A Proposal for the World’s Dumbest Smart Pointer, v4

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

Sometimes it is the quiet observer who sees the most.
— KATHRYN L. NELSON

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

Being a good observer is a great tool to have….
— LAUREN CONRAD

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.

\(^1\) 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.”
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 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:

```cpp
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).

---

\(^2\)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-31595.

\(^4\)Reflector message c++std-lib-31692.

\(^5\)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.
Consistent with Bjarne Stroustrup’s guideline\textsuperscript{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.\textsuperscript{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,\textsuperscript{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 \texttt{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 original working name, \texttt{exempt\_ptr}) from any role in managing any pointee, and is thus freely copyable independent of and without regard for its pointee.

It is a design feature that the conversion functions to and from bare pointers are \texttt{explicit}. The intent is to help users realize that they should carefully consider their pointee’s ownership, even when they intend only to observe, and hence not to participate in a pointee’s (\textit{watched object}’s ownership).

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\textsuperscript{13}

#### 4.1 Synopsis

Append the following, in namespace \texttt{std}, to [memory.syn]:

```cpp
// 20.8.x, class template observer\_ptr
template <class W> class observer\_ptr;

template <class W>
    void swap(observer\_ptr<W>&, observer\_ptr<W>&) noexcept;

// (in)equality operators
template <class W1, class W2>
    bool operator==(observer\_ptr<W1>, observer\_ptr<W2>);
```

\textsuperscript{10}See, for example, his keynote talk “C++11 Style” given 2012-02-02 during the \textit{GoingNative 2012} event held in Redmond, WA, USA. Video and slides at http://channel9.msdn.com/Events/GoingNative/GoingNative-2012.

\textsuperscript{11}Constructor parameters are a notable exception.

\textsuperscript{12}Reflector message c++std-lib-31742.

\textsuperscript{13}All proposed \texttt{additions} and \texttt{deletions} are relative to the pre-Urbana Working Draft \texttt{[N4140]}. Editorial notes are displayed against a \textcolor{gray}{gray} background.
template <class W1, class W2>
bool operator!=(observer_ptr<W1>, observer_ptr<W2>);

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

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

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

// 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, known as an observer, 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 its watched object, if any; in particular, there is no inherent relationship between the lifetimes of \( o \) and \( w \).

2 Specializations of observer_ptr shall meet the requirements of a CopyConstructible and CopyAssignable type. The template parameter \( W \) of an observer_ptr shall not be a reference type, but 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.
namespace std {

template <class W> class observer_ptr {
    public:
        using element_type = W;
        using pointer = add_pointer_t<W>;
        using reference = add_lvalue_reference_t<W>;

        constexpr observer_ptr() noexcept; // default c’tor
        constexpr observer_ptr(nullptr_t) noexcept;
        constexpr explicit observer_ptr(pointer) noexcept;

        template <class W2> constexpr observer_ptr(observer_ptr<W2>) noexcept;

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

        explicit operator pointer() const noexcept;

        pointer release() noexcept;
        void reset(pointer = nullptr) noexcept;
        void swap(observer_ptr&) noexcept;
    }

20.8.x.1.1 observer_ptr constructors

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

1 \textit{Effects}: Constructs an \texttt{observer ptr} object that has no corresponding watched object.

2 \textit{Postconditions}: \texttt{get() == nullptr}.

constexpr explicit observer_ptr(pointer other) noexcept;

3 \textit{Postconditions}: \texttt{get() == other}.

template <class W2> constexpr observer_ptr(observer_ptr<W2> other) noexcept;

4 \textit{Postconditions}: \texttt{get() == other.get()}.

5 \textit{Remarks}: This constructor shall not participate in overload resolution unless \texttt{W2\*} is convertible to \texttt{W\*}.

20.8.x.1.3 observer_ptr observers
constexpr pointer get() const noexcept;

1 Returns: The stored pointer.

constexpr reference operator*() const;

2 Requires: get() != nullptr.
3 Returns: *get().
4 Throws: Nothing.

constexpr pointer operator->() const noexcept;

5 Returns: get().

constexpr explicit operator bool() const noexcept;

6 Returns: get() != nullptr.

20.8.x.1.4 observer_ptr conversions
constexpr explicit operator pointer() const noexcept;

1 Returns: get().

20.8.x.1.5 observer_ptr modifiers
constexpr pointer release() noexcept;

1 Postconditions: get() == nullptr.
2 Returns: The value get() had at the start of the call to release.

constexpr void reset(pointer p = nullptr) noexcept;

3 Postconditions: get() == p.

constexpr 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

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> p1, observer_ptr<W2> p2);

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

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

4 Returns: not (p1 == p2).
template <class W>
  bool operator==(observer_ptr<W> p, nullptr_t) noexcept;

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

5 Returns: not p.

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

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

6 Returns: (bool)p.

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

7 Returns: less<W3>()(p1.get(), p2.get()), where W3 is the composite pointer type (Clause 5) of W1* and W2*.

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

8 Returns: p2 < p1.

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

9 Returns: not (p2 < p1).

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

10 Returns: not (p1 < p2).

Append the following new paragraph to [util.smartptr.hash] “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


8 Revision history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012-12-19</td>
<td>● Published as N3514.</td>
</tr>
<tr>
<td>2</td>
<td>2013-08-30</td>
<td>● 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.</td>
</tr>
<tr>
<td>3</td>
<td>2014-01-01</td>
<td>● 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, epigraph, bibliography, and hash support. ● Excised draft implementation and bikeshed questions. ● Published as N3840.</td>
</tr>
<tr>
<td>4</td>
<td>2014-11-07</td>
<td>● Updated to latest Working Draft. ● Added a paragraph to “Discussion” section re use of explicit. ● Revised “Proposed wording” section per LWG review at 2014 Urbana. ● Increased the epigraph count. ● Published as N4282.</td>
</tr>
</tbody>
</table>