Document number: P4221R0
Date: 2026-05-11
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

Introduction

This paper proposes adding **`compare`** and **`compare_load`** member functions to **`std::atomic<T>`**. These functions perform an atomic comparison of the atomic object's value with an expected value, following the same bitwise comparison semantics as **`compare_exchange_strong`**, but without writing a new value to the atomic object. The **`compare_load`** variant also updates the expected argument on failure. This paper is ready for review by SG1 in Brno 2026.

Motivation

The primary motivation is to provide dedicated operations that express intent more clearly than an atomic **`load`** followed separately by a manual comparison of non-atomic values. By providing dedicated member functions, the API allows the programmer to specify equality checks directly, supporting both pure predicates and integrated updates of the expected argument.

This approach offers higher semantic precision by establishing clear boundaries for information flow. While **`compare`** confirms equality without exposing the underlying bits to the local thread context, **`compare_load`** provides a structured update for retry patterns. Both functions provide a cleaner foundation for human reasoning and formal analysis of concurrent logic.

Atomic bitwise equality checks offer concurrent programmers an alternative to raw-pointer equality checks (==), consistent with compare_exchange equality check semantics.

Background

The existing **`compare_exchange`** operations, **`compare_exchange_weak`** and **`compare_exchange_strong`**, are defined in **[atomics.types.operations] p21–28**.

These operations provide the semantic basis for the proposed **`compare`** and **`compare_load`** functions. Specifically, they establish:

- Bitwise comparison based on the **value representation** (p23, p28).
- Update of the **`expected`** argument on comparison failure (p23).
- Categorizing the operation as an atomic load upon comparison failure (p23).
- Rationale for weak variant (p27).
- Handling of padding bits (p25, p28).

Proposed Functions

Overview

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

`compare`

```
constexpr bool compare(T expected, memory_order order = memory_order::seq_cst)
  const noexcept;
bool compare(T expected, memory_order order = memory_order::seq_cst)
  const volatile noexcept;
```

`compare_load`

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

Notes:

- Both **`compare`** and **`compare_load`**:
  - Perform bitwise equality check between the value of the atomic object and the value of **`expected`**.
  - Support the same strong comparison semantics as **`compare_exchange_strong`**.
  - Support the same memory order constraints as a load operation.
- **`compare`** only:
  - Is a pure predicate; the **`expected`** argument is passed by value and remains unchanged. It does not leak the current value if different from **`expected`**.
- **`compare_load`** only:
  - 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 standard C++ atomics (on the left) and using the proposed atomic **`compare`** and **`compare_load`** functions (on the right).

Current Standard C++	With Proposed Functions

`/// EXAMPLE 1`
`// Read ID of current owner`
`ID observed = owner.load();`
`if (observed == my_id) {`
`  perform_owner_action();`
`}`
`// If not owner, local scope holds the`
`// other ID (semantic leak).`
`/* ... more code ... */`
	`/// EXAMPLE 1`
	
	`// Pure equality check predicate`
	`if (owner.compare(my_id)) {`
	`  perform_owner_action();`
	`}`
	`// If not owner, local scope does not hold`
	`// the other ID (no semantic leak).`
	`/* ... more code ... */`

`/// EXAMPLE 2`
`T current = a.load();`
`while (true) {`
`  // Calculation based on observed value`
`  R result = calc(current);`
`  // Manually reload current state`
`  T observed = a.load();`
`  // Check validity of current state`
`  if (observed == current)`
`    return result;`
`  // Update current for next attempt`
`  current = observed;`
`}`
	`/// EXAMPLE 2`
	`T current = a.load();`
	`while (true) {`
	`  // Calculation based on observed value`
	`  R result = calc(current);`
	
	`  // Check validity of current state`
	`  if (a.compare_load(current))`
	`    return result;`
	
	`  // Failed compare updates current`
	`}`

`/// EXAMPLE 3: 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(mo_acq);`
`  if (p == ptr)`
`    return true;`
`  else {`
`    hp_.reset_protection();`
`    return false;`
`  }`
`}`
	`/// EXAMPLE 3: HP TRY_PROTECT`
	`bool try_protect(T*& ptr, atomic<T*>& src)`
	`{`
	
	`  hp_.reset_protection(ptr);`
	`  /* Light asymmetric memory barrier */`
	
	`  if (src.compare_load(ptr, mo_acq))`
	`    return true;`
	`  else {`
	`    hp_.reset_protection();`
	`    return false;`
	`  }`
	`}`

Notes:

- While **`'if (owner.load() == my_id)'`** avoids naming a local variable, it is fragile under refactoring, for instance, when adding debug logging. In contrast, **`'compare(my_id)'`** provides a primitive that is structurally incapable of leaking the value, making the intent remain intact regardless of local code changes.
- Concurrent code on the right avoids raw-pointer equality checks (==).

Proposed Wording

In **17.3.2 [version.syn] p2**, add the following:

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

Under **32.5 [atomics]**:

The following proposed wording is relative to the C++26 Draft International Standard (DIS) **[atomics]**. It provides the complete semantics for the primary **`std::atomic`** template. The semantics for **`std::atomic_ref`**, all specializations (integral, pointer, and floating-point), and corresponding non-member functions are intended to be identical.

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

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

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

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

```
constexpr bool compare(T expected, memory_order order = memory_order::seq_cst)
  const noexcept;
bool compare(T expected, memory_order order = memory_order::seq_cst)
  const volatile noexcept;
constexpr bool compare(T expected, memory_order success, memory_order failure)
  const noexcept;
bool compare(T expected, memory_order success, memory_order failure)
  const volatile 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: Atomically compares the value representation of the value pointed to by **`this`** for equality with that of **`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`. These operations are atomic load operations on the memory pointed to by **`this`**.

Returns: The result of the comparison.

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

[Editor's note: Add the declarations for **`compare`** and **`compare_load`** to the synopses of all specializations in **[atomics.types.int]**, **[atomics.types.pointer]**, and **[atomics.types.float]**, and the corresponding subsections of **[atomics.ref]**. Add corresponding non-member function templates to the synopsis in **[atomics.syn]**.]