Re-Gaining Exclusive Ownership from `shared_ptr`s

Document #:	P1116R0
Date:	June 17, 2019
Project:	Programming Language C++
	SG1 (for implementability)
	Library Evolution Working Group (for API review)

Reply-to:	Marc Mutz <marc.mutz@kdab.com>

Abstract

We propose to add a function, `lock_exclusive()`, to `shared_ptr` to re-gain exclusive ownership of the payload. Exclusive ownership is defined as a sole `shared_ptr` owning, combined with no `weak_ptr`s referencing, the resource. The reason to exclude `weak_ptr`s is that the presence of a `weak_ptr` (e.g. in a separate thread of execution) could materialize new `shared_ptr`s at any time, which would invalidate the exclusive ownership state.

The primary use-case is efficient copy-on-write implementations using `shared_ptr<const T>` to hide the details of ref-counting, a la Sean Parent's `document` example from [S.Parent]. As long as the objects involved are immutable, mutation is performed by copying from the existing state and modifying the state before storing in the new `shared_ptr<const T>`. The existing `unique()` function is not sufficient for this purpose, because it does not take the existence of associated `weak_ptr`s into account, which could materialize new `shared_ptr`s at any time from a different thread.

Contents

1 Motivation and Scope

1.1 Efficient Copy-on-Write

Sean Parent, in [S.Parent], introduced a pattern of using `shared_ptr<const T>` to share common data between different versions of a document, without the need to deep copy. In this way, he can implement undo as a simple `vector<document>` even with very large `documents`.

`shared_ptr<const T>` works well for this as long as one works with immutable objects. But it is well-known that actions[1] can often be more efficient than transformations[2], so immutability of objects may not always be desired.

Let's look at a typical mutator of a class that uses `shared_ptr<const T>` to store data:

Listing 1: Typical mutator of a class that uses `shared_ptr<const T>`

```
shared_ptr<const Data> m_data;

void set_foo(T foo) {
  auto uniq = clone_or_new(m_data);
  uniq->foo = std::move(foo);
  m_data = std::move(uniq);
}
```

where `clone_or_new` might be implemented like this:

Listing 2: Inefficient implementation of `clone_or_new()`

```
static auto clone_or_new(const shared_ptr<const Data> & sp) {
  return sp ? make_shared<Data>(*sp) : make_shared<Data>();
}
```

This is the classical copy-on-write implementation, with a drawback: every setter allocates, even if it just sets an `int` field. This is clearly not acceptable in a production-quality implementation, in particular for APIs that employ something like Qt's [Qt] Property-Based Design [API], which calls for preferring setters over long lists of contructor arguments:

Listing 3: Comparison between Qt 3 and Qt 4 slider creation

```
// Qt 3:
QSlider qt3(0, 100, 50, 10); // allocs once, but unreadable

// Qt 4+:
QSlider slider;              // allocates, because all properties
                             // have defined defaults
slider.setRange(0, 100);     // would allocate
slider.setPageStep(10);      // would allocate
slider.setValue(50);         // would allocate
```

[1] modifying an object's state in-place

[2] copying an object's state, making changes, and then creating a new object with the new state, leaving the old object alone

The obvious optimisation is to not create a new copy of the data if we're the sole owner:

Listing 4: `clone_or_new()` using `unique()`

```
static auto clone_or_new(shared_ptr<const Data> & sp) {
  return sp.unique() ? const_ptr_cast<Data>(std::move(sp)) :
         sp          ? make_shared<Data>(*sp) :
         /* else */    make_shared<Data>() ;
}
```

But this fails in multithreaded contexts, as `shared_ptr::unique()` is only approximate. Besides, `shared_ptr::unique()` is deprecated as of C++17.

The problem, of course, is that a new strong reference may be created as a copy of `sp` in between the calls to `unique()` and the move from `sp`, which will lead to a data race on `sp->foo` further down the road, as the return value of `clone_or_new()` is assumed, by `set_foo()`, to represent an exclusive owner.

We could try to define the problem away: assuming `sp.unique()` returns `true`, we're looking at the sole remaining strong reference. While the non-const function `set_foo()` executes, we can assume that no new copies of `sp` are taken, because the only way to do so would be from another thread. We can declare such use to be outside the contract of the function, along the lines of "to call mutators, you need to externally synchronise" as part of a "const access is thread-safe" policy.

And this view would be correct if it wasn't for `weak_ptr`, which can upgrade to a strong reference at any time.

It follows that the condition for when we are able to re-use the existing `shared_ptr` in `clone_or_new` is *not* `unique() == true`, but "weak reference count is one", iow: we control the only `shared_ptr` and there are no `weak_ptr`s registered with it. But we have no API to check for this implementation detail of `shared_ptr`.

Enter `shared_ptr::lock_exclusive()`.

This function returns a new owning pointer that represents the sole owner of the data, setting `*this` to `nullptr` upon success. The effects are atomic.

Our example function then becomes:

Listing 5: `clone_or_new()` using `lock_exclusive()`

```
static auto clone_or_new(shared_ptr<const Data> & sp) {
  auto uniq = const_pointer_cast<Data>(sp.lock_exclusive());
  if (!uniq)
    uniq = sp ? make_shared<Data>(*sp) : make_shared<Data>();
  return uniq;
}
```

1.2 Implementability

All implementations the authors have have come across share the split into a weak reference count, ref-counting the control block, and a strong reference count, ref-counting the lifetime of the payload object.

In all implementations, except `QSharedPointer`, the set of all `shared_ptr` instances are counted as one (1) in the weak reference count.

The condition for when `lock_exclusive()` succeeds is thus `weak_ref == 1 && strong_ref == 1`.

Proof: Since `strong_ref == 1`, there is exactly one `shared_ptr` (`*this`). Since `weak_ref == 1`, there is either one `weak_ptr` and no `shared_ptr` (contradicting the existence of `*this`), or there is at least one `shared_ptr` and no `weak_ptr`s. Taken together, it follows that there is exactly one `shared_ptr` (`*this`) and no `weak_ptr`s. ■

A simple simultaneous relaxed atomic load of `weak_ref` and `strong_ref` suffices to check for exclusive ownership, since taking a new copy of `*this` while a non-const member function is executing is already undefined behaviour, so the implementation of `lock_exclusive()` can assume that when `weak_ref` contains 1, it will stay that way for the remainder of the function.

No stronger memory ordering than relaxed is needed, either, since we don't touch the referenced data, in fact we change nothing except swapping pointers from `*this` to the return value.

The advantage of this implementation is that only `lock_exclusive()` needs to be added; all other operations, including `shared_ptr`/`weak_ptr` and `weak_ptr`/`shared_ptr` conversions, can remain unchanged.

NB: We'd like to ask for guidance from implementers about the general feasibility of this. The implementation sketched above requires a double-word relaxed atomic load to simultaneously determine that `weak_refs == 1` and `strong_refs == 1`, to exclude a second thread permanently converting `shared_ptr`s into `weak_ptr`s and vice versa. Maybe there are other protocols that can be used on platforms without double-word atomic operations?

2 Impact on the Standard

Minimal. We propose to add a new member function to `shared_ptr`. No other part of the standard is affected.

3 Proposed Wording

3.1 Changes to [N4810]

In section [**util.smartptr.shared**]:

- at the end of paragraph (1), before the synopsis, continue the last sentence with

  > [does not own a pointer], and to be an *exclusive owner* if it shares ownership with no other `shared_ptr` and there are no `weak_ptr` objects referring to it.

- at the end of *modifiers* section, add the function

```
[[nodiscard]] shared_ptr lock_exclusive() noexcept;
```

In section [**util.smartptr.shared.mod**]:

- add new paragraph at the end:

  `[[nodiscard]] shared_ptr lock_exclusive() noexcept;`

  – *Returns:* `std::move(*this)` if `*this` is an *exclusive owner (insert reference to [**util.smartptr.shared**]/1)*, otherwise returns {}. This function executes atomically.
  – *Synchronization:* none.

3.2 Feature Macro

We propose to use a new macro, `__cpp_lib_shared_ptr_lock_exclusive`, to indicate a library's support for this feature.

4 Design Decisions

4.1 Why `shared_ptr`?

One legitimate question is: why put this functionality into `shared_ptr` instead of inventing a new type (pair of types)? In particular, the difficulties of dealing with `weak_ptr`, which is probably not used at all with `shared_ptr`s that are used for copy-on-write, make inventing a new pair of types to represent shared and unique ownership attractive. However, there is the problem with names. What to call such new types if `shared_ptr` and `unique_ptr` are already taken? Also, `shared_ptr` is probably already in use for CoW systems, so adding the functionality there, even if not enitrely trivial, probably gives the biggest bang for the buck.

That said, we are open to explore the alternative with two different types as well, should LEWG prefer that approach.

4.2 Dealing with `weak_ptr`s

If, for the use-cases in which `lock_exclusive()` is useful, we do not envision use of `weak_ptr` at all, why not make calling `lock_exclusive()` in the presence of associated `weak_ptr`s undefined behaviour?

This is an attractive option, too, since it would mean that implementations would become easier. In this case, we would not need `lock_exclusive()` at all, though, since we could just un-deprecate `unique()` instead and use the implementation in Listing 4.

4.3 Return Type

To represent unique ownership, we customarily use `unique_ptr`, so `lock_exclusive()` could conceivably return a `unique_ptr`. But `shared_ptr`'s custom, type-erased deleter and the `make_shared` optimization of co-locating the control block and the payload object in a single memory allocation make this unrealistically complex: The `unique_ptr` returned would need to have some form of type-erased deleter template argument, making it effectively a different type from `unique_ptr`, and mostly layout-compatible with a `shared_ptr`.

4.4 Alternative Function Names

We have chosen to call the function `lock_exclusive()` to piggy-back on the existing `weak_ptr::lock()` function, which performs a similar (in particular, atomic), but different task. The `exclusive` part is intended to show that what is returned is an exclusive owner, if any, of the payload data. The word "exclusive" has been chosen over "unique" to avoid unwanted connotation with `unique_ptr`.

Alternatives considered include:

`unique()` This would have been the first choice if it wasn't already taken to mean `strong_ref == 1`. The name also has said unwanted connotation with `unique_ptr`, though.

`lock_unique()` Not preferred for the unwanted connotation with `unique_ptr`. Would be a good choice if `lock_exclusive()` returned an actual `unique_ptr`, though.

`release()` For symmetry with `unique_ptr::release()`; but we do not release the resource here, but merely conditionally move it into a new `shared_ptr`.

4.5 Atomic Shared Pointers

`std::atomic<shared_ptr>`, as added for C++20, is merely a container for `shared_ptr`s; they are not themselves `shared_ptr`s. E.g. `std::atomic<weak_ptr>` does not have a `::lock()` member function. We therefore do not propose to add `lock_exclusive()` to `std::atomic<shared_ptr>`, even though it would probably benefit some use-cases.

5 References

[S.Parent] Sean Parent
C++ Seasoning
in: *Going Native 2013*
https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning

[Qt] http://www.qt.io

[API] *Property-Based APIs*
in: *Qt Wiki: API Design Principles*
https://wiki.qt.io/API_Design_Principles#Property-Based_APIs

[N4810] Richard Smith (editor)
Working Draft: Standard for Programming Language C++
http://wg21.link/N4810