A Preliminary Proposal for a Deep-Copying Smart Pointer

1 Introduction and motivation

C++11 standardized the unique_ptr and shared_ptr smart pointer templates. One common use for these smart pointers is to serve as the pointer-to-implementation in the well-known pimpl idiom. However, when the application’s design calls for deep-copy semantics whenever the pimpl is copied, rather than moved or shared, neither of these smart pointers is the correct tool.

Similarly, none of the standard smart pointers is appropriate when deep-copying a pointee object whose type forms part of an inheritance hierarchy. Sometimes described as the polymorphic copying problem, its solution seems a viable candidate for standardization.

In these circumstances, among others, what is needed is a smart pointer that applies value semantics to its pointee. Therefore, this paper proposes a value_ptr smart pointer template for future C++ standardization. Prior art is discussed in the next section, then a specimen implementation is presented as a straw man for considering design issues. We conclude with a short list of open questions.

2 Prior art

The notion and utility of a smart pointer with embedded value semantics has been independently discovered, implemented, and discussed numerous times over the course of more than a decade. Representative examples (in publication order) include:

  Implements grin_ptr, “A template that looks after an object allocated on the heap, and ensures it is copied and deleted when appropriate.” Motivated in support of a “Cheshire Cat technique” (a forerunner of today’s pimpl idiom), grin_ptr delegates copying to a family of overloaded deep_copy functions.

Describes smart pointers with deep copy semantics as “vehicles for transporting polymorphic objects safely. You hold a smart pointer to a base class, which might actually point to a derived class. When you copy the smart pointer, you want to copy its polymorphic behavior, too. It’s interesting that you don’t exactly know what behavior and state you are dealing with, but you certainly need to duplicate that behavior and state.” Provides a policy class implementing such deep copying.


After explaining (in its preceding Item 30) why auto_ptr is unsuitable, derives ValuePtr as “a smart pointer class designed specifically for class membership.” The ultimate version provides “full traits-based customizability” for copying/cloning.


Provides clone_ptr and copy_ptr, asserting that “it helps solve problems associated with doing a deep copy of an abstract pointer.”


Starts a discussion thread. Proposes a non-pointer utility for manipulating dynamically-typed objects as “quite similar to a smart pointer that does deep-copying . . . .”


Implements value_ptr, a smart pointer with deep copy semantics to support inheritance hierarchies with value semantics. Uses metaprogramming to detect the presence of a member function named clone and employs it if found; else defaults to the copy c’tor.

## 3 A straw man implementation

While the following code is by no means an industrial-strength implementation, we present it as a preliminary specification of intent in order to serve as a basis for technical discussion.

```cpp
// value_ptr: A pointer treating its pointee as-if contained by value
// This smart pointer template mimics value semantics for its pointee:
// - the pointee lifetime matches the pointer lifetime, and
// - the pointee is copied whenever the pointer is copied.
// Having such a template provides a standard vocabulary to denote such
// pointers, with no need for further comment or other documentation to
describe the semantics involved.
// As a small bonus, this template’s c’tors ensure that all instance
// variables are initialized.
```
```cpp
#include <cstddef>  // nullptr_t
#include <functional>  // less
#include <memory>  // default_delete
#include <type_traits>  // add_pointer, ...
#include <utility>  // move, swap

namespace _ {
    template< class T > struct is_cloneable;

template< class Element
    , bool = std::is_polymorphic<Element>::value
    && _::is_cloneable<Element>::value
>
    struct default_action;

template< class Element 
    struct default_action<Element, false>;
} // _

template< class Element > struct default_copy;
template< class Element > struct default_clone;

template< class Element
    , class Cloner = _::default_action<Element>
    , class Deleter = std::default_delete<Element>
>
    class value_ptr;

template< class E, class C, class D >
void
    swap( value_ptr<E,C,D> & , value_ptr<E,C,D> & ) noexcept;

template< class E, class C, class D >
bool
    operator == ( value_ptr<E,C,D> const &, value_ptr<E,C,D> const & );

template< class E, class C, class D >
bool
    operator != ( value_ptr<E,C,D> const &, value_ptr<E,C,D> const & );

template< class E, class C, class D >
bool
    operator == ( value_ptr<E,C,D> const &, std::nullptr_t );

template< class E, class C, class D >
bool
    operator != ( value_ptr<E,C,D> const &, std::nullptr_t );

template< class E, class C, class D >
bool
    operator == ( std::nullptr_t, value_ptr<E,C,D> const & );

template< class E, class C, class D >
bool
    operator != ( std::nullptr_t, value_ptr<E,C,D> const & );

```
bool operator < ( value_ptr<E,C,D> const &, value_ptr<E,C,D> const & );

template< class E, class C, class D >
bool operator > ( value_ptr<E,C,D> const &, value_ptr<E,C,D> const & );

template< class E, class C, class D >
bool operator <= ( value_ptr<E,C,D> const &, value_ptr<E,C,D> const & );

template< class E, class C, class D >
bool operator >= ( value_ptr<E,C,D> const &, value_ptr<E,C,D> const & );

// ======================================================================

template< class T >
struct _::is_cloneable
{
    private:
        typedef char (& yes_t)[1];
        typedef char (& no_t)[2];

        template< class U, U* (U::*)() const = &U::clone > struct cloneable { };

        template< class U > static yes_t test( cloneable<U>* );
        template< class > static no_t test( ... );

    public:

        static bool const value = sizeof(test<T>(0)) == sizeof(yes_t);
}; // is_cloneable<>

// ======================================================================

template< class Element >
struct default_copy
{
    public:
        Element *
        operator () ( Element * p ) const { return new Element( *p ); }
    }; // default_copy<>

// ======================================================================

template< class Element >
struct default_clone
{
    public:
        Element *
        operator () ( Element * p ) const { return p->clone(); }
    }; // default_clone<>
template< class Element, bool >
struct _::default_action
  : public default_clone<Element>
{
public:
  using default_clone<Element>::operator();
}; // default_action<>

template< class Element >
struct _::default_action<Element, false>
  : public default_copy<Element>
{
public:
  using default_copy<Element>::operator();
}; // default_action<,false>

// -----------------------------------------------

template< class Element, class Cloner, class Deleter >
class value_ptr
{
public:
  // -- publish our template parameters and variations thereof:
  typedef Element element_type;
  typedef Cloner cloner_type;
  typedef Deleter deleter_type;
  typedef typename std::add_pointer<Element>::type pointer;
  typedef typename std::add_lvalue_reference<Element>::type reference;

private:
  template< class P >
  struct is_compatible
    : public std::is_convertible< typename std::add_pointer<P>::type, pointer >
  {
  };

public:
  // default c’tor:
  constexpr value_ptr() noexcept : p( nullptr ) { }

  // ownership-taking c’tors:
  constexpr value_ptr( std::nullptr_t ) noexcept : p( nullptr ) { }

  template< class E2 >
  explicit
  value_ptr( E2 * other ) noexcept
  : p( other )
  {
    static_assert( is_compatible<E2>::value,
      "value_ptr<>'s pointee type is incompatible!"
    );

    static_assert( ! std::is_polymorphic<E2>::value
|| ! (std::is_same< Cloner
179          , _::default_action<Element,false>
180          >::value)
181          , "value_ptr<>'s_pointee_type_would_slice_when_copying!"
182          );
183
184
185    // copying c'tors:
186    value_ptr( value_ptr const & other )
187    : p( clone_from(other.p) )
188    { }
189
190    template< class E2 >
191    value_ptr( value_ptr<E2,Cloner,Deleter> const & other
192          , typename std::enable_if< is_compatible<E2>::value
193          >::type * = 0
194          )
195    : p( clone_from(other.p) )
196    { }
197
198    // moving c'tors:
199    value_ptr( value_ptr && other ) noexcept
200    : p( other.release() )
201    { }
202
203    template< class E2 >
204    value_ptr( value_ptr<E2,Cloner,Deleter> && other
205          , typename std::enable_if< is_compatible<E2>::value
206          >::type * = 0
207          ) noexcept
208    : p( other.release() )
209    { }
210
211    // d'tor:
212    ~value_ptr( ) noexcept { reset(); }
213
214    // copying assignments:
215    value_ptr &
216    operator = ( std::nullptr_t ) noexcept
217    { reset( nullptr ); return *this; }
218
219    value_ptr &
220    operator = ( value_ptr const & other )
221    { value_ptr tmp(other); swap(tmp); return *this; }
222
223    template< class E2 >
224    typename std::enable_if< is_compatible<E2>::value, value_ptr & >::type
225    operator = ( value_ptr<E2,Cloner,Deleter> const & other )
226    { value_ptr tmp(other); swap(tmp); return *this; }
227
228    // moving assignments:
229    value_ptr &
230    operator = ( value_ptr && other )
231    { value_ptr tmp( std::move(other) ); swap(tmp); return *this; }
```cpp
template< class E2 >
typename std::enable_if< is_compatible<E2>::value, value_ptr & >::type
operator = ( value_ptr<E2,Cloner,Deleter> && other )
{ value_ptr tmp( std::move(other) ); swap(tmp); return *this; }

// observers:
reference
operator * ( ) const { return *get(); } // explicit
pointer
operator -> ( ) const noexcept { return get(); } // private:
pointer
get( ) const noexcept { return p; }

// modifiers:
pointer
release( ) noexcept { pointer old = p; p = nullptr; return old; }
void
reset( pointer t = pointer() ) noexcept { std::swap(p, t); Deleter()(t); }
void
swap( value_ptr & other ) noexcept { std::swap(p, other.p); }

private:
pointer p;

}; // value_ptr<>
operator == ( value_ptr<E,C,D> const & x, value_ptr<E,C,D> const & y )
{ return x.get() == y.get(); }

template< class E, class C, class D >
bool
operator != ( value_ptr<E,C,D> const & x, value_ptr<E,C,D> const & y )
{ return ! operator == (x, y); }

template< class E, class C, class D >
bool
operator == ( value_ptr<E,C,D> const & x, std::nullptr_t y )
{ return x.get() == y; }

template< class E, class C, class D >
bool
operator != ( value_ptr<E,C,D> const & x, std::nullptr_t y )
{ return ! operator == (x, y); }

template< class E, class C, class D >
bool
operator == ( std::nullptr_t x, value_ptr<E,C,D> const & y )
{ return x == y.get(); }

template< class E, class C, class D >
bool
operator != ( std::nullptr_t x, value_ptr<E,C,D> const & y )
{ return ! operator == (x, y); }

// -----------------------------------------------
// non-member ordering:

template< class E, class C, class D >
bool
operator < ( value_ptr<E,C,D> const & x, value_ptr<E,C,D> const & y )
{ typedef typename std::common_type< typename value_ptr<E,C,D>::pointer,
                                           typename value_ptr<E,C,D>::pointer>::type
     CT;
     return std::less<CT>()( x.get(), y.get() );
}

template< class E, class C, class D >
bool
operator > ( value_ptr<E,C,D> const & x, value_ptr<E,C,D> const & y )
{ return y < x; }

template< class E, class C, class D >
bool
operator <= ( value_ptr<E,C,D> const & x, value_ptr<E,C,D> const & y )
{ return ! (y < x); }

template< class E, class C, class D >
bool
4 Some open design questions

1. Should `value_ptr` be specialized to work with array types à la `unique_ptr`?

2. Should `value_ptr` take an allocator argument in addition to a cloner and a deleter? (Only the cloner would use the allocator.)

3. This implementation assumes that the cloner and deleter types are stateless; are these viable assumptions? If not, what policies should apply when they are being copied during a `value_ptr` copy?

4. With which, if any, standard smart pointers should this template innately interoperate, and to what degree?

5. What color should the bicycle shed be painted?

- `clone_ptr`
- `cloned_ptr`
- `cloning_ptr`
- `copycat_ptr`
- `copied_ptr`
- `copying_ptr`
- `deep_ptr`
- `dup_ptr`
- `duplicate_ptr`
- `duplicating_ptr`
- `deep_ptr`
- `matched_ptr`
- `matching_ptr`
- `replicating_ptr`
- `twin_ptr`
- `twinning_ptr`

5 Acknowledgments

Many thanks to the reviewers of early drafts of this paper for their helpful and constructive comments. We also acknowledge the Fermi National Accelerator Laboratory’s Computing Division, sponsor of our participation in the C++ standards effort, for its past and continuing support of our efforts to improve C++ for all our user communities.