

Revisiting `std::shared_ptr` comparison

Document number: N2637=08-0147
Document title: Revisiting `std::shared_ptr` comparisons
Author: Hervé Brönnimann
Contact: hbronnimann@bloomberg.net
Organization: Polytechnic University and Bloomberg L.P.
Date: 2008-5-16
Number: WG21/N2637
Working Group: Library

Abstract

We propose an alternate value-based specification for shared pointer comparisons that restores the consistency between `operator==` and `operator<` and addresses all the points raised in the rationale for the current (ownership-based) specification [2].

I Motivation and Scope

The definition of `operator<` for `std::shared_ptr` and its inconsistency with `operator==` is extremely surprising and counter-intuitive, especially for novices. As discussed in [2], there are two possible definitions for this operator: one is value-based (`p.get() < q.get()`) and the other ownership-based. The one currently chosen for the Working Draft [1] is the ownership-based, for reasons explained in [2]. Our proposal is to restore the value-based and solve the difficulties which prevented its selection in the Working Draft. Besides the desire to retain intuition and the benefits of a straightforward definition, an additional motivation is that the intuitive definition of `operator<` is consistent for almost all smart pointer types we can imagine, now and in the future, whereas the other definition is specific to `std::shared_ptr`. Writing generic code for pointer-like classes is easier if the definition is consistent. One more motivating factor is that aliasing allows

two shared pointers to point into the same array. Such pointers should have an `operator<` that is consistent with their position in the array. The current definition would make such pointers equivalent to one another. If we ever add `shared_ptr<T[]>` to the standard (e.g., in TR2 or TR3), this discrepancy would become even more problematic.

The main point of this proposal is to change the return clause of `operator<`. Specifically, we propose to:

- make `operator<` for `std::shared_ptr` consistent with `operator==`, and specify `std::less` accordingly (see below);
- remove `operator<` for `std::weak_ptr` altogether (and therefore `std::less`), as is done for `operator==` in `std::function`;
- provide ownership-based comparisons as `p.before(q)`, for both `std::shared_ptr` and `std::weak_ptr`;
- provide a polymorphic functor `std::owner_less` returning `bool`, comparing an instance of `std::shared_ptr` and `std::weak_ptr` with another instance of `std::shared_ptr` and `std::weak_ptr`.

This alternate solution to the problem of designing comparisons for shared pointers has the only drawback that the functor `std::owner_less` must be provided explicitly to associative containers and algorithms (with the currently proposed specifications, none is required). This, however, may more appropriately convey the intent (where the key to the associative container is in effect the pointer owned, and not stored, by the shared pointer).

A minor issue is that the standard declares `operator<`, but not `operator>`, or `operator<=`, or `operator>=`. They must be declared in the synopsis of header `<memory>` (pages 530) and in that of class `std::shared_ptr` (page 548). By 20.2.1 [operators], it is not necessary to provide definitions for them in section 20.6.6.2.7 [util.smartptr.shared.cmp].

Note that the specializations of `std::less<T*>` must not use `operator<`, since the former defines a total order (paragraph 20.5.7[comparisons], clause 8) while the latter is only required to compute a partial order (paragraph 5.9 [expr.rel], clause 2). Correspondingly, specializations of `std::less<shared_ptr<T> >` must be declared and defined as follows:

```
template <class T>
    std::less<shared_ptr<T>>
    : public binary_function<shared_ptr<T>, shared_ptr<T>, bool>
    {
        bool operator()(shared_ptr<T> const& x,
```

```

        shared_ptr<T> const& y) const
    {
        return std::less<T*>()(x.get(), y.get());
    }
};

```

Failure to do so would imply that `std::less<shared_ptr<T> >` would use `operator<(shared_ptr<T> const&, shared_ptr<T> const&)`, and would fail to define a total order on `shared_ptr<T>`.

II Impact On the Standard

This proposal modifies the existing clauses of `operator<` and removes this operator for `weak_ptr`. The behavior of `std::map<shared_ptr<T>, U>` is changed, and (the perhaps more common) `std::map<weak_ptr<T>, U>` no longer compiles; occurrences need to be replaced by `std::map<shared_ptr<T>, U, std::owner_less<std::shared_ptr<T> > >` and `std::map<weak_ptr<T>, U, std::owner_less<std::weak_ptr<T> > >`.

The definitions of `>`, `<=`, `>=`, the partial specializations of templates `greater`, `less`, `greater_equal`, and `less_equal` for `shared_ptr`, and `std::owner_less` are pure library additions and should not affect existing code.

III Design Decisions

The design basically aims at satisfying three constraints:

- compatibility between `operator==` (necessarily value-based) and `operator<` for `std::shared_ptr`;
- impossibility to order weak pointers based on their value (so that `operator<` for `std::weak_ptr` must be ownership-based);
- consistency between `std::map<shared_ptr<T>, U>` and `std::map<weak_ptr<T>, U>` (where the only difference is that the former keeps the elements alive).

It clearly is not possible to satisfy all three constraints at once. We simply choose to satisfy a different set of constraints, which we deem more intuitive, than were chosen in the current Working Draft [1]. In addition, we feel that it is easier to explain

why `operator<` isn't defined for `std::weak_ptr` than to explain the current design of `operator<` for `std::shared_ptr` (although [2] is probably good reading no matter what). Finally, we feel that requiring the explicit usage of the comparison functor `std::owner_less` in sets and maps better conveys the intended usage. Note that users for whom it does not matter what the ordering `std::shared_ptr` is, just so long as one is defined, can retain their code unchanged.

Regarding the `owner_before()` member templates, the name (suggested as `before()` by P. Dimov as consistent with `type_info::before`) has been prefixed by `owner_`, because the comparison is based only on ownership. Unlike `owner_less`, they provide mixed comparisons of shared and weak pointers of different types.

Regarding the `owner_less` class template, we have chosen to leave it undefined except for the two specializations. This gives `owner_less` a more consistent syntax: `owner_less<shared_ptr<T> >` is then similar to `less<shared_ptr<T> >`. Also, specializing for `owner_less<shared_ptr<T> >` doesn't prohibit additional specializations for other smart pointer types in the future.

Regarding the polymorphism of the `std::owner_less` functor, we made that choice somewhat arbitrarily and out of a desire for simplicity and allowing mixed comparisons. If mixed comparisons are not deemed important, it is equally possible to only define a single `operator()` in each specialization.

IV Proposed Text for the Standard

All references are taken against the most recent version of the Working Draft [1].

IV.1 Operators

In the **header `<memory>` synopsis**, paragraph // 20.6.6.2.7, *shared_ptr comparisons:*, add after `operator<`:

```
template<class T, class U>
    bool operator>(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U>
    bool operator<=(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U>
    bool operator>=(shared_ptr<T> const& a, shared_ptr<U> const& b);
```

In section 20.6.6.2.7 [util.smartptr.shared.cmp], remove operator!= and clauses 3,4 (by 20.2.1 [operators], it is not necessary to provide a definition), and change paragraph 5 to:

```
template<class T, class U>
    bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);
```

5 *Returns:* `x.get() < y.get()`

Also append at the end of this section:

6 For templates `greater`, `less`, `greater_equal`, and `less_equal`, the partial specializations for `shared_ptr` yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not. Moreover, `less<shared_ptr<T> >::operator()(a, b)` shall return `std::less<T*>::operator()(a.get(), b.get())`.

In section 20.6.6.3 [util.smartptr.weak], add to the class body in the synopsis:

```
// comparison
template<class Y> bool operator<(weak_ptr<Y> const&) const = delete;
template<class Y> bool operator<=(weak_ptr<Y> const&) const = delete;
template<class Y> bool operator>(weak_ptr<Y> const&) const = delete;
template<class Y> bool operator>=(weak_ptr<Y> const&) const = delete;
```

and remove `// comparison...` from synopsis (in the `std` namespace) in both 20.6 (<memory>) and 20.6.6.3, as well as the whole section 20.6.6.3.6 [util.smartptr.weak.cmp].

IV.2 Member template `owner_before`

In section 20.6.6.2 [util.smartptr.shared], add to the class body in the synopsis:

```
// observers
...
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
```

and in section 20.6.6.2.5 [util.smartptr.shared.obs], add:

```
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
```

19 *Returns:* an unspecified value such that

— `x.owner_before(y)` defines a strict weak ordering as described in 25.3;

- under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

In section 20.6.6.3 [`util.smartptr.weak`], add to the class body in the synopsis:

```
// observers
...
template<class U> bool owner_before(shared_ptr<U> const& b);
template<class U> bool owner_before(weak_ptr<U> const& b);
```

and in section 20.6.6.3.5 [`util.smartptr.weak.obs`], add:

```
template<class U> bool owner_before(shared_ptr<U> const& b);
template<class U> bool owner_before(weak_ptr<U> const& b);
```

- 9 *Returns*: an unspecified value such that
 - `x.owner_before(y)` defines a strict weak ordering as described in 25.3;
 - under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

IV.3 Class template `owner_less`

Finally, in the `<memory>` synopsis in 20.6, after // 20.6.6.3.7, *weak_ptr specialized algorithms*, add:

```
// 20.6.6.4 Class template owner_less:
template<class T> class owner_less;
```

and insert a new section between 20.6.6.3 and 20.6.6.4:

20.6.6.4 Class template `owner_less` [`util.smartptr.ownerless`]

- 1 The `owner_less` class template allows ownership-based mixed comparisons of shared and weak pointers.

```
namespace std {
    template<class T> struct owner_less;

    template <class T> struct owner_less<shared_ptr<T> >
        : binary_function<shared_ptr<T>, shared_ptr<T>, bool>
    {
```

```

    typedef bool result_type;
    bool operator()(shared_ptr<T> const&, shared_ptr<T> const&) const;
    bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
};

template <class T> struct owner_less<weak_ptr<T> >
    : binary_function<weak_ptr<T>, weak_ptr<T>, bool>
{
    typedef bool result_type;
    bool operator()(weak_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
};

```

- 2 `operator()(x,y)` returns `x.before(y)`. [*Note*:Note that
- `operator()` is a strict weak ordering as described in 25.3;
 - under the equivalence relation defined by `operator()`, `!operator()(a,b) && !operator()(a,b)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.
- *end note*]

V Acknowledgements

Thanks to Peter Dimov for pointing out [2], and Peter Dimov and Pablo Halpern for providing feedback on earlier versions of this proposal. Thanks to my co-workers Pablo Halpern, Vladimir Kliatchko and John Lakos for guidance and support.

References

- [1] Working Draft, Standard for Programming Language C++. Document number: N2521=08-0031, 2008-02-04. (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2521.pdf>)
- [2] Smart Pointer Comparison Operators Peter Dimov <pdimov@mmltd.net> Document number: N1590=04-0030, 2004-02-11. (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1590.html>)