Improving the wording of `std::shared_ptr`

Document number:	NN2638=08-0148
Document title:	Improving the wording of `std::shared_ptr`
Author:	Hervé Brönnimann
Contact:	hbronnimann@bloomberg.net
Organization:	Polytechnic University and Bloomberg L.P.
Date:	2008-1-5
Number:	WG21/N2638=08-0148
Working Group:	Library

Abstract

The class template `std::shared_ptr` wording could be improved by referencing more clearly the three entities making up a shared pointer: the pointer stored, the pointer owned (under shared ownership), and the deleter. We propose such a wording.

I Motivation and Scope

The wording of the current Working Draft [1] of the C++ Standard does not clearly separate between the pointer *stored* by `shared_ptr`, and the pointer *owned*.

Owned pointer vs. stored pointer.

Section 20.6.6.2 [util.smartptr.shared], paragraph 1, talks about the class template `shared_ptr` storing a pointer, but does not introduce explicit the pointer owned by the shared pointer (with the corresponding object of some type `Y` that is being owned). This makes it hard for the reader to realize that this is different from the pointer stored by this same shared pointer (the address of the object of type `T` that is being pointed to). This difference exists because of pointer conversions, but is

made quite explicit by the introduction of aliasing [2, 3]. It also does not define what an "empty" pointer is.

The proposed resolution of this problem is to explicitly introduce all three distinct entities used by a shared pointer (owned object, owned deleter, and stored pointer) in the introduction of 20.6.6.2 and to systematically use those entities in the remaining documentation.

The template `weak_ptr` suffers from the same drawbacks. In addition, the notion of "weak reference" is *nowhere* defined in this standard, nor used anywhere else than 20.6.6.3, paragraph 1, and it is far from self-explanatory. The suggested wording below makes the definition hopefully clear(er).

In LWG 711, it is mentioned: "The LWG is getting more uncomfortable with the aliasing proposal (N2351) now that we realize some of its implications, and we need to keep an eye on it, but there isn't support for removing this feature at this time." We hope that this proposal will restore some measure of comfort and help provide support for keeping aliasing.

Another proposal (mentioned in LWG 710, not submitted to my knowledge) by Alberto Ganesh Barbati [6] proposes to use wording such as "has no ownership". This is essentially what this proposal does. We do not, however, propose to add non-null pointers with no ownership.

Other missing functionality.

A related issue that could easily be corrected is that there is (for no apparent reason) no member function for creating an alias from a normally-constructed `shared_ptr`, mirroring the "aliasing" constructor:

```
template<class Y> reset(shared_ptr<Y> const& r, T *p);
```

In an earlier draft, I proposed a function template `std::get_shared<Y>(p)`, very similar to `get_deleter`, to retrieve the pointer owned by `p`. I do not have a strong use case for it, though, and so it is not in the current proposal. I will be happy to submit an additional (separate) proposal if requested.

Specific examples.

This section provides additional motivation. If such is not needed, or the reader already understands the issue, this section can be skipped.

Problems come in the formulations of several members Returns/Effects: clauses. For instance, read LWG Issues 710 [4] and 711 [5]. The latter issue also mentions the question of what "empty" means. The term is introduced in [util.smartptr.shared.const] paragraph 1, but is never defined since, in the absence of aliasing, "empty" simply refers to "not storing a pointer". We suggest clarifiying the definition by making empty mean "owning no pointer", which is equivalent to `use_count() == 0` (the meaning I believed is used in the current draft).

Note that in clause 14, the difference between storage and ownership is clearly mentioned. Compare this with ([util.smartptr.shared.const]):

```
template<class Y> explicit shared_ptr(Y* p);
```

...

5 *Effects:* Constructs a `shared_ptr` object that owns the pointer `p`.

6 *Postcondition:* `use_count() == 1 && get() == p`.

Here, we are told nothing abouth the stored pointer of the newly constructed `shared_ptr`, until the precondition mentions `get() == p`. It would have been better (as is done in e.g., [util.smartptr.shared.const] paragraph 20) to say

5 *Effects:* Constructs a `shared_ptr` object that both owns and stores the pointer `p`.

Consider the `auto_ptr` constructor ([util.smartptr.shared.const]):

```
template<class Y> shared_ptr(auto_ptr<Y>& r);
```

...

32 *Requires:* `r.release()` shall be convertible to `T*`. `Y` shall be a complete type. The expression delete `r.release()` shall be well-formed, shall have well defined behavior, and shall not throw exceptions.

33 *Effects:* Constructs a `shared_ptr` object that stores and owns `r.release()`.

The problem with the formulation of paragraph 33 is that the stored pointer is always of type T*, thus `r.release()` must be cast to T*. But the owned pointer should remain of type Y* or else undefined behavior results. This is not at all apparent from the formulation which seems to imply that the pointers stored and owned are the same (both in type and value).

For another (crucial) example, consider ([util.smartptr.shared.mod]):

```
void swap(shared_ptr& r);
```

1 *Effects:* Exchanges the contents of `*this` and `r`.

2 *Throws:* nothing.

Paragraph 1 is hopelessly vague, and to top it off there are no postconditions to disambiguate. Are both the owned object and the pointer value swapped (the intended meaning)? Just ownership? or just the pointer value? Under the formulation of [util.smartptr.shared] paragraph 1, we might conclude that the stored pointer (the only pointer mentioned) reflects ownership, thus only ownership is swapped, which of course is not what the standard *means*.

II Impact On the Standard

This proposal is purely editorial: it clarifies the wording of many members of `shared_ptr` class template. It addresses points which were raised in LWG Issues 710 and 711.

III Design Decisions

1 The design is purely driven by a desire for precision and consistency. There are no changes to the syntax or to the contracts. I used 0 instead of `std::nullptr` because of lack of familiarity, but the proposed text could be updated further to use `std::nullptr`.

IV Proposed Text for the Standard

All references are to the current Working Draft [1].

In section 26.6.6.2 [util.smartptr.shared], change the text of paragraph 1 to:

1 The `shared_ptr` class template owns a pointer to an object and (optionally) a deleter, and stores a pointer to its template parameter `T`. The owned pointer, usually obtained via `new`, is usually (but not necessarily) of some type `Y*` convertible to `T*` and usually equal to the stored pointer. `shared_ptr` implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the owned pointer, using the owned deleter if any.

In section 26.6.6.2.1 [util.smartptr.shared.const], change the paragraphs of the same number to:

1 *Effects:* Constructs an empty `shared_ptr` object (i.e., that owns no pointer) and that stores 0.

5 *Effects:* Constructs a `shared_ptr` object that owns the pointer `p` and stores a copy of `p`.

10 *Effects:* Constructs a `shared_ptr` object that owns the pointer `p` and the deleter `d`, and stores a copy of `p`. The second constructor shall use a copy of `a` to allocate memory for internal use.

20 *Effects:* If `r` is empty, constructs an empty `shared_ptr` object; otherwise, constructs a `shared_ptr` object that shares ownership with `r` and stores a copy of the pointer stored in `r`.

28 *Effects:* Constructs a `shared_ptr` object that shares ownership with `r` and stores a copy of the pointer stored in `r`.

33 *Effects:* Constructs a `shared_ptr` object that owns `r.release()` and stores a copy of that pointer.

In section 26.6.6.2.4 [util.smartptr.shared.mod], change paragraph 1 to:

1 *Effects:* Exchanges ownership of pointer and deleter (if any) owned by `*this`, and value of pointer stored by `*this`, with those of `r`.

In section 26.6.6.3 [util.smartptr.weak], change the text of paragraph 1 to:

1 The `weak_ptr` class template tracks (but does not share) ownership of a pointer that is already owned by a `shared_ptr`, and stores a pointer to its template parameter `T`, usually (but not necessarily) equal to the tracked pointer (after conversion to `T*`). To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

and append to this paragraph (before the class synopsis):

[*Note:* Since the pointer tracked by a `weak_ptr` instance is not counted in the `use_count` of the `shared_ptr` instances that share ownership of the tracked pointer, the `weak_ptr` can be used to prevent leakage of resources owned by instances owning `shared_ptr` to each other in a cycle, by making one of these `shared_ptr` a `weak_ptr` instead. — *end note*]

In section 20.6.6.3.1 [util.smartptr.weak.const], change paragraph 1 to:

1 *Effects:* Constructs an empty `weak_ptr` object (i.e., that tracks ownership of no pointer) and that stores 0.

Split the clause of the three constructors in 4–7 into:

```
template<class Y> weak_ptr(shared_ptr<Y> const& r);
```

4 *Requires:* `Y*` shall be convertible to `T*`.
5 *Effects:* If `r` is empty, constructs an empty `weak_ptr` object that stores 0; otherwise, constructs a `weak_ptr` object that tracks ownership of the pointer owned by `r` and stores a copy of the pointer stored in `r`.
6 *Postconditions:* `use_count() == r.use_count()`.
7 *Throws:* nothing.

```
weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);
```

8 *Requires:* `Y*` shall be convertible to `T*`.
9 *Effects:* If `r` is empty, constructs an empty `weak_ptr` object that stores 0; otherwise, constructs a `weak_ptr` object that tracks the same ownership as `r` and stores a copy of the pointer stored in `r`.
10 *Postconditions:* `use_count() == r.use_count()`.
11 *Throws:* nothing.

In section 20.6.6.3.1 [util.smartptr.weak.dest], change paragraph 1 to:

1 *Effects:* Destroys this `weak_ptr` object but has no effect on the object it tracks ownership of.

In section 20.6.6.3.1 [util.smartptr.weak.mod], change paragraph 1 to:

1 *Effects:* Exchanges pointers tracked and stored by `*this` with those of `r`.

In section 20.6.6.3.1 [util.smartptr.weak.obs], change paragraph 1 to:

1 *Returns:* 0 if `*this` is empty; otherwise, the number of `shared_ptr` instances that share ownership of the pointer tracked by `*this`.

In section 26.6.6.4 [util.smartptr.enab], change paragraphs 1 and 11 to:

1 A class `T` can inherit from `enable_shared_from_this<T>` to inherit the `shared_from_this` member functions that obtain a `shared_ptr` instance both owning and storing the pointer `this`.

11 *Returns:* A `shared_ptr<T>` object `r` that shares ownership with `p` and stores `p`.

IV.1 Adding `reset`

At the end of *// 20.6.6.2.4, modifiers*, in the synopsis, add:

```
template<class Y> void reset(shared_ptr<Y> const& r, T *p);
```

In section 20.6.6.2.4 [util.smartptr.shared.mod], append the following paragraph:

```
template<class Y> void reset(shared_ptr<Y> const& r, T *p);
```

7 *Effects:* Equivalent to `shared_ptr(r, p).swap(*this)`.

V Acknowledgements

Thanks to Peter Dimov for pointing out [4] and [5] and providing feedback on an earlier version 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] Improving Usability and Performance of TR1 Smart Pointers. Vladimir Kliatchko and Ilougino Rocha. (`www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2351.htm`)

[3] Improving `shared_ptr` for C++0x, Revision 2. Peter Dimov (`<pdimov@pdimov.com>`), Beman Dawes, `<bdawes@acm.org>`). Document number: N2351=07-0211.

[4] LWG Issue 710. Peter Dimov. (http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#710)

[5] LWG Issue 711. Peter Dimov. (http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#711)

[6] Alberto Ganesh Barbati. (http://barbati.net/c++/shared_ptr.pdf).