Remove Deprecated `shared_ptr` Atomic Access APIs From C++26

Annex D of the C++ Standard, deprecated features, maintains an easily misused API for atomic access to `shared_ptr` objects. This paper proposes removing that API from the C++ Standard Library.

## 1 Abstract

Annex D of the C++ Standard, deprecated features, maintains an easily misused API for atomic access to `shared_ptr` objects. This paper proposes removing that API from the C++ Standard Library.

## 2 Revision history

### 2.1 R0: Varna 2023

Original version of this document, extracted from the C++23 proposal [P2139R2].

Key changes since that earlier paper:

- Rebased wording onto [N4944]
- Added examples of how to update deprecated code
- Consider proposal to minimize impact on header usage
- Added Annex C wording

## 2.1 R0: Varna 2023

Original version of this document, extracted from the C++23 proposal [P2139R2].

Key changes since that earlier paper:

- Rebased wording onto [N4944]
- Added examples of how to update deprecated code
- Consider proposal to minimize impact on header usage
- Added Annex C wording
3 Introduction

At the start of the C++23 cycle, [P2139R2] tried to review each deprecated feature of C++ to see which we would benefit from actively removing and which might now be better undeprecated. Consolidating all this analysis into one place was intended to ease the (L)EWG review process but in return gave the author so much feedback that the next revision of the paper was not completed.

For the C++26 cycle, a much shorter paper, [P2863R0], will track the overall analysis, but for features that the author wants to actively progress, a distinct paper will decouple progress from the larger paper so that the delays on a single feature do not hold up progress on all.

This paper takes up the deprecated C-style API for race-free access to shared_ptr objects, D.24 [depr.util.smartptr.shared.atomic].

4 History

This removal was originally suggested for C++23 as part of [P2139R2], and at the LEWG telecon of 2020/07/13 was deferred (without technical discussion) to SG1 for its initial review, after which it should come back to LEWG. That initial review did not occur, so this paper has been produced for C++26 to enable easier tracking of each deprecated topic.

4.1 Origin

The free-function API for atomic access to shared_ptr was introduced with C++11, which introduced both the concurrency-aware memory model (including atomics) and shared_ptr.

4.2 Deprecation

The API was first deprecated by C++20, along with the introduction of its type-safe replacement, atomic<shared_ptr<T>>.

5 Proposal

It is now time to complete the cycle and remove the original fragile facility.

The legacy C-style atomic API for manipulating shared pointers, provided since C++11, is subtle and frequently misunderstood: a shared_ptr object that is to be used with the atomic API can never be used directly, but (other than construction and destruction) may be manipulated only through the atomic API. Its failure mode on misuse (any direct use of that shared_ptr object, before, after, or concurrent with the first use of the atomic access API) is silently undefined behavior, typically producing a data race.

C++20 provides atomic<shared_ptr<T>>, a type-safe alternative that encapsulates its shared_ptr object, safely providing a complete replacement for the original functionality, and also providing support for atomic<weak_ptr<T>>.

5.1 Impact of Removal

There are no other overloads in the standard for the C style atomics interface taking pointers to T rather than pointers to atomic<T>, so all existing usage should be easily diagnosed by recompiling (if not already diagnosed by a deprecation warning today). The fix for old code should be as simple as replacing shared_ptr<T> with atomic<shared_ptr<T>> in the affected places. The existing C style atomic interface should then pick up support for the atomic<shared_ptr<T>> type.

For example, consider migrating this legal (but deprecated) program from the original C++11 API to the type safe C++20 form:
### Deprecated vs. Supported

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;memory&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>std::shared_ptr&lt;int&gt; x;</code></td>
<td><code>#include &lt;atomic&gt;</code></td>
</tr>
<tr>
<td><code>int main() {</code></td>
<td><code>std::atomic&lt;shared_ptr&lt;int&gt;&gt; x;</code></td>
</tr>
<tr>
<td><code>std::shared_ptr&lt;int&gt; y =</code></td>
<td><code>    int main() {</code></td>
</tr>
<tr>
<td><code>std::atomic_load(&amp;x);</code></td>
<td><code>std::shared_ptr&lt;int&gt; y =</code></td>
</tr>
<tr>
<td><code>y.reset(new int(42));</code></td>
<td><code>std::atomic_load(&amp;x);</code></td>
</tr>
<tr>
<td><code>std::atomic_store(&amp;x, y);</code></td>
<td><code>y.reset(new int(42));</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>std::atomic_store(&amp;x, y);</code></td>
</tr>
</tbody>
</table>

Observe that only the global variable is changed by wrapping it in a `std::atomic`. No further changes to the code are necessary, as the existing overloads for the C-style API expect `std::atomic<T>` pointers in the same argument positions, and those calls provide the correct behavior.

Note we must also `#include` the `<atomic>` header as the (never deprecated) C style API for atomics is defined in that header, once the deprecated overloads for `shared_ptr` have been removed from `<memory>`.

Alternatively, the user may prefer to further refactor the code to use the `std::atomic` member functions directly:

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>Refactored</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;memory&gt;</code></td>
<td><code>#include &lt;memory&gt;</code></td>
</tr>
<tr>
<td><code>std::shared_ptr&lt;int&gt; x;</code></td>
<td><code>std::atomic&lt;shared_ptr&lt;int&gt;&gt; x;</code></td>
</tr>
<tr>
<td><code>int main() {</code></td>
<td><code>int main() {</code></td>
</tr>
<tr>
<td><code>std::shared_ptr&lt;int&gt; y =</code></td>
<td><code>std::shared_ptr&lt;int&gt; y =</code></td>
</tr>
<tr>
<td><code>std::atomic_load(&amp;x)</code></td>
<td><code>x.load();</code></td>
</tr>
<tr>
<td><code>y.reset(new int(42));</code></td>
<td><code>y.reset(new int(42));</code></td>
</tr>
<tr>
<td><code>std::atomic_store(&amp;x, y);</code></td>
<td><code>x.store(y);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

While this refactored example contains more changes, it might be argued to be more idiomatic C++. Also, the header dependencies remain the same as the original code as full specification for `atomic<shared_ptr<T>>` is in the `<memory>` header.

### 5.2 Addressing the Header Dependency

One concern when migrating to type-safe use of `atomic<shared_ptr<T>>` is that the overloaded functions for atomic types are declared only in the `<atomic>` header. The “obvious” solution would be to add the relevant atomic overloads that correspond to the old `<shared_ptr>` API. Wording for this solution is provided below, but what are the precedents and concerns?

#### 5.2.1 Leave to user

MSVC already implicitly provides the API from just including `<memory>`. The gcc libstdc++ library strictly requires the user to include `<atomic>` for themselves. The LLVM libc++ library does not yet implement this C++20 library.
The main reason no solution is proposed here is that the author has an aesthetic distaste for the way the container API has leaked across headers, but in practice the wording below seems like a practical solution to simplify the process of updating code when the deprecated API is removed.

5.2.2 Add minimal atomic free-functions to <memory>

The obvious precedent is the set of container overloads in the <iterator> header, such as begin, end, and data. The same overloads are present in each container header so that clients of that container can easily use these functions. However, the specification for these functions remains in the iterators part of the standard. Similarly, the subset of atomic overloads could be added to the <memory> header along side the declaration of atomic<shared_ptr<T>>, while the specification remains untouched in the atomics part of the library.

5.2.3 Add all atomic free-functions to <memory>

The chief concern with this approach is that, while it contains all the overloads necessary to support the shared_ptr API, it is just a subset of the whole set of overloaded declarations in the <atomic> header, notably missing all volatile overloads, and those functions that would be ill-formed for shared_ptr. Given that the primary atomic<T> cannot be instantiated for types other than shared_ptr without also including the <atomic> header, it seems tricky to abuse this partial overload set in practice.

Another option, if worried about that partial overload set, would be to add all the free function interface of the <atomic> header to <memory>, but that seems to be an excessive creep of unnecessary functionality into another header.

5.2.4 Include <atomic> from <memory>

A simpler and more practical approach might be to simply mandate that the <memory> header directly includes <atomic>, just as it already includes <compare>. While this seems to be a bigger leak of excessive functionality through an unrelated header, in practice the implementation of shared_ptr requires the use of atomic integers to handle the strong and weak reference counts. Nevertheless, this does seem to be a more impactful change than necessary with potential to impact compile times.

6 Wording

All wording is relative to [N4944], the latest working draft at the time of writing.

Add to the synopsis of the header in 20.2.2 [memory.syn]:

```cpp
namespace std {
  // ...

  // 33.5.8.7[util.smartptr.atomic], atomic smart pointers
  template<class T> struct atomic; // freestanding
  template<class T> struct atomic<shared_ptr<T>>;
  template<class T> struct atomic<weak_ptr<T>>;

  // 33.5.9[atomics.nonmembers], atomic non-member functions
  template <class T>
    bool atomic_is_lock_free(const atomic<T>*) noexcept; // freestanding
  template <class T>
    T atomic_load(const atomic<T>*) noexcept; // freestanding
  template <class T>
    T atomic_load_explicit(const atomic<T>*, memory_order) noexcept; // freestanding

  template <class T>
```
void atomic_store(atomic<T>*, typename atomic<T>::value_type) noexcept;  // freestanding

template <class T>
void atomic_store_explicit(atomic<T>*,
        typename atomic<T>::value_type,
        memory_order) noexcept;

template <class T>
T atomic_exchange(atomic<T>*, typename atomic<T>::value_type) noexcept;  // freestanding

template <class T>
T atomic_exchange_explicit(atomic<T>*,
        typename atomic<T>::value_type,
        memory_order) noexcept;

template <class T>
bool atomic_compare_exchange_weak(atomic<T>*,
        typename atomic<T>::value_type*,
        typename atomic<T>::value_type) noexcept;

template <class T>
bool atomic_compare_exchange_strong(atomic<T>*,
        typename atomic<T>::value_type*,
        memory_order) noexcept;

template <class T>
bool atomic_compare_exchange_weak_explicit(atomic<T>*,
        typename atomic<T>::value_type*,
        memory_order, memory_order) noexcept;

template <class T>
bool atomic_compare_exchange_strong_explicit(atomic<T>*,
        typename atomic<T>::value_type*,
        memory_order, memory_order) noexcept;

// 20.3.4.1, class template out_ptr_t

template<class Smart, class Pointer, class... Args>
    class out_ptr_t;

// ... }

Annex C (informative) Compatibility [diff]
C.1 C++ and ISO C++ 2023 [diff.cpp23]
C.1.1 General [diff.cpp23.general]

Subclause C.1 lists the differences between C++ and ISO C++ 2023 (ISO/IEC 14882:2023, Programming Languages — C++), by the chapters of this document.

C.1.X Annex D: compatibility features [diff.cpp23.depr]

Change: Removal of atomic access API for shared_ptr objects.

Rationale: The old behavior was brittle. shared_ptr objects using the old API were not protected by the type system, and any interaction with code not using this API would silently produce undefined behavior. A complete type-safe replacement is provided in the form of atomic<shared_ptr<T>>.

Effect on original feature: Deletion of an old feature where a superior replacement exists within the standard.
Difficulty of converting: Violations will be diagnosed by the C++ translator, as there are no remaining overloads that would match such calls. Violations are address by replacing affected shared_ptr<T> objects with atomic<shared_ptr<T>>.

D.24 [depr.util.smartptr.shared.atomic] Deprecated shared_ptr atomic access

1 The header `<memory>` (20.2.2 [memory.syn]) has the following additions:

```cpp
namespace std {
    template <class T>
        bool atomic_is_lock_free(const shared_ptr<T>* p);
    template <class T>
        shared_ptr<T> atomic_load(const shared_ptr<T>* p);
    template <class T>
        shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
    template <class T>
        void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
    template <class T>
        void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
    template <class T>
        shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
    template <class T>
        shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
    template <class T>
        bool atomic_compare_exchange_weak(
            shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
    template <class T>
        bool atomic_compare_exchange_strong(
            shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
    template <class T>
        bool atomic_compare_exchange_weak_explicit(
            shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
            memory_order success, memory_order failure);
    template <class T>
        bool atomic_compare_exchange_strong_explicit(
            shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
            memory_order success, memory_order failure);
}
```

2 Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

3 The meaning of the arguments of type memory_order is explained in 33.5.4 [atomics.order].

```cpp
    template <class T>
        bool atomic_is_lock_free(const shared_ptr<T>* p);
```

4 Requires: p shall not be null.

5 Returns: true if atomic access to *p is lock-free, false otherwise.

6 Throws: Nothing.

```cpp
    template <class T>
        shared_ptr<T> atomic_load(const shared_ptr<T>* p);
```
Requires: p shall not be null.

Returns: atomic_load_explicit(p, memory_order::seq_cst).

Throws: Nothing.

    template<class T>
    shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

Requires: p shall not be null.

Requires: mo shall not be memory_order::release or memory_order::acq_rel.

Returns: *p.

Throws: Nothing.

    template<class T>
    void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);

Requires: p shall not be null.

Effects: As if by atomic_store_explicit(p, r, memory_order::seq_cst).

Throws: Nothing.

    template<class T>
    void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

Requires: p shall not be null.

Requires: mo shall not be memory_order::acquire or memory_order::acq_rel.

Effects: As if by p->swap(r).

Throws: Nothing.

    template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);

Requires: p shall not be null.

Returns: atomic_exchange_explicit(p, r, memory_order::seq_cst).

Throws: Nothing.

    template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

Requires: p shall not be null.

Effects: As if by p->swap(r).

Returns: The previous value of *p.

Throws: Nothing.

    template<class T>
    bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

Requires: p shall not be null.

Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst).

Throws: Nothing.

    template<class T>
    bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
31 Returns: atomic_compare_exchange_strong_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst).

```cpp
template <class T>
bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
```

```cpp
template <class T>
bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
```

32 Requires: p shall not be null and v shall not be null. The failure argument shall not be `memory_order::release` nor `memory_order::acq_rel`.

33 Effects: If *p is equivalent to *v, assigns w to *p and has synchronization semantics corresponding to the value of success, otherwise assigns *p to *v and has synchronization semantics corresponding to the value of failure.

34 Returns: true if *p was equivalent to *v, false otherwise.

35 Throws: Nothing.

36 Remarks: Two shared_ptr objects are equivalent if they store the same pointer value and share ownership. The weak form may fail spuriously. See 33.5.8.2 [atomics.types.operations].

7 Acknowledgements

Thanks to Michael Parks for the pandoc-based framework used to transform this document’s source from Markdown.

Thanks to Herb Sutter for first bringing this problem to the attention of WG21, along with the proposed solution, a decade ago!

8 References

https://wg21.link/n4944

https://wg21.link/p2139r2