A cloning pointer-class for C++

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P0201R0

Working Group: Library

Date: 2016-02-01

Jonathan Coe <jbcoe@me.com>

Ville Voutilainen <ville.voutilainen@gmail.com>

TL;DR

Add a class template, `cloned_ptr<T>`, to the standard library to allow compiler-generated copy constructors to correctly copy composite objects with polymorphic components.

Introduction

We propose the addition of a class template, `cloned_ptr`, for which the copy constructor copies the pointee. W.E.Brown proposed a deep-copying pointer [N3339] where copying the pointer copies the pointee. We propose a deeper-still-copy that is able to copy derived-type objects through base-type pointers.

Motivation: Composite objects

The class template, `cloned_ptr`, is designed to allow a class with polymorphic components to be correctly copied by a compiler-generated copy constructor.

Use of components in the design of object-oriented class hierarchies can aid modular design as components can be potentially re-used as building-blocks for other composite classes.

We can write a simple composite object formed from two components as follows:

```
// Simple composite
class CompositeObject_1 {
  Component1 c1_;
  Component2 c2_;

 public:
  CompositeObject_1(const Component1& c1,
                    const Component2& c2) :
                    c1_(c1), c2_(c2) {}
```

```
  void foo() { c1_.foo(); }
  void bar() { c2_.bar(); }
};
```

The composite object can be made more flexible by storing pointers to objects allowing it to take derived components in its constructor. (We store pointers to the components rather than references so that we can take ownership of them).

```
// Non-copyable composite with polymorphic components (BAD)
class CompositeObject_2 {
  IComponent1* c1_;
  IComponent2* c2_;

 public:
  CompositeObject_2(const IComponent1* c1,
                    const IComponent2* c2) :
                    c1_(c1), c2_(c2) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }

  CompositeObject_2(const CompositeObject_2&) = delete;
  CompositeObject_2& operator=(const CompositeObject_2&) = delete;

  CompositeObject_2(CompositeObject_2&& o) : c1_(o.c1_), c2_(o.c2_) {
    o.c1_ = nullptr;
    o.c2_ = nullptr;
  }

  CompositeObject_2& operator=(CompositeObject_2&& o) {
    delete c1_;
    delete c2_;
    c1_ = o.c1_;
    c2_ = o.c2_;
    o.c1_ = nullptr;
    o.c2_ = nullptr;
  }

  ~CompositeObject_2()
  {
    delete c1_;
    delete c2_;
  }
};
```

`CompositeObject_2`'s constructor API is unclear without knowing that the class takes ownership of the objects. We are forced to explicitly suppress the compiler-generated copy constructor and copy assignment operator to avoid double-deletion of the components `c1_` and `c2_`. We also need to write a move constructor and move assignment operator.

Using `unique_ptr` makes ownership clear and saves us writing or deleting compiler generated methods:

```
// Non-copyable composite with polymorphic components
class CompositeObject_3 {
  std::unique_ptr<IComponent1> c1_;
  std::unique_ptr<IComponent2> c2_;

 public:
  CompositeObject_3(std::unique_ptr<IComponent1> c1,
                    std::unique_ptr<IComponent2> c2) :
                    c1_(std::move(c1)), c2_(std::move(c2)) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }
};
```

The design of `CompositeObject_3` is good unless we want to copy the object.

We can avoid having to define our own copy constructor by using shared pointers. As `shared-ptr`'s copy constructor is shallow, we need to modify the component pointers to be pointers-to `const` to avoid introducing shared mutable state [S.Parent].

```
// Copyable composite with immutable polymorphic components class
class CompositeObject_4 {
  std::shared_ptr<const IComponent1> c1_;
  std::shared_ptr<const IComponent2> c2_;

 public:
  CompositeObject_4(std::shared_ptr<const IComponent1> c1,
                    std::shared_ptr<const IComponent2> c2) :
                    c1_(std::move(c1)), c2_(std::move(c2)) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }
};
```

`CompositeObject_4` has polymorphism and compiler-generated destructor, copy, move and assignement operators. As long as the components are not mutated, this design is good. If non-const methods of components are used then this won't compile.

Using `cloned_ptr` a copyable composite object with polymorphic components can be written as:

```
// Copyable composite with mutable polymorphic components
class CompositeObject_5 {
  std::cloned_ptr<IComponent1> c1_;
  std::cloned_ptr<IComponent2> c2_;

 public:
  CompositeObject_5(std::cloned_ptr<IComponent1> c1,
                    std::cloned_ptr<IComponent2> c2) :
                    c1_(std::move(c1)), c2_(std::move(c2)) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }
};
```

`CompositeObject_5` has a (correct) compiler-generated destructor, copy, move, and assignment operators. In addition to enabling compiler-generation of functions, `cloned_ptr` performs deep copies of `c1_` and `c2_` without the class author needing to provide a special 'clone' method.

Deep copies

To allow correct copying of polymorphic objects, `cloned_ptr` uses the copy constructor of the derived-type pointee when copying a base type `cloned_ptr`. Similarly, to allow correct destruction of polymorphic component objects, `cloned_ptr` uses the destructor of the derived-type pointee in the destructor of a base type `cloned_ptr`.

The requirements of deep-copying can be illustrated by some simple test code:

```
// GIVEN base and derived classes.
class Base { virtual void foo() const = 0; };
class Derived : Base { void foo() const override {} };

// WHEN a cloned_ptr to base is formed from a derived pointer
cloned_ptr<Base> dptr(new Derived());
// AND the cloned_ptr to base is copied.
auto dptr_copy = dptr;

// THEN the copy points to a distinct object
assert(dptr.get() != dptr_copy.get());
// AND the copy points to a derived type.
assert(dynamic_cast<Derived*>(dptr_copy.get());
```

Note that while deep-destruction of a derived class object from a base class pointer can be performed with a virtual destructor, the same is not true for deep-copying. C++ has no concept of a virtual copy constructor and we are not proposing its addition. The class template `shared_ptr` already implements deep-destruction without needing virtual destructors. deep-destruction and deep-copying can be implemented using type-erasure [Impl].

Impact on the standard

This proposal is a pure library extension. It requires additions to be made to the standard library header `memory`.

Technical specifications

X.Y Class template `cloned_ptr` [cloned.ptr]

X.Y.1 Class template `cloned_ptr` general [cloned.ptr.general]

A *cloned pointer* is an object that owns another object and manages that other object through a pointer. A cloned pointer will copy the managed object when it is copied so that each copy of a cloning pointer has its own distinct copy of the managed object. A cloned pointer, `u`, will dispose of its managed object when `u` is destroyed. A cloned pointer object is empty if it does not own a pointer. The template parameter `T` of `cloned_ptr` may be an incomplete type.

[*Note: `cloned_ptr` is designed to enable the compiler-generated copy, move and assignment operations to behave correctly for classes with polymorphic components. —endnote*]

X.Y.2 Class template `cloned_ptr` synopsis [cloned.ptr.synopsis]

```
namespace std {
  template <class T> class cloned_ptr {
   public:
    // Constructors
    cloned_ptr() noexcept;
    cloned_ptr(std::nullptr_t) noexcept;
    template <class U> explicit cloned_ptr(U* p); // see below
    cloned_ptr(const cloned_ptr& p);
    template <class U> cloned_ptr(const cloned_ptr<U>& p); // see below
    cloned_ptr(cloned_ptr&& p) noexcept;
    template <class U> cloned_ptr(cloned_ptr<U>&& p); // see below

    // Destructor
    ~cloned_ptr();

    // Assignment
```

```
  cloned_ptr &operator=(const cloned_ptr& p);
  template <class U> cloned_ptr &operator=(const cloned_ptr<U>& p); // see below
  cloned_ptr &operator=(cloned_ptr &&p) noexcept;
  template <class U> cloned_ptr &operator=(cloned_ptr<U>&& p); // see below

  // Modifiers
  void swap(cloned_ptr<T>& p) noexcept;
  T* release() noexcept;
  template <class U> void reset(U* p); // see below

  // Observers
  T* get() const noexcept;
  T& operator*() const noexcept;
  T* operator->() const noexcept;
  operator bool() const noexcept;
};

// cloned_ptr creation
template <class T, class ...Ts> cloned_ptr<T>
  make_cloned_ptr(Ts&& ...ts); // see below

// cloned_ptr comparison
template <typename T, typename U>
  bool operator==(const cloned_ptr<T> &t, const cloned_ptr<U> &u) noexcept;
template <typename T, typename U>
  bool operator!=(const cloned_ptr<T> &t, const cloned_ptr<U> &u) noexcept;
template <typename T, typename U>
  bool operator<(const cloned_ptr<T> &t, const cloned_ptr<U> &u) noexcept;
template <typename T, typename U>
  bool operator>(const cloned_ptr<T> &t, const cloned_ptr<U> &u) noexcept;
template <typename T, typename U>
  bool operator<=(const cloned_ptr<T> &t, const cloned_ptr<U> &u) noexcept;
template <typename T, typename U>
  bool operator>=(const cloned_ptr<T> &t, const cloned_ptr<U> &u) noexcept;
template <typename T>
  bool operator==(const cloned_ptr<T> &t, std::nullptr_t) noexcept;
template <typename T>
  bool operator==(std::nullptr_t, const cloned_ptr<T> &t) noexcept;
template <typename T>
  bool operator!=(const cloned_ptr<T> &t, std::nullptr_t) noexcept;
template <typename T>
  bool operator!=(std::nullptr_t, const cloned_ptr<T> &t) noexcept;
template <typename T>
  bool operator<(const cloned_ptr<T> &t, std::nullptr_t) noexcept;
template <typename T>
  bool operator<(std::nullptr_t, const cloned_ptr<T> &t) noexcept;
```

```
  template <typename T>
    bool operator>(const cloned_ptr<T> &t, std::nullptr_t) noexcept;
  template <typename T>
    bool operator>(std::nullptr_t, const cloned_ptr<T> &t) noexcept;
  template <typename T>
    bool operator<=(const cloned_ptr<T> &t, std::nullptr_t) noexcept;
  template <typename T>
    bool operator<=(std::nullptr_t, const cloned_ptr<T> &t) noexcept;
  template <typename T>
    bool operator>=(const cloned_ptr<T> &t, std::nullptr_t) noexcept;
  template <typename T>
    bool operator>=(std::nullptr_t, const cloned_ptr<T> &t) noexcept;

  // cloned_ptr specialized algorithms
  void swap(cloned_ptr<T>& p, cloned_ptr<T>& u) noexcept;

  // cloned_ptr casts
  template <typename T, typename U>
    cloned_ptr<T> static_pointer_cast(const cloned_ptr<U> &p);
  template <typename T, typename U>
    cloned_ptr<T> dynamic_pointer_cast(const cloned_ptr<U> &p);
  template <typename T, typename U>
    cloned_ptr<T> const_pointer_cast(const cloned_ptr<U> &p);

  // cloned_ptr I/O
  template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os,
                                     const cloned_ptr<Y>& p);

  //cloned_ptr hash support
  template <class T> struct hash<cloned_ptr<T>>;

} // end namespace std
```

X.Y.3 Class template `cloned_ptr` constructors [cloned.ptr.ctor]

```
cloned_ptr() noexcept;
cloned_ptr(std::nullptr_t) noexcept;
```

- *Effects:* Constructs an empty `cloned_ptr`.
- *Postconditions:* `get() == nullptr`

```
template <class U> explicit cloned_ptr(U* p);
```

- *Effects*: Creates a `cloned_ptr` object that *owns* the pointer `p`.
- *Postconditions:* `get() == p`

- *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.
- *Exception safety:* If an exception is thrown, `delete p` is called.
- *Requires:* `U` is copy-constructible.
- *Remarks:* This constructor shall not participate in overload resolution unless `U*` is convertible to `T*`.

[*Note: When a `cloned_ptr` is copied the resource is copied using the copy constructor of `U`. —endnote*]

```
cloned_ptr(const cloned_ptr &p);
template <class U> cloned_ptr(const cloned_ptr<U> &p);
```

- *Remarks:* The second constructor shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects:* Creates a `cloned_ptr` object that owns a copy of the resource managed by `p`.
- *Postconditions:* If `p` is non-empty `get() != p.get()`. Otherwise `get() == nullptr`.

```
cloned_ptr(cloned_ptr &&p) noexcept;
template <class U> cloned_ptr(cloned_ptr<U> &&p);
```

- *Remarks*: The second constructor shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects:* Move-constructs a `cloned_ptr` instance from `p`.
- *Postconditions:* `*this` shall contain the old value of `p`. `p` shall be empty. `p.get() == nullptr`.

X.Y.4 Class template `cloned_ptr` destructor [cloned.ptr.dtor]

```
~cloned_ptr();
```

- *Effects:* If `*this` *owns* a pointer `p` then `delete p` is called.

X.Y.5 Class template `cloned_ptr` assignment [cloned.ptr.assignment]

```
cloned_ptr &operator=(const cloned_ptr &p);
template <class U> cloned_ptr &operator=(const cloned_ptr<U>& p);
```

- *Remarks*: The second function shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects:* `*this` shall own a copy of the resource managed by `p`.

- *Returns:* `*this`.
- *Postconditions:* If `p` is non-empty `get() != p.get()`. Otherwise `get() == nullptr`.

```
cloned_ptr &operator=(cloned_ptr&& p) noexcept;
template <class U> cloned_ptr &operator=(cloned_ptr<U> &&p);
```

- *Remarks*: The second function shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects:* `*this` shall own a copy of the resource managed by `p`.
- *Returns:* `*this`.
- *Postconditions:* `*this` shall contain the old value of `p`. `p` shall be empty. `p.get() == nullptr`.

X.Y.6 Class template `cloned_ptr` modifiers [cloned.ptr.modifiers]

```
void swap(cloned_ptr<T>& p) noexcept;
```

- *Effects:* Exchanges the contents of `p` and `*this`.

```
T* release() noexcept;
```

- *Effects:*
- – `*this` is empty and no longer owns the managed resource.
- – The resource is not deleted.
- *Postconditions:* `get() == nullptr`.
- *Returns:* The value `get()` had at the start of the call to `release()`.

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

- *Effects:* Equivalent to `cloned_ptr(p).swap(*this)`.

X.Y.7 Class template `cloned_ptr` observers [cloned.ptr.observers]

```
T* get() const noexcept;
```

- *Returns:* The stored pointer `p_`.

```
T& operator*() const noexcept;
```

- *Requires:* `get()!=nullptr`.
- *Returns:* `*get()`.

```
T* operator->() const noexcept;
```

- *Requires:* `get()!=nullptr`.

- *Returns:* get().

```
operator bool() const noexcept;
```

- *Returns:* get()!=nullptr.

X.Y.8 Class template cloned_ptr creation [cloned.ptr.creation]

```
template <class T, class ...Ts> cloned_ptr<T>
  make_cloned_ptr(Ts&& ...ts);
```

- *Returns:* cloned_ptr<T>(new T(std::forward<Ts>(ts)...);

X.Y.9 Class template cloned_ptr comparison [cloned.ptr.comparison]

Identical to shared_ptr (which looks underspecified).

X.Y.10 Class template cloned_ptr specialized algorithms [cloned.ptr.spec]

```
template <typename T>
void swap(cloned_ptr<T>& p, cloned_ptr<T>& u) noexcept;
```

- *Effects:* Equivalent to p.swap(u).

X.Y.11 Class template cloned_ptr casts [cloned.ptr.casts]

```
template <typename T, typename U>
  cloned_ptr<T> static_pointer_cast(const cloned_ptr<U>& p);
```

- *Requires:* The expression static_cast<T*>(p.get()) shall be well-formed.
- *Returns:* If p is empty, an empty cloned_ptr<T>; otherwise a cloned_ptr<T> which owns a copy of the resource in p.
- *Postconditions:* If p is non-empty, w.get() != static_cast<T*>(p.get()) where w is the return value.

```
template <typename T, typename U>
  cloned_ptr<T> dynamic_pointer_cast(const cloned_ptr<U>& p);
```

- *Requires:* The expression dynamic_cast<T*>(p.get()) shall be well-formed and have well-defined behaviour.
- *Returns:*
-
  - When dynamic_cast<T*>(p.get()) returns a non-null value, a cloned_ptr<T> which owns a copy of the resource in p.

- – Otherwise an empty `cloned_ptr<T>`.
- *Postconditions:* If `p` is non-empty, `w.get() != dynamic_cast<T*>(p.get())` where `w` is the return value.

```
template <typename T, typename U>
  cloned_ptr<T> const_pointer_cast(const cloned_ptr<U>& p);
```

- *Requires:* The expression `const_cast<T*>(p.get())` shall be well-formed.
- *Returns:* If `p` is empty, an empty `cloned_ptr<T>`; otherwise a `cloned_ptr<T>` which owns a copy of the resource in `p`.
- *Postconditions:* If `p` is non-empty, `w.get() != const_cast<T*>(p.get())` where `w` is the return value.

X.Y.12 Class template `cloned_ptr` I/O [cloned.ptr.io]

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

- *Effects:* `os << p.get()`.
- *Returns:* `os`.

X.Y.13 Class template `cloned_ptr` hash support [cloned.ptr.hash]

```
template <class T> struct hash<cloned_ptr<T>>;
```

The template specialization shall meet the requirements of class template hash (20.9.12). For an object `p` of type `cloned_ptr<T>`, `hash<cloned_ptr<T> >()(p)` shall evaluate to the same value as `hash<T*>()(p.get())`.

Feedback

The authors would like feedback from the committee on the issues below.

Public pointer-constructor

The proposal has a public pointer-constructor template

```
template<class U> cloned_ptr(U* u)
```

This is consistent with `unique_ptr` and `shared_ptr` but makes `cloned_ptr` potentially unsafe.

The deleted and copied type will depend on the static type `U` which could be different from the dynamic type of the pointer. For instance:

```
struct A {/* data members */};
struct B : A {/* data members */};
struct C : B {/* data members */};

auto uptr_c = std::make_unique<C>();
B* p_b = uptr_c.release();
auto dptr_b = cloned_ptr<A>(p_b);
auto dptr_b2 = cloned_ptr<A>(dptr_b);
```

`dptr_b` will have a static type of `B`, not `C`, and will be copied and deleted as a `B`. This will verly likely result in slicing and incorrect behaviour.

The destructor of `shared_ptr` has the same issue but it can be fixed by making destructors virtual. There is not such fix for copy constructors so `cloned_ptr` remains vulnerable to dynamic/static type mismatches.

The authors offer `make_cloned_ptr` as a fix to the possible dynamic/static type mismatch issue above. Using `make_cloned_ptr` in place of the pointer-constructor will prevent dynamic/static type mismatches. In addition, a static analysis tool could be developed to detect possible mismatches.

The authors request the following straw poll:

- *The pointer-constructor of `cloned_ptr` should be made public.*

Naming `make_cloned_ptr`

Prior art from `shared_ptr` and `unique_ptr` suggests that the `cloned_ptr` creating function should be called `make_cloned`. The authors prefer `make_cloned_ptr` as it follows prior art from `make_tuple`, `make_array` and `make_pair`. Where consistency is not an option, clarity is preferable.

The authors request the following straw poll:

- *The free function to create a `cloned_ptr` should be made called `make_cloned_ptr`.*
- *The free function to create a `cloned_ptr` should be made called `make_cloned`.*

Support for custom allocators, deleters and copiers

The reference implementation author has no implementation experience with adding custom allocators, deleters or copiers but anticipates that these would be welcome additions to `cloned_ptr`.

The authors request the following straw polls:

- *`cloned_ptr` should support a custom allocator.*

- `cloned_ptr` *should support a custom deleter.*
- `cloned_ptr` *should support a custom copier.*
- `cloned_ptr` *should support a combined custom copier/deleter.*

Target C++17 or Library Fundamentals TS 3

The authors request the following straw polls:

- `cloned_ptr` *should be added to the C++ Standard Library for C++17*
- `cloned_ptr` *should be added to a further Library Fundamentals TS*

Acknowledgements

The authors would like to thank Maciej Bogus, Germán Diago, Bengt Gustafsson, David Krauss, Nevin Leiber and Patrice Roy for useful discussion.

References

[N3339] "A Preliminary Proposal for a cloning-Copying Smart Pointer", W.E.Brown, 2012
`<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3339.pdf>`

[S.Parent] "C++ Seasoning", Sean Parent, 2013
`<https://github.com/sean-parent/sean-parent.github.io/wiki/Papers-and-Presentations>`

[Impl] Reference implementation: `cloned_ptr`, J.B.Coe
`<https://github.com/jbcoe/cloned_ptr>`