

Document: P2034R4
Author: Ryan McDougall <mcdougall.ryan@gmail.com>
Nestor Subiron Montoro <nsubiron@gmail.com>
Audience: EWG-I
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Partially Mutable Lambda Captures

Or

A More Uniform Const for Lambdas

Revision History

Changes from R3:

- Meta-motivation: safety and security – const should be easier to get right and harder to get wrong.
- Cleaned up some examples.

Changes from R2:

- Update author email addresses.
- Rename `any_invocable` to `move_only_function`.

Changes from R1:

- Add discussion of const captures on move construction and assignment.
- Add vocabulary type ``as_mutable``.
- Add alternative implementation strategy for const members.
- Selective move feature in top section.

Changes from R0: [Concerns from EWG-I](#)

- Interactions with `this` pointer.
- Interactions with init-capture packs.
- Clarify const as it applies to pointers.
- Add const-reference use case.
- Expanded prose.

Background

Lambdas were introduced in [N2550](#), and while [previous](#) drafts considered mutable capture by value, the original wording left captures entirely const. [N2658](#) salvaged mutable for *all* captures by allowing the `mutable` keyword to modify the call.

[P0288](#) (`move_only_function`) was approved by LEWG, and a central improvement is that it respects the `const` modifier on function types (ie. `move_only_function<void(int) const>`). This means an `move_only_function` with a `const` modifier on its call type will only bind to lambdas that are not marked `mutable`.

A type that is “[logically const](#)” is a type that has some mutable members that do not fundamentally change the invariants of the object, even when it is `const`. This means `move_only_function`, and *any* other `const`-correct library, *cannot* work with logically `const` lambdas.

Meta-Motivation

The proposal and most extensions would allow programmers to **apply `const` with simplicity and precision** to lambda captures – improving applicability of `const` in cases where programmers would otherwise:

1. Declare the lambda blanket `mutable`.
2. Declare captures by `const {non-}propagating wrapper`.

Applying `const` with more purpose and simpler syntax would improve the safety and security of such code – especially for programmers that have learned about the `const` declarations, but are not yet comfortable with `const-{non-}propagating wrappers`. Avoiding use of wrappers also makes lambda captures smaller and thus easier to read and reason about.

Motivation

Type erased callables like `std::move_only_function` are the backbone of most asynchronous systems. Users of such systems close their operations in lambdas and place them in a concurrent queue to be processed elsewhere. Performance is often key in such systems, and such operations may want its own local reusable scratch memory. Or perhaps an accumulator for hysteresis over multiple calls.

```
struct MyRealtimeHandler {
    Callback callback_;
    State state_;
    mutable Buffer accumulator_;

    void operator()(Timestamp t) const {
```

```

        callback_(state_, accumulator_, t);
    }
};

```

```

concurrent::queue<move_only_function<void(Timestamp) const> queue;
queue.push(MyRealtimeHandler{f, s});

```

Lambdas in such cases require work-arounds, such as abandoning logical const correctness, abandoning ownership, or introducing intermediary {non-}const-propagating intermediary types. Strict ownership rules are important due the asynchronous nature of the handler, and const correctness is important for memory- and thread-safety.

Proposal

Mutable Capture By Value

Allow [lambda capture initialization](#) to be `mutable` qualified, as below. This would have the effect of declaring the captured variable to be mutable.

```

auto a = [mutable x, y]() {};

```

// **equivalent to:**

```

struct A {
    mutable X x;
    Y y;
    void operator() () const {}
};

```

Before	After
<pre> struct A { const State state; mutable Buffer buf; void operator() () const { // ... } }; // manual bespoke type move_only_function<void() const> f = A{s, b}; </pre>	<pre> move_only_function<void() const> f = [s, mutable b] { // ... }; </pre>
<pre> template <typename T> class as_owned_mutable { mutable T value; public: T& ref() const { return value; } }; </pre>	

<pre> } }; // new vocabulary type move_only_function<void() const> f = [s, b = as_owned_mutable<Buffer>{}] () { auto& buffer = b.ref(); // ... }; </pre>	<pre> move_only_function<void() const> f = [s, mutable b] { // ... }; </pre>
<pre> // loss of const correctness move_only_function<void()> f = [s, b]() mutable { // ... }; </pre>	<pre> move_only_function<void() const> f = [s, mutable b] { // ... }; </pre>
<pre> // loss of ownership move_only_function<void() const> f = [s, buf_ptr = &b]() { // ... }; </pre>	<pre> move_only_function<void() const> f = [s, mutable b] { // ... }; </pre>

Selective Moves with init-capture Packs

Following the direction set out in [P2095](#), using the example in [P0780](#), we are able to move arguments from caller, to lambda, to callee -- without having to stop at the lambda:

```

template <class... Args>
auto delay_invoke_foo(Args... args, State s) {
    return [s, mutable ...args=std::move(args)] { // <-- new
        return foo(s, std::move(args)...);        // <-- improved
    };
}

```

Possible Extensions

Extensions are motivated by use cases, and listed in order of perceived usefulness -- however it should be noted that they also introduce increasing precision, consistency, and symmetry – which the authors believe is a justification in its own right.

1. Const Capture on Mutable Call Operator

If lambda capture initialization can be modified by `mutable` and lambda closure call can be modified by `mutable`, then lambda closure calls modified by `mutable` should be able to declare some of their captures `const` – an inversion of this paper's core proposal.

Value

If most of the values captured are mutable, but one should be `const`, then this variation would be shorter and more readable. The alternative is to simply leave otherwise `const` captures mutable, or to use `std::cref`. The former is less safe, and the latter may be undesirable because the lambda does not own the object referred to, which may create lifetime issues. Moreover it requires a more verbose assignment syntax.

Allowing `const` captures is ergonomic and simple.

Before	After
<pre>template <typename T> class as_owned_const { T value; public: const T& ref() const { return value; } }; // new vocabulary type move_only_function<void()> f = [s, b = as_owned_const<Buffer>{}] mutable { auto& buffer = b.ref(); // ... };</pre>	<pre>move_only_function<void() const> f = [s, const b] mutable { // ... };</pre>
<pre>// loss of const correctness move_only_function<void()> f = [s, b]() mutable { // ... };</pre>	<pre>move_only_function<void() const> f = [s, const b] mutable { // ... };</pre>
<pre>// loss of ownership move_only_function<void()> f = [s, buf = std::cref(b)]() mutable { // ... };</pre>	<pre>move_only_function<void() const> f = [s, const b] mutable { // ... };</pre>

Implementation

```
auto b = [x, const y]() mutable {};
```

```
// equivalent to:
```

```
struct B {
    X x;
    const Y y;
    void operator() () {}
};
```

```
};
```

A `const` member would make the lambda closure assignment operators deleted, but lambda closures with captures [already delete the copy assignment operator](#).

A `const` member would also cause the move constructor to be implemented via copy, potentially causing it non-noexcept, depending on the copy constructor of the `const` member.

We can avoid these problems with another implementation strategy by invoking “as-if”:

```
// equivalent to:  
  
struct B {  
    X x;  
    Y y;  
    void operator() () {  
        // as-if y was declared const  
    }  
};
```

2. Const Capture by Reference

Capture by reference is not implicitly `const`, as capture by value is. However there are situations where it would be useful to capture by `const` reference, such as when a read-only object is too large to copy – or as a novel means to create a read-only code block.

Value

The same effect can be achieved using `std::cref` and `std::as_const` – but must be manually applied to each captured variable – unlike the capture-all in the second example below. This represents a chance to miss a variable and lose the protection of `const`.

Before	After
<pre>move_only_function<void() const> f = [s, huge = std::cref(huge)] mutable { // ... };</pre>	<pre>move_only_function<void() const> f = [s, const& huge] mutable { // ... };</pre>
<pre>X a, b, c; ... { // manual wrapping std::as_const(a); std::as_const(b); std::as_const(c); // ... enter const context }</pre>	<pre>X a, b, c; ... [const &] { // ... const context }();</pre>

Implementation

```
auto b = [&x, const &y]() {};
```

```
// equivalent to:
```

```
struct B {  
    X &x;  
    const Y &y;  
    void operator() () const {}  
};
```

We could also invoke compiler magic using “as-if”

```
// equivalent to:
```

```
struct B {  
    X &x;  
    Y &y;  
    void operator() () {  
        // as-if y was declared const Y&  
    }  
};
```

3. Const Call Operator

For symmetry with the call operator of bespoke types, declaring the lambda const should not be an error.

```
auto c = [x]() const {};
```

```
// equivalent to:
```

```
struct C {  
    X x;  
    void operator() () const {}  
};
```

4. Const Capture on Const Call Operator

For symmetry and principle of least surprise, declaring a const capture of a const lambda should not be an error.

```
auto c = [const x]() {};
```

See Const Capture on Mutable Call Operator.

5. Mutable Capture on Mutable Call Operator

For symmetry and principle of least surprise, declaring a mutable capture of a mutable lambda should not be an error.

```
auto c = [mutable x]() mutable {};  
  
// equivalent to:  
  
struct C {  
    mutable X x;  
    void operator() () {}  
};
```

Benefits of Consistency and Symmetry

The core benefits of extensions 3, 4 and 5 is lower cognitive load for programmers learning C++, and principle of least surprise. We can teach why lambdas default the way they do, but lambdas should have consistent and symmetric vocabulary for teaching how lambdas transform into callable types under the hood.

Experienced users will also benefit from additional self-documentation, especially in critical reliability contexts where verbosity and redundancy are preferred. Users would declare the lambda `mutable` or `const` according to ideal or majority semantics, and some minority of capture initialization would be the opposite, as an exception.

Concerns

1. Move construction with const captures

Const members cannot be moved from effectively, and lambdas with const captures would silently inhibit the (potentially noexcept) move constructors of what it captures by value, in favor of their copy constructors. This is more than just a pessimization, it may turn a non-throwing move operation into a potentially throwing copy.

```
struct A {  
    std::string s;  
};  
static_assert(std::is_move_constructible_v<A>);  
static_assert(std::is_nothrow_move_constructible_v<A>);  
  
struct B {  
    const std::string s;  
};  
static_assert(std::is_move_constructible_v<B>);  
static_assert(!std::is_nothrow_move_constructible_v<B>);
```

While users can do this today with classes, the concern is this would be making it easier to do unwittingly via lambdas.

Whether the improved symmetry and teachability of const capture lambdas is worth the possible footgun, remains an open question.

2. Assignment operations with const captures

Const members cannot be assigned to, and lambdas with const captures would be creatable but not assignable. In practice it is rare to assign a lambda, and wrappers like `std::function` use assignment to replace the object.

3. East v. West Const

In both East or West-const, the const always appears before the identifier. This proposal does not change that.

4. Pointer to Const v. Const Pointer

Current lambda behavior mandates bitwise const, which is const-pointer (not pointer to const). This proposal seeks to continue and not to modify that rule.

```
auto c = [const x = ptr]() {
    *x = {};          // ok
    x = nullptr;     // error
};
```

5. Interactions with `this`

The keyword `this` is a prvalue expression, and is special cased with regard to lambda captures. As such, the meaning of `mutable this` and `const this` doesn't have obvious semantics -- or if we defined them may be hard to teach. We recommend these two combinations be disallowed until further experience is accrued.

Students will likely expect the following to compile (it would not):

```
struct A {
    void mutate() {}
    void test() const {
        [mutable this] {
            this->mutate();
        }();
    }
};
```

Whereas the following would compile and work:

```
struct B {
```

```
void mutate() {}  
};  
  
void test(B* that) {  
    [mutable that] {  
        that->mutate();  
        that = nullptr;  
    }();  
}
```

Recall const pointer lambda capture is *bitwise* const, which affects if the pointer itself can be modified. The `this` pointer can never be modified and so `mutable this` or `const this` would either be meaningless if bitwise const, or inconsistent if logically const.

The meaning of `mutable *this` and `const *this` is much clearer, but for the sake of consistency when teaching “this is special”, we recommend dis-allowing this form as well.

6. These extensions seem like a lot. Could traps be lurking?

Everything being proposed has a direct and consistent transformation into callable types that are *already allowed*. Consistency and symmetry improve the teachability of lambdas, and the defaults chosen for C++11 lambdas can be easily explained.

That said, this proposal is easily separable.

Thanks

Thanks Patrick McMichael for suggesting the idea. Thanks to Nevin Liber, Matt Calabrese for offering important corrections. Thanks to Nevin Liber, Davis Herring, Barry Revzin, and Victoria Tsai, for examples and suggestions. Thanks to Nestor Subiron Montoro for becoming second author.