Document number: P4221R2
Date: 2026-06-10
Project: Programming Language C++, SG1
Authors: Maged M. Michael, Paul McKenney, Michael Wong
Email: maged.michael@gmail.com, paulmck@kernel.org, fraggamuffin@gmail.com

Atomic Compare

Table of Contents

History **2**
Introduction **3**
Motivation 3
Equality Comparison in the Standard 3
New Capabilities 4
Relationship to compare_exchange 4
Preview **5**
Usage Examples 6
Proposed Wording **7**

History

R1

R1 was reviewed by SG1 in Brno with the following feedback:

> Poll: SG1 likes the P4221R1 design direction and would like to see this again with following changes:
>
> - compare either removed, or include an extended discussion on the value of compare and its potential pitfalls on provenance due to expected not being updated; potentially update compare to take expected by reference and update its value instead.
> - discussion on memcmp to address whether this is adding a new capability to the standard that was not possible before, or just adding new APIs to make something that's already possible with memcmp more ergonomic.
> - discussion on padding for the new APIs and the impacted examples
> - add C free functions and atomic_ref support

R2

R2 addresses SG1 feedback by:

- Removing **`compare`**.
- Clarifying motivation.
- Discussing equality comparison in the standard, including the effect of padding and **`memcmp`** capabilities.
- Stating the new capabilities provided by the proposed function.
- Completing the wording, including free functions, **`atomic_ref`,** and specializations.

Introduction

This paper proposes adding a **`compare_load`** member function to **`std::atomic<T>`** and **`std::atomic_ref<T>`**, alongside corresponding free functions. This function performs an atomic comparison of the atomic object's value with an expected value, following the same value representation comparison semantics as **`compare_exchange_strong`**. It updates the expected argument on failure, operating entirely as a pure read without writing a new value to the atomic object.

Motivation

Standard C++ currently lacks a mechanism to perform a consistent, read-only, padding-independent value representation equality check on an atomic object.

A secondary motivation is to provide programmers with distinct memory orders for comparison success and failure. Currently, programmers must either commit upfront to a single memory order regardless of the evaluation outcome, or navigate the complex semantics of standalone fences.

The proposed **`compare_load`** function natively resolves both issues, providing a read-only, padding-independent value representation equality check with built-in support for outcome-dependent memory ordering.

Equality Comparison in the Standard

To clarify the exact semantics evaluated by **`compare_load`**, it is necessary to distinguish the three forms of equality comparison in C++ (see **[basic.types.general] p4**):

- **Semantic Equality (`operator==`):** Evaluates logical equivalence. It inherently ignores padding bits and can be arbitrarily customized via user-defined overloads.
- **Object Representation (`memcmp`):** Evaluates bitwise equality of the entire memory footprint of an object, including padding bits.
- **Value Representation:** Evaluates bitwise equality of only the bits that participate in representing the value, excluding any padding bits.

These distinct definitions result in behavioral differences when writing concurrent code:

- **`operator==`** can yield different results from both **`memcmp`** and value representation, even when a type has zero padding.
- **`memcmp`** is functionally identical to a value representation comparison only for types *without* padding. For types *with* padding, **`memcmp`** diverges and becomes an unsafe proxy for value equality due to the potential for indeterminate padding bits.
- Currently, **`compare_exchange`** is the only mechanism in the standard library capable of consistently evaluating value representation equality in the presence of padding, but it couples this evaluation to a mutating memory write.

New Capabilities

The introduction of **`compare_load`** provides fundamental concurrency capabilities that cannot be achieved by combining existing standard library facilities:

- **Read-only padding-neutral value representation comparison:** Unlike **`operator==`** (which relies on semantic logic), **`memcmp`** (which unsafely evaluates indeterminate padding bits), and **`compare_exchange`** (which incurs a hardware write), **`compare_load`** provides a consistent, read-only mechanism to evaluate value representation equality.
- **Outcome-dependent memory ordering:** Unlike a standard atomic **`load`** followed by a nonatomic **`memcmp`** comparison, which requires a programmer to commit to a single memory order upfront or navigate complicated standalone fences, **`compare_load`** allows programmers to express distinct memory orders for success and failure, optimizing performance based on the evaluation outcome.

Relationship to **`compare_exchange`**

The existing **`compare_exchange`** operations, **`compare_exchange_weak`** and **`compare_exchange_strong`**, defined in **[atomics.types.operations] p21–28**, provide the semantic basis for the proposed **`compare_load`** function. Specifically, they establish:

- **Value representation** comparison (p23, p28).
- Handling of **padding bits** (p25, p28).
- Update of the **`expected`** argument on comparison failure (p23).
- Categorizing the operation as an atomic load upon comparison failure (p23).

Note that the rationale for a weak variant (p27) is inapplicable to **`compare_load`**, as it operates strictly as a read-only evaluation without a corresponding conditional store.

Furthermore, **`compare_load`** strictly follows the pointer provenance semantics of **`compare_exchange`**. The design intent is that any future standard modifications regarding pointer provenance or "zap" fixes applied to the specification of **`compare_exchange`** will apply equally and systematically to **`compare_load`**.

Preview

We propose adding a **`compare_load`** member function to **`std::atomic<T>`** and **`std::atomic_ref<T>`**, along with corresponding non-member functions.

compare_load

```
bool compare_load(T& expected, memory_order success, memory_order failure)
    const volatile noexcept;
constexpr bool compare_load(T& expected, memory_order success, memory_order failure)
    const noexcept;
bool compare_load(T& expected, memory_order order = memory_order::seq_cst)
    const volatile noexcept;
constexpr bool compare_load(T& expected, memory_order order = memory_order::seq_cst)
    const noexcept;
```

Note that **`compare_load`**:

- Performs value representation equality check between the value of the atomic object and the value of **`expected`**.
- Supports the same strong comparison semantics as **`compare_exchange_strong`**.
- Supports the same memory order constraints as a **`load`** operation but with optional distinct orders for comparison success and failure.
- Updates the **`expected`** argument with the value read from the atomic object if the equality comparison fails, as with the failure case of **`compare_exchange`**.

Usage Examples

The following table shows code snippets using the current standard C++ (on the left) and using the proposed atomic **`compare_load`** function (on the right).

Current Standard C++	With Proposed Function

/// EX1 Load-compute-validate loop
T value = a.load();
while (true) {
  // Calculation based on read value
  R result = calc(value);
  // Manually reload current value
  T current = a.load();
  // Check validity of used value
  if (current == value)
  return result;
  // Update value manually
  value = current;
}
	/// EX1 Load-compute-validate loop
	T value = a.load();
	while (true) {
	  // Calculation based on read value
	  R result = calc(value);
	
	  // Check validity of used value
	  if (a.compare_load(value))
	  return result;
	  // Failed compare updates value
	  // automatically.
	}

/// EX2 HP TRY_PROTECT
bool try_protect(T*& ptr, atomic<T*>& src)
{
  T* p = ptr;
  hp_.reset_protection(ptr);
  /* Light asymmetric memory barrier */
  ptr = src.load(acquire);
  if (p == ptr)
  return true;
  else {
  hp_.reset_protection(); return false;
  }
}
	/// EX2 HP TRY_PROTECT
	bool try_protect(T*& ptr, atomic<T*>& src)
	{
	
	  hp_.reset_protection(ptr);
	  /* Light asymmetric memory barrier */
	
	  if (src.compare_load(ptr, acquire))
	  return true;
	  else {
	  hp_.reset_protection(); return false;
	  }
	}

/// EX3 Outcome-dependent memory ordering
T observed = a.load(relaxed);
if (observed == value) {
 	
  std::atomic_thread_fence(seq_cst);
  /* success path */
} else {
  std::atomic_thread_fence(acquire);
  value = observed;
  /* failure path */
}
	/// EX3 Outcome-dependent memory ordering
	
	if (a.compare_load(
	  value, seq_cst, acquire)) {
	  // No need for explicit fence
	  /* success path */
	} else {
	  // No need for explicit fence
	
	  /* failure path */
	}

Proposed Wording

In **17.3.2** [version.syn] **p2**, add the following macro to the list in alphabetical order:

```
__cpp_lib_atomic_compare_load 202XXXXXL // freestanding, also in <atomic>
```

In **32.5.2** [atomics.syn], add the following to namespace **std** after the **atomic_compare_exchange** declarations:

```
template<class T>
  bool atomic_compare_load(
    volatile atomic<T>*, typename atomic<T>::value_type*) noexcept;
template<class T>
  constexpr bool atomic_compare_load(
    atomic<T>*, typename atomic<T>::value_type*) noexcept;
template<class T>
  bool atomic_compare_load_explicit(volatile atomic<T>*,
    typename atomic<T>::value_type*, memory_order, memory_order) noexcept;
template<class T>
  constexpr bool atomic_compare_load_explicit(atomic<T>*,
    typename atomic<T>::value_type*, memory_order, memory_order) noexcept;
```

In **32.5.7.1** [atomics.ref.generic.general], add the following to the **atomic_ref<T>** class template synopsis after **compare_exchange**:

```
constexpr bool compare_load(value_type&, memory_order, memory_order) const noexcept;
constexpr bool compare_load(value_type&, memory_order = memory_order::seq_cst)
    const noexcept;
```

In **32.5.7.2** [atomics.ref.ops], add the following after **compare_exchange**:

```
constexpr bool compare_load(T& expected, memory_order success, memory_order failure)
    const noexcept;
constexpr bool compare_load(T& expected, memory_order order = memory_order::seq_cst)
    const noexcept;
```

Preconditions: **success** and **failure** are each one of **memory_order::relaxed**, **memory_order::acquire**, or **memory_order::seq_cst**.

Effects: Retrieves the value in **expected**. It then atomically compares the value representation of the value pointed to by ***ptr** for equality with that previously retrieved from **expected**. If and only if the comparison is true, memory is affected according to the value of **success**, and if the comparison is false, memory is affected

according to the value of **`failure`**. When only one **`memory_order`** argument is supplied, the value of **`success`** is **`order`**, and the value of **`failure`** is **`order`**. If and only if the comparison is false then, after the atomic operation, the value in **`expected`** is replaced by the value pointed to by ***`ptr`** during the atomic comparison. These operations are atomic load operations on the memory pointed to by ***`ptr`**.

Returns: The result of the comparison.

In **32.5.7.3** **[atomics.ref.int]**, add the following after **`compare_exchange`**:

```
constexpr bool compare_load(value_type&, memory_order, memory_order) const noexcept;
constexpr bool compare_load(value_type&, memory_order = memory_order::seq_cst)
    const noexcept;
```

In **32.5.7.4** **[atomics.ref.float]**, add the following after **`compare_exchange`**:

```
constexpr bool compare_load(value_type&, memory_order, memory_order) const noexcept;
constexpr bool compare_load(value_type&, memory_order = memory_order::seq_cst)
    const noexcept;
```

In **32.5.7.5** **[atomics.ref.pointer]**, add the following after **`compare_exchange`**:

```
constexpr bool compare_load(value_type&, memory_order, memory_order) const noexcept;
constexpr bool compare_load(value_type&, memory_order = memory_order::seq_cst)
    const noexcept;
```

In **32.5.8.1** **[atomics.types.generic.general]**, add the following to the **`atomic<T>`** class template synopsis after **`compare_exchange`**:

```
bool compare_load(T&, memory_order, memory_order) const volatile noexcept;
constexpr bool compare_load(T&, memory_order, memory_order) const noexcept;
bool compare_load(T&, memory_order = memory_order::seq_cst) const volatile noexcept;
constexpr bool compare_load(T&, memory_order = memory_order::seq_cst) const noexcept;
```

In **32.5.8.2** **[atomics.types.operations]**, add the following after **`compare_exchange`**:

```
bool compare_load(T& expected, memory_order success, memory_order failure)
    const volatile noexcept;
constexpr bool compare_load(T& expected, memory_order success, memory_order failure)
    const noexcept;
bool compare_load(T& expected, memory_order order = memory_order::seq_cst)
    const volatile noexcept;
constexpr bool compare_load(T& expected, memory_order order = memory_order::seq_cst)
```

```
    const noexcept;
```

Constraints: For the volatile overload of this function, **`is_always_lock_free`** is true.

Preconditions: **`success`** and **`failure`** are each one of **`memory_order::relaxed`**, **`memory_order::acquire`**, or **`memory_order::seq_cst`**.

Effects: Retrieves the value in **`expected`**. It then atomically compares the value representation of the value pointed to by **`this`** for equality with that previously retrieved from **`expected`**. If and only if the comparison is true, memory is affected according to the value of **`success`**, and if the comparison is false, memory is affected according to the value of **`failure`**. When only one **`memory_order`** argument is supplied, the value of **`success`** is **`order`**, and the value of **`failure`** is **`order`**. If and only if the comparison is false then, after the atomic operation, the value in **`expected`** is replaced by the value pointed to by **`this`** during the atomic comparison. These operations are atomic load operations on the memory pointed to by this.

Returns: The result of the comparison.

In **32.5.8.2 [atomics.types.operations] p28**, modify Note 6, Note 7, and Note 8 as follows:

[*Note 6*: Under cases where the memcpy and memcmp semantics of the compare-and-exchange and **`compare_load`** operations apply, the comparisons can fail for values that compare equal with **`operator==`** if the value representation has trap bits or alternate representations of the same value. ... -- *end note*]

[*Note* 7: Because compare-and-exchange and **`compare_load`** act~~s~~ on an object's value representation, padding bits that never participate in the object's value representation are ignored. ... -- *end note*]

[*Note* 8: For a union with bits that participate in the value representation of some members but not others, compare-and-exchange **`and compare_load`** might always fail. This is because such padding bits have an indeterminate value when they do not participate in the value representation of the active member. ... -- *end note*]

In **32.5.8.3 [atomics.types.int]**, add the following after **`compare_exchange`**:

```
bool compare_load(integral-type&, memory_order, memory_order) const volatile noexcept;
constexpr bool compare_load(integral-type&, memory_order, memory_order) const noexcept;
bool compare_load(integral-type&, memory_order = memory_order::seq_cst) const volatile
    noexcept;
constexpr bool compare_load(integral-type&, memory_order = memory_order::seq_cst) const
    noexcept;
```

In **32.5.8.4 [atomics.types.float]**, add the following after **`compare_exchange`**:

```
bool compare_load(floating-point-type&, memory_order, memory_order) const volatile
```

```
    noexcept;
constexpr bool compare_load(floating-point-type&, memory_order, memory_order) const
    noexcept;
bool compare_load(floating-point-type&, memory_order = memory_order::seq_cst) const
    volatile noexcept;
constexpr bool compare_load(floating-point-type&, memory_order = memory_order::seq_cst)
    const noexcept;
```

In **32.5.8.5** [atomics.types.pointer], add the following after **compare_exchange**:

```
bool compare_load(T*&, memory_order, memory_order) const volatile noexcept;
constexpr bool compare_load(T*&, memory_order, memory_order) const noexcept;
bool compare_load(T*&, memory_order = memory_order::seq_cst) const volatile noexcept;
constexpr bool compare_load(T*&, memory_order = memory_order::seq_cst) const noexcept;
```