A Proposal for the World's Dumbest Smart Pointer, v3

Document #: WG21 N3840
Date: 2014-01-01
Revises: N3740
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes **`observer_ptr`**, a (not very) smart pointer type that takes no ownership responsibility for its pointees, i.e., for the objects it observes. As such, it is intended as a near drop-in replacement for raw pointer types, with the advantage that, as a vocabulary type, it indicates its intended use without need for detailed analysis by code readers.

I'm an observer in these matters, not a participant.
— MARSHALL MCLUHAN

The universe as we know it is a joint product of the observer and the observed.
— PIERRE TEILHARD DE CHARDIN

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.

Copyright © 2014 by Walter E. Brown. All rights reserved.

[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. See also the accepted-but-later-rescinded C++14 proposal N3672 by Fernando Cacciola and Andrzej Krzemieński.

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

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 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.

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

[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.

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

3 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 former working name, **exempt_ptr**) 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.

4 Proposed wording[13]

4.1 Synopsis

Append the following, in namespace **std**, to [memory.syn]:

```
// 20.8.x, class template observer_ptr
template <class W> class observer_ptr;
template <class W, size_t N> class observer_ptr<E[N]>;

template <class W>
  void swap(const observer_ptr<W>&, const observer_ptr<W>&) noexcept;
template <class W>
  observer_ptr<W> make_observer(W*) noexcept;

// (in)equality operators
template <class W1, class W2>
  bool operator==(const observer_ptr<W1>&, const observer_ptr<W2>&);
template <class W1, class W2>
  bool operator!=(const observer_ptr<W1>&, const observer_ptr<W2>&);

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

template <class W>
  bool operator==(nullptr_t, const observer_ptr<W>&) noexcept;
template <class W>
```

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

[13] All proposed additions and ~~deletions~~ are relative to the post-Chicago Working Draft [N3797]. Editorial notes are displayed against a gray background.

```
  bool operator!=(nullptr_t, const observer_ptr<W>&) noexcept;

// ordering operators
template <class W1, class W2>
  bool operator<(const observer_ptr<W1>&, const observer_ptr<W2>&);
template <class W1, class W2>
  bool operator>(const observer_ptr<W1>&, const observer_ptr<W2>&);
template <class W1, class W2>
  bool operator<=(const observer_ptr<W1>&, const observer_ptr<W2>&);
template <class W1, class W2>
  bool operator>=(const observer_ptr<W1>&, const observer_ptr<W2>&);

// arithmetic operators
template <class W, size_t N>
  observer_ptr<W> operator+(ptrdiff_t, const observer_ptr<W>[N]&);

// hash support
template <class T> struct hash<observer_ptr<T>>;
```

4.2 Class template, etc.

Create in [smartptr] a new subclause as follows:

20.8.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 inherent relationship between the lifetimes of any observer and any watched objects.

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

3 [*Note:* The uses of **observer_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.

20.8.x.1 Class template **observer_ptr** [observer.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< add_pointer_t<F>, add_pointer_t<E> >::value** is **false**.

```
namespace std {
template <class W> class observer_ptr {
public:
  // publish our template parameter and variations thereof
  using value_type      = W;
  using pointer         = add_pointer_t<W>;
  using const_pointer   = add_pointer_t<const W>;
  using reference       = add_lvalue_reference_t<W>;
  using const_reference = add_lvalue_reference_t<const W>;
  // default c'tor
  constexpr observer_ptr() noexcept;

  // pointer-accepting c'tors
  constexpr observer_ptr(nullptr_t) noexcept;
  explicit observer_ptr(pointer) noexcept;
  template <class W2> explicit observer_ptr(W2*) noexcept;

  // copying c'tors (in addition to compiler-generated copy c'tor)
  template <class W2> observer_ptr(const observer_ptr<W2>&) noexcept;

  // pointer-accepting assignments
  observer_ptr& operator=(nullptr_t) noexcept;
  template <class W2> observer_ptr& operator=(W2* other) noexcept;

  // copying assignments (in addition to compiler-generated copy assignment)
  template <class W2> observer_ptr& operator=(const observer_ptr<W2>&) noexcept;

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

  // conversions
  explicit operator pointer() noexcept;
  explicit operator const_pointer() const noexcept;

  // modifiers
  pointer release() noexcept;
  void reset(pointer t = nullptr) noexcept;
  void swap(observer_ptr&) noexcept;
};  // observer_ptr<>

template <class W, size_t N> class observer_ptr<W[N]> {
public:
  // publish our template parameter and variations thereof
  using value_type      = W;
  using pointer         = add_pointer_t<W>;
  using const_pointer   = add_pointer_t<const W>;
  using reference       = add_lvalue_reference_t<W>;
  using const_reference = add_lvalue_reference_t<const W>;

  // enable use as a pointer-like iterator
  using difference_type   = ptrdiff_t;
  using iterator_category = random_access_iterator_tag;

  // default c'tor
  constexpr observer_ptr() noexcept;

  // pointer-accepting c'tors
  constexpr observer_ptr(nullptr_t) noexcept;
  explicit observer_ptr(pointer) noexcept;
  template <class W2> explicit observer_ptr(W2*) noexcept;
  // copying c'tors (in addition to compiler-generated copy c'tor)
  template <class W2> observer_ptr(const observer_ptr<W2>&) noexcept;

  // pointer-accepting assignments
  observer_ptr& operator=(nullptr_t) noexcept;
  template <class W2> observer_ptr& operator=(W2* other) noexcept;

  // copying assignments (in addition to compiler-generated copy assignment)
  template <class W2> observer_ptr& operator=(const observer_ptr<W2>&) noexcept;

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

  // conversions
  explicit operator pointer() noexcept;
  explicit operator const_pointer() const noexcept;
  explicit operator observer_ptr() noexcept;
  explicit operator const observer_ptr() const noexcept;

  // modifiers
  pointer release() noexcept;
  void reset(pointer t = nullptr) noexcept;
  void swap(observer_ptr&) noexcept;

  // arithmetic
  observer_ptr& operator++();
  observer_ptr& operator--();
  observer_ptr operator++(int);
  observer_ptr operator--(int);
  observer_ptr operator+() const;
  observer_ptr operator+(ptrdiff_t) const;
  observer_ptr operator-(ptrdiff_t) const;
  ptrdiff_t  operator-(const observer_ptr&);
  value_type& operator[](ptrdiff_t);
  const value_type& operator[](ptrdiff_t) const;
};  // observer_ptr<W[N]>
}
```

20.8.x.1.1 **observer_ptr** constructors [observer.ptr.ctor]

```
constexpr observer_ptr() noexcept;
constexpr observer_ptr(nullptr_t) noexcept;
```

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

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

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

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

```
template <class W2> explicit observer_ptr(W2* other) noexcept;
```

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

5 *Remarks:* This constructor shall not participate in overload resolution if **W2** is pointer-incompatible with **W**.

```
template <class W2> observer_ptr(const observer_ptr<W2>& other) noexcept;
```

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

7 *Remarks:* This constructor shall not participate in overload resolution if **W2** is pointer-incompatible with **W**.

20.8.x.1.2 **observer_ptr** assignment [observer.ptr.assign]

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

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

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

```
template <class W2> observer_ptr& operator=(W2* other) noexcept;
```

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

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

5 *Remarks:* This operator shall not participate in overload resolution if **W2** is pointer-incompatible with **W**.

```
template <class W2> observer_ptr& operator=(const observer_ptr<W2>& 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 **W2** is pointer-incompatible with **W**.

20.8.x.1.3 **observer_ptr** observers [observer.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**.

20.8.x.1.4 **observer_ptr** conversions [observer.ptr.conv]

```
explicit operator pointer() noexcept;
```

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

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

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

20.8.x.1.5 **`observer_ptr`** modifiers [observer.ptr.mod]

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

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

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

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

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

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

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

20.8.x.1.6 **`observer_ptr`** specialized algorithms [observer.ptr.special]

```
template <class W>
  void swap(observer_ptr<W> & p1, observer_ptr<W> & p2) noexcept;
```

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

```
template <class W> observer_ptr<W> make_observer(W * p) noexcept;
```

2 *Returns:* **`observer_ptr<W>{p}`**.

```
template <class W1, class W2>
  bool operator==(observer_ptr<W1> const & p1, observer_ptr<W2> const & p2);
```

3 *Returns:* **`equal_to<void>()(p1.get(), p2.get())`**.

```
template <class W1, class W2>
  bool operator!=(observer_ptr<W1> const & p1, observer_ptr<W2> const & p2);
```

4 *Returns:* **`not operator==(p1, p2)`**.

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

5 *Returns:* **`not p`**.

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

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

```
template <class W1, class W2>
  bool operator<(observer_ptr<W1> const & p1, observer_ptr<W2> const & p2);
```

7 *Returns:* **`less<void>()(x.get(), y.get());`**.

```
template <class W>
  bool operator>(observer_ptr<W> const & p1, observer_ptr<W> const & p2);
```

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

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

9 *Returns:* **`not (p2 < p1)`**.

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

10 *Returns:* **`not (p1 < p2)`**.

20.8.x.2 Class template specialization **`observer_ptr<W[N]>`** [observer.arr]

1 The members of this template specialization constitute a superset of the members of the primary template. Descriptions are provided here only for members that are not described in the primary template or for members where there is additional semantic information.

20.8.x.2.1 **`observer_ptr`** array conversions [observer.arr.conv]

```
explicit operator observer_ptr<W>() noexcept;
```

1 *Returns:* **`observer_ptr<W>(get())`**.

```
explicit operator observer_ptr<W> const() const noexcept;
```

2 *Returns:* **`observer_ptr<W>(get())`**.

20.8.x.2.2 **`observer_ptr`** array arithmetic [observer.arr.arith]

```
observer_ptr& operator++();
observer_ptr& operator--();
```

1 *Effects:* Increments (or, in the second form, decrements) the stored pointer.

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

3 *Returns:* **`*this`**.

```
observer_ptr operator++(int);
observer_ptr operator--(int);
```

4 *Effects:* Increments (or, in the second form, decrements) the stored pointer.

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

6 *Returns:* **`observer_ptr(p.get()++)`** or **`observer_ptr(p.get()--)`**, respectively.

```
observer_ptr operator+() const;
```

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

```
ptrdiff_t operator-(const observer_ptr& p2)
```

8 *Requires:* **`get() != nullptr`** and **`p2.get() != nullptr`**.

9 *Returns:* **`get() - p2.get()`**

```
observer_ptr operator+(ptrdiff_t d) const;
observer_ptr operator-(ptrdiff_t d) const;
```

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

11 *Returns:* **`observer_ptr(p.get()++)`** or **`observer_ptr(p.get()--)`**, respectively.

```
value_type& operator[](ptrdiff_t k);
const value_type& operator[](ptrdiff_t k);
```

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

13 *Returns:* **`observer_ptr(get()[k])`**.

20.8.x.2.3 Other **`observer_ptr`** array arithmetic [observer.free.arith]

```
template< class W, size_t N >
observer_ptr<W> operator+(ptrdiff_t d, observer_ptr<const W[N]>& p)
```

1 *Requires:* **`p.get() != nullptr`**.

2 *Returns:* **`observer_ptr(p.get() + d)`**.

Append the following new paragraph to [util.smartptr.hash] (20.8.2.6) "Smart pointer hash support"; the proposed wording is a direct analog of existing wording for **`shared_ptr`**. (Note to Project Editor: the code snippets throughout the current subclause carefully, but unnecessarily since C++11, avoid adjacent closing angle brackets.)

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

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

5 Feature-testing macro

For the purposes of SG10, we recommend the macro name **`__cpp_lib_observer_ptr`**.

6 Acknowledgments

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

7 Bibliography

[N3797] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3797 (post-Chicago mailing), 2012-11-02.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

8 Revision history

Version	Date	Changes
1	2012-12-19	• Published as N3514.

2	2013-08-30	• Added this "Revision history" section. • Augmented the proposal with conversion and arithmetic operators. • Removed two no-longer-open questions. • Consolidated all proposed wording into a single section. • Updated code for C++14 compliance. • Reflected C++14 status of **`optional`** proposal. • Augmented the proposal with additional member typedefs. • Published as N3740.
3	2014-01-01	• Renamed **`exempt_ptr`** to **`observer_ptr`**. • Revised further per LEWG consensus at 2013 Chicago. • Adjusted to reflect rescinded C++14 status of **`optional`** proposal. • Added abstract, epigram, bibliography, and hash support. • Excised draft implementation and bikeshed questions. • Published as N3840.