A Proposal for the World's Dumbest Smart Pointer

Document #:	WG21/N3514
Date:	2012-12-19
Revises:	None
Project:	Programming Language C++
Reply to:	Walter E. Brown <webrown.cpp@gmail.com>

Contents

1 Introduction and motivation

C++11's **`shared_ptr`** and **`unique_ptr`** facilities, like C++98's **`auto_ptr`** before them, provide considerable expressive power for handling memory resources. In addition to the technical benefits of such *smart pointers*, their names provide *de facto* vocabulary types[1] for describing certain common coding idioms that encompass pointer-related policies such as pointee copying and lifetime management.

As another example, consider **`boost::optional`**,[2] which provides a pointer-like interface to access underlying (possibly uninitialized) values. Dave Abrahams characterizes[3] "the fundamental semantics of **`optional`** [as] identical to those of a (non-polymorphic) **`clone_ptr`**." Thus **`optional`** provides vocabulary for another common coding idiom in which bare pointers have been historically used.

Code that predates or otherwise avoids such smart pointers generally relies on C++'s native pointers for its memory management and allied needs, and so makes little or no coding use of any kind of standard descriptive vocabulary. As a result, it has often proven to be very challenging and time-consuming for a programmer to inspect code in order to discern the use to which any specific bare pointer is put, even if that use has no management role at all. As Loïc A. Joly observed,[4] "it is not easy to disambiguate a **`T*`** pointer that is only observing the data. . . . Even if

[1] Defined by Pablo Halperin in N1850 as "ubiquitous types used throughout the internal interfaces of a program." He goes on to say, "The use of a well-defined set of vocabulary types . . . lends simplicity and clarity to a piece of code."

[2] `http://www.boost.org/doc/libs/1_52_0/libs/optional/doc/html/index.html`. This functionality has been proposed for future standardization; see N3406.

[3] Reflector message c++std-lib-31692.

[4] Reflector message c++std-lib-31595.

it would just serve for documentation, having a dedicated type would have some value I think." Our experience leads us to agree with this assessment.

2 Alternative approaches

Responding to Joly's above-cited comment, Howard Hinnant presented[5] the following (lightly reformatted, excerpted) C++11 code to demonstrate one candidate mechanism for achieving Joly's objective:

```
struct do_nothing
{
  template< class T >
  void operator () ( T* ) { }; // do nothing
};

template< class T >
  using non_owning_ptr = unique_ptr<T, do_nothing>;
```

At first glance, this certainly seems a reasonable approach. However, on further reflection, the copy semantics of these **non_owning_ptr<>** types seem subtly wrong for non-owning pointers (*i.e.*, for pointers that behave strictly as observers): while the aliased underlying **unique_ptr** is (movable but) not copyable, we believe that an observer should be freely copyable to another observer object of the same or compatible type. Joly appears to concur with this view, stating[6] that "**non_owning_ptr** should be CopyConstructible and Assignable."

Later in the same thread, Howard Hinnant shared[7] his personal preference: "I use raw pointers for non-owning relationships. And I actually *like* them. And I don't find them difficult or error prone." While this assessment from an acknowledged expert (with concurrence from others[8]) is tempting, it seems most applicable when developing new code. However, we have found that a bare pointer is at such a low level of abstraction[9] that it can mean any one of quite a number of possibilities, especially when working with legacy code (*e.g.*, when trying to divine its intent or trying to interoperate with it).

Consistent with Bjarne Stroustrup's guideline[10] to "avoid very general types in interfaces," our coding standard has for some time strongly discouraged the use of bare pointers in most public interfaces.[11] However, it seems clear that there is and will continue to be a role for non-owning, observe-only pointers.

As Ville Voutilainen reminded us,[12] "we haven't standardized every useful smart pointer yet." We certainly agree; in our experience, it has proven helpful to have a standard vocabulary type with which to document the observe-only behavior via code that can also interoperate with bare

[5] Reflector message c++std-lib-31596.

[6] Reflector message c++std-lib-31725.

[7] Reflector message c++std-lib-31734.

[8] For example, Nevin Liber in c++std-lib-31729 expresses a related preference: "for non-owning situations use references where you can and pointers where you must. . . , and only use smart pointers when dealing with ownership." Other posters shared similar sentiments.

[9] It has been said that bare pointers are to data structures as **goto** is to control structures.

[10] See, for example, his keynote talk "C++11 Style" given 2012-02-02 during the *GoingNative 2012* event held in Redmond, WA, USA. Video and slides at http://channel9.msdn.com/Events/GoingNative/GoingNative-2012.

[11] Constructor parameters are a notable exception.

[12] Reflector message c++std-lib-31742.

pointers. The next section exhibits the essential aspects of **exempt_ptr**, our candidate for the (facetious yet descriptive) title of "World's Dumbest Smart Pointer."

3 A straw man implementation

We present the following code as a preliminary specification of intent in order to serve as a basis for technical discussion. Designed as a pointer that takes no formal notice of its pointee's lifetime, this not-very-smart pointer template is intended as a replacement for near-trivial uses of bare/native/raw/built-in/dumb C++ pointers, especially when used to communicate with (say) legacy code that traffics in such pointers. It is, by design, exempt (hence its working name) from any role in managing any pointee, and is thus freely copyable independent of and without regard for its pointee.

We have found that such a template provides us a standard vocabulary to denote non-owning pointers, with no need for further comment or other documentation to describe the near-vacuous semantics involved. As a small bonus, this template's c'tors ensure that all instance variables are initialized.

```
// ======================================================================
//
// exempt_ptr:  A pointer that is nearly oblivious to its pointee
//
// ======================================================================

#include <cstddef>      // nullptr_t
#include <functional>   // less
#include <type_traits>  // add_pointer, enable_if, ...
#include <utility>      // swap

// ======================================================================
// interface:

template< class E >
  class exempt_ptr;

template< class E >
void
  swap( exempt_ptr<E> &, exempt_ptr<E> & ) noexcept;

template< class E >
exempt_ptr<E>
  make_exempt( E * ) noexcept;

template< class E >
bool
  operator == ( exempt_ptr<E> const &, exempt_ptr<E> const & );
template< class E >
bool
  operator != ( exempt_ptr<E> const &, exempt_ptr<E> const & );

template< class E >
bool
  operator == ( exempt_ptr<E> const &, std::nullptr_t ) noexcept;
template< class E >
```

```
bool
  operator != ( exempt_ptr<E> const &, std::nullptr_t ) noexcept;

template< class E >
bool
  operator == ( std::nullptr_t, exempt_ptr<E> const & ) noexcept;
template< class E >
bool
  operator != ( std::nullptr_t, exempt_ptr<E> const & ) noexcept;

template< class E >
bool
  operator < ( exempt_ptr<E> const &, exempt_ptr<E> const & );
template< class E >
bool
  operator > ( exempt_ptr<E> const &, exempt_ptr<E> const & );
template< class E >
bool
  operator <= ( exempt_ptr<E> const &, exempt_ptr<E> const & );
template< class E >
bool
  operator >= ( exempt_ptr<E> const &, exempt_ptr<E> const & );

// ======================================================================
// implementation:

template< class E >
  class exempt_ptr
{
public:
  // publish our template parameter and variations thereof:
  using element_type = E;
  using pointer      = typename std::add_pointer<E>::type;
  using reference    = typename std::add_lvalue_reference<E>::type;

private:
  template< class P >
  constexpr  bool
    is_compat( )
  {
    return std::is_convertible< typename std::add_pointer<P>::type
                              , pointer
                              >::value;
  }

public:
  // default c'tor:
  constexpr  exempt_ptr( ) noexcept : p{ nullptr }  { }

  // pointer-accepting c'tors:
  constexpr  exempt_ptr( std::nullptr_t ) noexcept : exempt_ptr{}  { }
  explicit   exempt_ptr( pointer other  ) noexcept : p{ other   }  { }
  template< class E2
          , class = typename std::enable_if< is_compat<E2>() >::type
          >
  explicit   exempt_ptr( E2 * other ) noexcept
  : p( other )
  { }

  // copying c'tors (in addition to compiler-generated copy c'tor):
  template< class E2
          , class = typename std::enable_if< is_compat<E2>() >::type
          >
  exempt_ptr( exempt_ptr<E2> const & other ) noexcept
  : p{ other.get() }
  { }

  // pointer-accepting assignments:
  exempt_ptr &
    operator = ( std::nullptr_t ) noexcept
  { reset(nullptr); return *this; }
  template< class E2 >
  typename std::enable_if< is_compat<E2>(), exempt_ptr & >::type
    operator = ( E2 * other ) noexcept
  { reset(other); return *this; }

  // copying assignments (in addition to compiler-generated copy assignment):
  template< class E2 >
  typename std::enable_if< is_compat<E2>(), exempt_ptr & >::type
    operator = ( exempt_ptr<E2> const & other ) noexcept
  { reset(other.get()); return *this; }

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

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

private:
  pointer  p;

};  // exempt_ptr<>

// ----------------------------------------------
// non-member swap:

template< class E >
inline void
  swap( exempt_ptr<E> & x, exempt_ptr<E> & y ) noexcept
{ x.swap(y); }

// ----------------------------------------------
// non-member make_exempt:
```

```
template< class E >
exempt_ptr<E>
  make_exempt( E * p ) noexcept
{ return exempt_ptr<E>{p}; }

// ----------------------------------------------
// non-member (in)equality comparison:

template< class E >
bool
  operator == ( exempt_ptr<E> const & x, exempt_ptr<E> const & y )
{ return x.get() == y.get(); }

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

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

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

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

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

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

template< class E >
bool
  operator < ( exempt_ptr<E> const & x, exempt_ptr<E> const & y )
{
  using  PTR = typename exempt_ptr<E>::pointer;
  return std::less<PTR>()(x.get(), y.get());
}

template< class E >
bool
  operator > ( exempt_ptr<E> const & x, exempt_ptr<E> const & y )
{ return y < x; }
```

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

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

4 Some open questions

1. At the moment, **`exempt_ptr`** knows of no other smart pointer. Should **`exempt_ptr`** innately interoperate with any of the standard smart pointers? If so, with which one(s) and to what degree? More generally, can WG21 articulate a *smart pointer interoperability* policy or rationale in order to guide us in such decisions?

2. Even though there is a **`get()`** member, would a conversion operator to the underlying pointer type (*i.e.*, **`exempt_ptr<E>::operator E*()`**) provide any significant user benefit? If so, should the operator be **`explicit`**?

3. To what, if any, degree should **`exempt_ptr`** cater to array types **`T[]`**? For example, should we provide a complete specialization, or would it suffice to provide a member **`operator[]`**?

4. Alternative names[13] (shown alphabetically) for bike-shed consideration:

- **aloof_ptr**
- **agnostic_ptr**
- **apolitical_ptr**
- **ascetic_ptr**
- **attending_ptr**
- **austere_ptr**
- **bare_ptr**
- **blameless_ptr**
- **blond_ptr**
- **blonde_ptr**
- **classic_ptr**
- **core_ptr**
- **disinterested_ptr**
- **disowned_ptr**
- **disowning_ptr**
- **dumb_ptr**
- **emancipated_ptr**
- **estranged_ptr**
- **excused_ptr**
- **faultless_ptr**
- **free_ptr**
- **freeagent_ptr**
- **guiltless_ptr**
- **handsoff_ptr**
- **ignorant_ptr**
- **impartial_ptr**
- **independent_ptr**
- **innocent_ptr**
- **irresponsible_ptr**
- **just_a_ptr**
- **legacy_ptr**
- **naked_ptr**
- **neutral_ptr**
- **nonown_ptr**
- **nonowning_ptr**
- **notme_ptr**
- **oblivious_ptr**
- **observer_ptr**
- **observing_ptr**
- **open_ptr**
- **ownerless_ptr**
- **pointer**
- **ptr**
- **pure_ptr**
- **quintessential_ptr**
- **severe_ptr**
- **simple_ptr**
- **stark_ptr**
- **straight_ptr**
- **true_ptr**
- **unfettered_ptr**
- **uninvolved_ptr**
- **unmanaged_ptr**
- **unowned_ptr**
- **untainted_ptr**
- **unyoked_ptr**
- **virgin_ptr**
- **visiting_ptr**
- **watch_ptr**
- **watcher_ptr**
- **watching_ptr**
- **witless_ptr**
- **witness_ptr**

[13] Most of these names were suggested by readers of earlier drafts. While not all suggestions seem viable (some are clearly intended humorously), we have opted to preserve all of them for the record.

5 Proposed wording: synopsis

Append the following to [memory.syn]:

```
namespace std {
// 20.7.x, class template exempt_ptr
template <class E> class exempt_ptr;

template <class E>
  void swap(exempt_ptr<E> &, exempt_ptr<E> &) noexcept;

template <class E>
  exempt_ptr<E> make_exempt(E *) noexcept;

template <class E1, class E2>
  bool operator==(exempt_ptr<E1> const &, exempt_ptr<E2> const &);
template <class E1, class E2>
  bool operator!=(exempt_ptr<E1> const &, exempt_ptr<E2> const &);

template <class E>
  bool operator==(exempt_ptr<E> const &, nullptr_t) noexcept;
template <class E>
  bool operator!=(exempt_ptr<E> const &, nullptr_t) noexcept;

template <class E>
  bool operator==(nullptr_t, exempt_ptr<E> const &) noexcept;
template <class E>
  bool operator!=(nullptr_t, exempt_ptr<E> const &) noexcept;

template <class E1, class E2>
  bool operator<(exempt_ptr<E1> const &, exempt_ptr<E2> const &);
template <class E1, class E2>
  bool operator>(exempt_ptr<E1> const &, exempt_ptr<E2> const &);
template <class E1, class E2>
  bool operator<=(exempt_ptr<E1> const &, exempt_ptr<E2> const &);
template <class E1, class E2>
  bool operator>=(exempt_ptr<E1> const &, exempt_ptr<E2> const &);
}
```

6 Proposed wording: class template, *etc.*

Create in [smartptr] a new subclause as follows:

20.7.x Non-owning pointers

1 A *non-owning pointer*, also known as an *observer* or *watcher*, is an object *o* that stores a pointer to a second object *w*. In this context, *w* is known as a *watched* object. [*Note:* There is no watched object when the stored pointer is **nullptr**. — *end note*] An observer takes no

responsibility or ownership of any kind for the watched object, if any. In particular, there is no predetermined relationship between the lifetimes of any observer and any watched objects.

2 Each type instantiated from the **`exempt_ptr`** template specified in this subclause shall meet the requirements of a **`CopyConstructible`** and **`CopyAssignable`** type. The template parameter **`E`** of **`exempt_ptr`** may be an incomplete type.

3 [*Note:* The uses of **`exempt_ptr`** include clarity of interface specification in new code, and interoperability with pointer-based legacy code. — *end note*]

Following the practice of C++11, another copy of the synopsis above is to be inserted here. However, comments are omitted from this copy.

20.7.x.1 Class template **`exempt_ptr`** [exempt.ptr]

1 For the purposes of this subclause, a type **`F`** is said to be *pointer-incompatible* with a type **`E`** if the expression **`is_convertible< typename add_pointer<F>::type, typename add_pointer< E>::type >::value`** is **`false`**.

```
namespace std {
template <class E> class exempt_ptr {
public:
  // -- publish our template parameter and variations thereof:
  using element_type = E;
  using pointer      = typename add_pointer<E>::type;
  using reference    = typename add_lvalue_reference<E>::type;

  // -- default c'tor:
  constexpr  exempt_ptr() noexcept;

  // pointer-accepting c'tors:
  constexpr exempt_ptr(nullptr_t) noexcept : exempt_ptr() {}
  explicit exempt_ptr(pointer other) noexcept;
  template <class E2> explicit exempt_ptr(E2 * other) noexcept;

  // copying c'tors (in addition to compiler-generated copy c'tor):
  template <class E2> exempt_ptr(exempt_ptr<E2> const & other) noexcept;

  // pointer-accepting assignments:
  exempt_ptr & operator=(nullptr_t) noexcept;
  template< class E2 > exempt_ptr & operator=(E2 * other) noexcept;

  // copying assignments (in addition to compiler-generated copy assignment):
  template< class E2 >
    exempt_ptr & operator=(exempt_ptr<E2> const & other) noexcept;

  // observers:
  pointer get() const noexcept;
  reference operator*() const noexcept;
  pointer operator->() const noexcept;
  explicit operator bool() const noexcept;

  // modifiers:
  pointer release() noexcept;
  void reset(pointer t = nullptr) noexcept;
  void swap(exempt_ptr & other) noexcept;

};  // exempt_ptr<>
}
```

20.7.x.1.1 **exempt_ptr** constructors [exempt.ptr.ctor]

```
constexpr exempt_ptr() noexcept;
```

1 *Effects:* Constructs an **exempt_ptr** object that has no corresponding watched object.

2 *Postcondition:* **get() == nullptr**.

```
explicit exempt_ptr(pointer other) noexcept;
```

3 *Effects:* Constructs an **exempt_ptr** object whose watched object is ***other**.

```
template <class E2> explicit exempt_ptr(E2 * other) noexcept;
```

4 *Effects:* Constructs an **exempt_ptr** object whose watched object is ***dynamic_cast<pointer>(other)**.

5 *Remarks:* This constructor shall not participate in overload resolution if **E2** is pointer-incompatible with **E**.

```
template <class E2> exempt_ptr(exempt_ptr<E2> const & other) noexcept;
```

6 *Effects:* Constructs an **exempt_ptr** object whose watched object is ***dynamic_cast<pointer>(other)**.

7 *Remarks:* This constructor shall not participate in overload resolution if **E2** is pointer-incompatible with **E**.

20.7.x.1.2 **exempt_ptr** assignment [exempt.ptr.assign]

```
exempt_ptr & operator=(nullptr_t) noexcept;
```

1 *Effects:* Same as if calling **reset(nullptr);**

2 *Returns:* ***this**.

```
template <class E2> exempt_ptr & operator=(E2 * other) noexcept;
```

3 *Effects:* Same as if calling **reset(other);**

4 *Returns:* ***this**.

5 *Remarks:* This operator shall not participate in overload resolution if **E2** is pointer-incompatible with **E**.

```
template< class E2 >
  exempt_ptr & operator=(exempt_ptr<E2> const & other) noexcept;
```

6 *Effects:* Same as if calling **reset(other.get());**

7 *Returns:* ***this**.

8 *Remarks:* This operator shall not participate in overload resolution if **E2** is pointer-incompatible with **E**.

20.7.x.1.3 **exempt_ptr** observers [exempt.ptr.obs]

```
pointer get() const noexcept;
```

1 *Returns:* The stored pointer.

```
reference operator*() const noexcept;
```

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

3 *Returns:* **`*get()`**.

```
pointer operator->() const noexcept;
```

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

5 *Returns:* **`get()`**.

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

6 *Returns:* **`get() != nullptr`**.

20.7.x.1.4 **`exempt_ptr`** modifiers [exempt.ptr.mod]

```
pointer release() noexcept;
```

1 *Postcondition:* **`get() == nullptr`**.

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

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

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

```
void swap(exempt_ptr & other) noexcept;
```

4 *Effects:* Invokes **`swap`** on the stored pointers of **`*this`** and **`other`**.

20.7.x.1.5 **`exempt_ptr`** specialized algorithms [exempt.ptr.special]

```
template <class E>
  void swap(exempt_ptr<E> & p1, exempt_ptr<E> & p2) noexcept;
```

1 *Effects:* Calls **`p1.swap(p2)`**.

```
template <class E> exempt_ptr<E> make_exempt(E * p) noexcept;
```

2 *Returns:* **`exempt_ptr<E>{p}`**.

```
template <class E1, class E2>
  bool operator==(exempt_ptr<E1> const & p1, exempt_ptr<E2> const & p2);
```

3 *Returns:* **`p1.get() == p2.get()`**.

```
template <class E1, class E2>
  bool operator!=(exempt_ptr<E1> const & p1, exempt_ptr<E2> const & p2);
```

4 *Returns:* **`p1.get() != p2.get()`**.

```
template <class E>
  bool operator==(exempt_ptr<E> const & p, nullptr_t) noexcept;
template <class E>
  bool operator==(nullptr_t, exempt_ptr<E> const & p) noexcept;
```

5 *Returns:* **`!p`**.

```
template <class E>
  bool operator!=(exempt_ptr<E> const & p, nullptr_t) noexcept;
template <class E>
  bool operator!=(nullptr_t, exempt_ptr<E> const & p) noexcept;
```

6 *Returns:* **`(bool)p`**.

```
template <class E1, class E2>
  bool operator<(exempt_ptr<E1> const & p1, exempt_ptr<E2> const & p2);
```

7 *Requires:* Let **`CT`** be **`common_type< exempt_ptr<E1>::pointer, exempt_ptr<E2>::pointer >::type`**. Then the specialization **`less<CT>`** shall be a function object type (20.8) that induces a strict weak ordering (25.4) on the pointer values.

8 *Returns:* **`less<CT>()(p1.get(), p2.get())`**.

```
template <class E>
  bool operator>(exempt_ptr<E> const & p1, exempt_ptr<E> const & p2);
```

9 *Returns:* **`p2 < p1`**.

```
template <class E>
  bool operator<=(exempt_ptr<E> const & p1, exempt_ptr<E> const & p2);
```

10 *Returns:* **`!(p2 < p1)`**.

```
template <class E>
  bool operator>=(exempt_ptr<E> const & p1, exempt_ptr<E> const & p2);
```

11 *Returns:* **`!(p1 < p2)`**.

7 Acknowledgments

Many thanks to the reviewers of early drafts of this paper for their helpful and constructive comments.