Constness and locking

Document #:	P3703R0
Date:	2025-05-18
Project:	Programming Language C++
Audience:	Concurrency Study Group

Reply-to: Yaodan Zhang
<katherinezh@uchicago.edu>
Alec Cepeda
Alexander Buzanis
Aushveen Vimalathas
Charlie Sabino
Cory Turnbaugh
Guanduo Mu
Hongli Zhao
Joneskim Kimo
Matthias Zajdela
Mitch Verhelle
Pierre-Yves Sojic
Steven Arellano
Roshan Surabhi
Wei Cai
Mike Spertus
<spertus@uchicago.edu>

Contents

1 Abstract **1**

2 Existing code **2**
2.1 GitHub findings: . . . 2
2.2 Experience from other languages . . . 2

3 Comparisons **3**

4 Arguments against **6**
4.1 Inadvertently enabling unsafe operations . . . 6
4.2 Programmers may incorrectly believe that const mutexes are ROMable . . . 6
4.3 ABI breakage . . . 6
4.4 Should this apply to just `shared_mutex`? . . . 6

1 Abstract

Class definitions commonly contain a `std::shared_mutex` or `std::mutex` field to support concurrent access to objects of that class. Such mutexes are inevitably declared `mutable` so that logically `const` methods can lock or unlock them—an idiom the C++11 community dubs the "Mutable comes with Mutex" **M&M Rule**[1]. In effect, the mutex protects the object's value rather than contributes to it, which can confuse newcomers and make `mutable` feel like a const-circumventing hack rather than a safety feature.

[1] See "Considering Threadability," in *C++ Best Practices*, lefticus.gitbooks.io: https://lefticus.gitbooks.io/cpp-best-practices/content/07-Considering_Threadability.html

Beyond mutexes, other standard library types hint at controlled interior mutation in `const` contexts: for example, `atomic<T>::load()` is always `const` even though it may acquire locks associated with the data (h/t Hans Boehm), and `shared_ptr(const shared_ptr&) noexcept` and `shared_ptr& operator=(const shared_ptr&) noexcept` modify a reference count under the hood while preserving a `const shared_ptr&` interface.

This paper investigates the implications of promoting mutex locking methods to `const` methods, examining how such a change interacts with existing `const`-correctness idioms and C++'s model of thread safety.

Before

```
struct BankAccount {
  double getBalance() const {
    shared_lock lck(mtx);
    return balance;
  }

  void setBalance(double d) {
    unique_lock lck(mtx);
    balance = d;
  }
  mutable shared_mutex mtx;
  double balance;
};
```

After

```
struct BankAccount {
  double getBalance() const {
    shared_lock lck(mtx);
    return balance;
  }

  void setBalance(double d) {
    unique_lock lck(mtx);
    balance = d;
  }
  shared_mutex mtx;
  double balance;
};
```

While simple, this change feels easier for typical programmers and we feel better aligns with intent. We also consider several variations, including its impacts on comparison operators.

2 Existing code

2.1 GitHub findings:

— `mutable` is widely used to avoid removing `const` qualifiers even when interior mutation is needed, which can be confusing for new C++ programmers and sometimes leads developers to remove `const` altogether (a dangerous workaround).

— There are 42.2k files that have declared `std::shared_mutex` on GitHub, and among which, 17.4k (41.2 %) declare with a `mutable` keyword, highlighting the prevalence of this pattern.[2]

— In a representative examination, we did not observe any examples where we felt making locking `const` would be harmful to the code (and some would be improved by eliminating the "dangerous workaround" mentioned above). We would be very interested in feedback from the committee on scenarios where the proposed changes would hurt code correctness or safety. See Arguments against for more on this.

2.2 Experience from other languages

2.2.1 Rust

In Rust, neither `Mutex::lock()` nor `RwLock::read()`/`write()` requires `&mut self`. You call them on an immutable reference (`&self`), and they perform the lock-state mutation entirely behind the scenes, concurring with our proposal that locking operations on mutexes should be regarded as const. We did not find any criticism regarding this as weakness of Rust.

[2] Empirical survey of public C++ repositories on GitHub (GitHub Code Search, May 2025).

2.2.2 Go

Go has no explicit notion of `const` methods, yet its `sync.Mutex` similarly treats the lock as an invisible, non-value field.

3 Comparisons

Early in the `<=>` design discussions there was considerable debate over whether the automatically generated comparisons should ignore `mutable` fields—precisely due to the intuition that `const` operations should not change the value of an object. In particular, it seems especially intuitive that locking an object to examine its value should not in itself change the value of the object. In this sense, mutexes do not contribute to the "value" an object but rather protects the object's value, existing purely for side-effects on thread-safety rather than any change in the value of the host object.

It was ultimately decided that the default spaceship operator should not exclude mutable objects as all members should be consistently compared [3]. We do not disagree with this decision but it does provide insight into how to handle the particular case of mutex comparison. Currently, mutexes have no comparison operators. The M & M rule indicates that mutex operations should not be regarded as changing the value of the object the mutex belongs to, in which case, defining the spaceship operator for mutex types to unconditionally return `std::strong_ordering::equal` would make the default spaceship operator for the containing class more usable.

[3] Sutter, "Consistent Comparison," 2017. http://wg21.link/p0515r3

Before	After

Before

```
class Account {
public:
  CustomerId = getCustomerId() const {
    shared_lock lck(mtx);
    return id;
  }

  void setCustomerId(CustomerId const &i) {
    unique_lock lck(mtx);
    id = i;
  }

  AccountType getAccountType() const {
    shared_lock lck(mtx);
    return type;
  }

  void setAccountType(AccountType t) {
    unique_lock lck(mtx);
    type = t;
  }

  double getBalance() const {
    shared_lock lck(mtx);
    return balance;
  }

  void setBalance(double d) {
    unique_lock lck(mtx);
    balance = d;
  }
  mutable shared_mutex mtx;
  std::strong_ordering
    operator<=>(const Account &a) const {
      if (auto cmp = id <=> a.id; cmp != 0)
        return cmp;
      if (auto cmp = type <=> a.type; cmp != 0)
        return cmp;
      return balance <=> a.balance;
  }

private:
  CustomerId id;
  double balance;
  AccountType type;
};

void f(Account &a, Account &b) {
  lock l(a, b);
  if (a == b) return;
    // Do stuff
}
```

After

```
class Account {
public:
  CustomerId = getCustomerId() const {
    shared_lock lck(mtx);
    returnid;
  }

  void setCustomerId(CustomerId const &i) {
    unique_lock lck(mtx);
    id = i;
  }

  AccountType getAccountType() const {
    shared_lock lck(mtx);
    return type;
  }

  void setAccountType(AccountType t) {
    unique_lock lck(mtx);
    type = t;
  }

  double getBalance() const {
    shared_lock lck(mtx);
    return balance;
  }

  void setBalance(double d) {
    unique_lock lck(mtx);
    balance = d;
  }

  shared_mutex mtx;
  std::strong_ordering
    operator<=>(const Account &a) const = default;

private:
  CustomerId id;
  double balance;
  AccountType type;
  std::strong_ordering
};

void f(Account &a, Account &b) {
  lock l(a, b);
  if (a == b) return;
    // Do stuff
}
```

Before	After

4 Arguments against

4.1 Inadvertently enabling unsafe operations

During the spaceship operator discussions, it was discussed whether it should be opt-in or always-on, with the decision that it should be opt-in to avoid implicitly creating unexpected behaviors on complex types but rather gating it with an explicit programmer choice. While our proposal provides support for classes with mutexes to leverage the default comparison operator, programmers still have to explicitly opt-in to using it, so the gate is still there.

Indeed, we would feel uncomfortable generating this comparison without an opt-in because examples like those above do require the programmer to lock the objects. However, we believe that if they explicitly request the comparison, having the additional inconvenience of writing out an equivalent comparison manually is less safe. Note that there is no breakage to existing code as the default equality operator does not currently work for classes with mutex members.

4.2 Programmers may incorrectly believe that const mutexes are ROMable

Some early reviewers mentioned that some programmers may erroneously confuse constness with ROM As this is already false for many types, such as atomics, classes with mutable members (including but not limited to mutexes), etc., we do not see much value in trying to draw a slightly narrower perimeter

4.3 ABI breakage

As constness is part of the signature of a method, the signature changes proposed in this paper could result in ABI breakage in existing code. We propose addressing this serious concern for existing mutex types by defining both `const` and non-`const` variants of the locking methods. (New mutex types would only need the `const` ones). This would still have the potential breakage in that `&mutex:lock` would become ambiguous in some circumstances.

Fortunately, code that would be impacted by this overload is both extremely rare and readily fixable. A GitHub search reveals 3 distinct files with `&std::mutex::lock`. However, two continue to work unchanged with this overload added because they the result of `&std::mutex::lock` is assigned to a specific function type that resolves the ambiguity, leaving only a single impacted file. The affected file is a Python wrapper for a C++ library that uses pybind11[4] to create a binding for `&std::mutex::lock`, which would need to be replaced with `static_cast<void (std::mutex::*)()>(&std::mutex::lock)`. This same fix could be applied to any other non-public files, which we expect would be similarly rare.

4.4 Should this apply to just `shared_mutex`?

While our proposal grew out of observing that `std::shared_mutex` is overwhelmingly used for "logical reads" in `const` methods, the underlying rationale is far more general: any mutex that serves solely to protect data rather than contribute to it carries no observable value. In C++ today, you must mark a `std::mutex` (or any of its variants) `mutable` in order to lock it from a `const` context—yet this is purely a technical workaround, not a reflection of logical state changes.

[4] https://github.com/pybind/pybind11