Doc. No. P1408R0
Date: 2018-01-04
Programming Language C++
Audience LEWG and EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

Abandon observer_ptr

Bjarne Stroustrup

Summary

I propose to drop **std::observer_ptr** from Working Draft, C++ Extensions for Library Fundamentals, Version 3. It was adopted from A Proposal for the World's Dumbest Smart Pointer, v4 in 2014 after a discussion that was remarkably short for a proposal that aims to change every C++ program in existence. I find that paper's rationale and discussion of alternatives very brief and rather weak. To quote Walter Brown from the minutes of the 2013 Urbana-Champaign meeting:

> **Objective: get rid of bare pointers from user code.**

I don't agree with that objective and I don't think it would be feasible to do even if I did agree. That is not the only view of **observer_ptr**, though (see "benefits" below) and the minutes may not have caught Walter's statements accurately. However, that's a view I have heard expressed many times in various contexts and that makes me wary.

Pointers are really good at pointing to "things" and **T*** is a really good notation for that, much nicer than the verbose **std::observer_ptr<T>**. What pointers are not good at is representing ownership and directly support safe iteration.

Problems

- We have 50 years of experience with the **T*** notation - whatever we do will not make **T*** go away (in C++ or C). For starters there are billions of lines of code using that notation and 50 years of books, articles, documentation, tutorials, blogs, etc. in current use.
- The conventional **T***s notation is far simpler than **std::observer_ptr<T>**.
- In many code bases, **T***s and **std::observer_ptr<T>**s would co-exist for a long time ("forever"). However, they are not completely interchangeable (e.g., will **p=q** work? I'll have to look at the definitions of **p** and **q** to know).
- Many uses of **T*** are in C-style interfaces, using a class, such as **std::observer_ptr<T>** will cause compatibility problems and/or ABI breakage.
- Using a template, such as **std::observer_ptr<T>** will slow down compilation and complicate some forms of debugging. The use of **std::observer_ptr<T>** will not slow compilations down

much compared to **T*** but it is part of a disturbing trend to add to header files which over the years have greatly increased the amount of code a compiler must process.
- Most **T***s are non-owning, so **std::observer_ptr<T>** is complicating the common case for the benefit of a minority case.
- Using **std::observer_ptr<T>** will slow down unoptimized code.
- Adding **std::observer_ptr<T>** would be seen as WG21 recommending its use over **T*** (whether we want to do that or not) making these problems universal.

This would complicate teaching/explanation and reinforce C++'s reputation for complexity. Students of all ability want to know how a feature is implemented; explaining **std::observer_ptr<T>**'s implementation is far, far harder than explaining **T***. Using combinations of **T***, **std::observer_ptr**, and **std::unique_ptr** would be necessary, and learning how to do this well is non-trivial (see "Usage concerns" below).

Benefits

Obviously, **observer_ptr** offers some benefits (or it would not have been voted in).

- Not all pointers are non-owning, so some way of distinguishing the two uses of pointers is obviously useful.
- You can't convert an **observer_ptr** to a **void*** without a cast. That is sometimes a benefit.
- You can't apply **delete** to an **observer_ptr**. Good.
- You can't increment an **observer_ptr**. That is sometimes a benefit.
- In a transition from a traditional codebase to one using explicit owners there can be a need to distinguish known non-owners (observers) from yet unexamined pointers.

The question thus becomes "do the benefits outweigh the problems?"

Do we have experience reports?

Usage concerns

We can classify current used of pointers like this:

- *Owner*; that is, must be **delete**d
- *Non-owner*; that is, must not be **delete**d
- *Iterator*; that is, can be used with **++** and needs some "end of legal range" information for safe use; iterators should be non-owning.

Basically, an **observer_ptr** is a non-owner that is not an iterator. I am very doubtful that's the right point in the design space.

Consider:

```
void f1(std::observer_ptr<int>);
void f2(int*);

void g()
{
        observer_ptr<int> p{new int};
        int* q = new int;
        f1(p);          // OK
        f2(p);          // error
        f1(q);          // error
        f2(q);          // OK
        ++p;            // error
        ++q;            // OK (as ever)
        p=q;            // error
        q=p;            // error
        delete p;       // error
        delete p.get(); // OK
        delete q;       // OK (as ever)
        delete q.get(); // error (someone got confused)
}
```

We will have to decide whether functions take **observer_ptr**s or raw pointers. Also our style of use, including the use of **new**, must take into account that **observer_ptr**s serve only a very limited need.

For ownership, **unique_ptr** is a good solution, so we would have to deal with combinations of **unique_ptr**s, **observer_ptr**s, and **T***s.

For iteration and non-owning references to multiple objects (e.g., arrays), **Range**s and **span**s are (IMO) a much better solution than a mix of **observer_ptr**s and raw pointers.

Alternatives

If you need a pointer representing non-ownership, I suggest:

```
template<typename T> using observer_ptr = T*;
```

Until we replace that **typename** with the appropriate concept this will be the best definition:

- It allows people to mark pointers as non-owning.
- It doesn't create interface problems.
- It interoperates perfectly with **T***s.
- The benefits from an **observer_ptr** class could be had in the form of compiler warnings and static analysis tools.

However, I don't propose that because that is still "the tail wagging the dog" and doesn't keep the simplest and most common case simple. What we should do instead is to mark pointers that are owners

as owners. We do that using **unique_ptr** and **shared_ptr** except when we need to pass a pointer through a C-style interface. For that case, I recommend GSL's

template<typename T> using owner = T;

or the more stl-style

template<typename T> using owner_ptr = T*;

This minimizes the syntactic clutter, enables humans and tools to recognize the intent, and doesn't cause interface problems.

Using both an owner and a non-owner (possibly called observer) alias could facilitate a transition as mentioned in "benefits."

Also, a user who wants to can always use **experimental::observer_ptr** or equivalent.