implicit
    template<~~class~~PointeeType Y>
        requires Convertible<Y*, T*> && Deletable<Y*>
        ~~explicit~~ shared_ptr(auto_ptr<Y>&& r);

    // No longer needed if && doesn't bind to lvalues
    ~~template<class Y, class D>~~
        ~~explicit shared_ptr(const unique_ptr<Y, D>& r) = delete;~~

    // Note: suggesting implicit
    // SmartPointerDeleter<D, Y> and Callable<D, D::pointer> are implied
    template<~~class~~PointeeType Y, class D>
        requires PointerType<D::pointer> && Convertible<D::pointer, T*>
        ~~explicit~~ shared_ptr(unique_ptr<Y, D>&& r);

    shared_ptr(nullptr_t): shared_ptr() {}

    // 20.7.13.2.2, destructor:
    ~shared_ptr();

    // 20.7.13.2.3, assignment:
    shared_ptr& operator=(const shared_ptr& r);

    template<~~class~~PointeeType Y>
        requires Convertible<Y*, T*>
        shared_ptr& operator=(const shared_ptr<Y>& r);

    shared_ptr& operator=(shared_ptr&& r);

    template<~~class~~PointeeType Y>
        requires Convertible<Y*, T*>
        shared_ptr& operator=(shared_ptr<Y>&& r);

    template<~~class~~PointeeType Y>
        requires Convertible<Y*, T*> && Deletable<Y*>
        shared_ptr& operator=(auto_ptr<Y>&& r);

    // No longer needed if && doesn't bind to lvalues
    ~~template <class Y, class D>~~
        ~~shared_ptr& operator=(const unique_ptr<Y, D>& r) = delete;~~

    // SmartPointerDeleter<D, Y> and Callable<D, D::pointer> are implied
    template<~~class~~PointeeType Y, class D>
        requires PointerType<D::pointer> && Convertible<D::pointer, T*>
        shared_ptr& operator=(unique_ptr<Y, D>&& r);

    // 20.7.13.2.4, modifiers:
    void swap(shared_ptr& r);

    void reset();

    template<~~class~~PointeeType Y>
        requires Convertible<Y*, T*> && Deletable<Y*>
        void reset(Y* p);

    template<~~class~~PointeeType Y, MoveConstructible D>
        requires Convertible<Y*, T*> && Callable<D, Y*>
        void reset(Y* p, D d);

    template<~~class~~PointeeType Y, MoveConstructible D, Allocator A>
        requires Convertible<Y*, T*> && Callable<D, Y*>
        void reset(Y* p, D d, A a);

    // 20.7.13.2.5, observers:
    T* get() const;

    requires ReferentType<T>
        T& operator*() const;

    T* operator->() const;
    long use_count() const;
    bool unique() const;
    explicit operator bool() const;

    template <~~class~~PointeeType U> bool owner_before(shared_ptr<U> const& b) const;
    template <~~class~~PointeeType U> bool owner_before(weak_ptr<U> const& b) const;
};
```

```
// 20.7.13.2.6, shared_ptr creation
template<class T, class... Args>
    requires Constructible<T, Args...>
    shared_ptr<T> make_shared(Args&&... args);

template<class T, Allocator A, class... Args>
    requires Constructible<T, Args...>
    shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.7.13.2.7, shared_ptr comparisons:
template<~~class~~PointeeType T, ~~class~~PointeeType U>
    requires HasEqualTo<T*, U*>
    bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b);

template<~~class~~PointeeType T, ~~class~~PointeeType U>
    requires HasNotEqualTo<T*, U*>
    bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b);

// This operator is a defect under the new < semantics
// Heterogeneous < comparisons using < make no sense
~~template<class T, class U>~~
    ~~bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b);~~

// Suggested:
template<PointeeType T>
    bool operator<(const shared_ptr<T>& a, const shared_ptr<T>& b);

// 20.7.13.2.8, shared_ptr I/O:

// basic_ostream is not conceptified
// not clear what we should do here

template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>&
p);

// 20.7.13.2.9, shared_ptr specialized algorithms:
template<~~class~~PointeeType T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// 20.7.13.2.10, shared_ptr casts:
template<~~class~~PointeeType T, ~~class~~PointeeType U>
    requires HasStaticCast<U*, T*>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);

template<~~class~~PointeeType T, ~~class~~PointeeType U>
    requires HasDynamicCast<U*, T*>
    shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r);

template<~~class~~PointeeType T, ~~class~~PointeeType U>
    requires HasConstCast<U*, T*>
    shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r);

// 20.7.13.2.11, shared_ptr get_deleter:
template<~~class~~ObjectType D,~~class~~PointeeType T>
    D* get_deleter(const shared_ptr<T>& p);
```