P2034: Partially Mutable Lambda Captures

Document	P2034R6
Authors	Ryan McDougall <mcdougall.ryan@gmail.com>
	Lakshay Garg <lakshayg.xyz@gmail.com>
Audience	EWG
Project	ISO/IEC JTC1/SC22/WG21 14882: Programming Language – C++

Contents

Revision History

Changes from R5: EWG Discussion

– Incorporate extensions into the main proposal.
– Add discussion of capture defaults to the proposal.
– Rearranged some sections and updated links.
– Add wording for:
  – mutable captures
  – const-ref captures
  – const-ref capture-default
  – const specifier

Changes from R4: EWG Discussion

– Implementation experience.

Changes from R3: EWG-I Discussion

– 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.

Polls

2025-11 Kona, R5

We [EWG] encourage further work on this paper towards C++29.

SF	F	N	A	SA
21	27	5	0	0

Strong Consensus

2025-06 Sofia, R4

EWG encourages more work in the direction of Partially Mutable Lambda Captures.

SF	F	N	A	SA
1	10	4	2	1

Consensus

EWG encourages more work in the direction of Partially Mutable Lambda Captures, including extensions.

SF	F	N	A	SA
2	15	3	1	0

Stronger consensus

2024-03 Tokyo, R2

EWGI believes P2034R3 should include a `const` qualifier for lambda captures.

SF	F	N	A	SA
2	4	4	1	0

Barely consensus

Comment: motivation could be better.

EWGI believes P2034R3 is sufficiently well developed, EWGI forwards it to EWG.

SF	F	N	A	SA
3	7	0	0	0

Consenus

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 a `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

We propose a number of enhancements to the lambda syntax that simplify creating const-correct lambdas. In addition to const-correctness, these features improve the consistency and symmetry – which the authors believe is a justification in its own right.

The proposed enhancements are summarized below in the order of their perceived usefulness followed by a more detailed explanation for each of these items.

1. Mutable capture on const call operator
2. Const capture on mutable call operator
3. Const capture by reference
4. Const default capture
5. Const default capture by reference
6. Explicitly const call operator
7. Const capture on const call operator
8. Mutable capture on mutable call operator

Feature 1: Mutable Capture on Const Call Operator

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

`struct A {`
`  const State state;`
`  mutable Buffer buf;`
`  void operator()() const {`
`  // ...`
`  }`
`};`
 	
`// manual bespoke type`
`move_only_function<void() const> f = A{s, b};`
	`move_only_function<void() const> f =`
	`  [s, mutable b] {`
	`  // ...`
	`  };`

`template <typename T>`
`class as_owned_mutable {`
`  mutable T value;`
`public:`
`  T& ref() const {`
`  return value;`
`  }`
`};`
 	
`// new vocabulary type`
`move_only_function<void() const> f =`
`  [s, b = as_owned_mutable<Buffer>{}]() {`
`  auto& buffer = b.ref();`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, mutable b] {`
	`  // ...`
	`  };`

`// loss of const correctness`
`move_only_function<void()> f =`
`  [s, b]() mutable {`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, mutable b] {`
	`  // ...`
	`  };`

`// loss of ownership`
`move_only_function<void() const> f =`
`  [s, buf_ptr = &b]() {`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, mutable b] {`
	`  // ...`
	`  };`

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
  };
}
```

Feature 2: 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

`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();`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, const b] mutable {`
	`  // ...`
	`  };`

`// loss of const correctness`
`move_only_function<void()> f =`
`  [s, b]() mutable {`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, const b] mutable {`
	`  // ...`
	`  };`

`// loss of ownership`
`move_only_function<void()> f =`
`  [s, buf = std::cref(b)]() mutable {`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, const b] mutable {`
	`  // ...`
	`  };`

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
  }
};
```

Feature 3: 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.

Value

The same effect can be achieved using `std::cref` and `std::as_const` – but this syntax is intuitive, concise and improves symmetry of this proposal.

Before	After

`move_only_function<void() const> f =`
`  [s, huge = std::cref(huge)] mutable {`
`  // ...`
`  };`
	`move_only_function<void() const> f =`
	`  [s, const& huge] mutable {`
	`  // ...`
	`  };`

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&
  }
};
```

Feature 4: Const Default Capture

When capturing by value with `=`, the constness of the captured entities depends on the declaration of the captured entity and the presence of the mutable specifier. There is no way to express the intent of const capture for capture defaults.

```
int x = 42;
auto b = [const =] () {
    // x is captured as-if it was const
    x = 1; // error
}
```

Value

Declaring `[const =]` makes the read-only intent clear even on a mutable lambda, without requiring each capture to be individually annotated const. It is the default-capture analogue of "Const Capture on Mutable Call Operator".

Feature 5: Const Default Capture by Reference

Similarly to "Const Default Capture", the constness of entities captured by reference depends on the declaration of the entity. `[const &]` as a capture-default expresses the read-only intent clearly. Applying `std::cref` or `std::as_const` to each captured entity represents a chance to miss a variable and lose the protection of `const`. The capture-all does not post this issue. This can also be used as a novel means of creating read-only code blocks.

Before	After

`X a, b, c;`
`...`
`{`
`  // manual wrapping`
`  auto& c_a = std::as_const(a);`
`  auto& c_b = std::as_const(b);`
`  auto& c_c = std::as_const(c);`
`  // ... enter const context`
`}`
	`X a, b, c;`
	`...`
	`[const &] {`
	
	`  // ... const context`
	
	`}();`

Feature 6: Explicitly 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 {}
};
```

Feature 7: Const Capture by Value 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 by Value on Mutable Call Operator”.

Feature 8: 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 features 6, 7, and 8 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

East v. West Const

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

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
};
```

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.

Complexity of Implementation

Ville Voutilainen implemented the proposal along with the extensions proposed in P2034R5 in GCC with regression tests, and gave the following report.

> In general, the implementation was very straightforward, after discussing the approach with the maintainer, and coming to the conclusion that it's simply a matter of adjusting the types of the capture members of lambda for const, and the storage-class-specifier for mutable. The implementation effort was a matter of a single afternoon.

The implementation is available on GitHub and can be tested on Compiler Explorer.

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 Ville for the exploratory implementation! Thanks to Lakshay Garg for becoming the second author. Thanks to Daveed Vandevoorde for providing suggestions and feedback on the wording.

Proposed Wording

expr.prim.id.unqual

Change in expr.prim.id.unqual (7.5.5.2) paragraph 4

If
– the *unqualified-id* appears in a *lambda-expression* at program point P,
– the entity is a local entity or a variable declared by an *init-capture*,
– naming the entity within the *compound-statement* of the innermost enclosing *lambda-expression* of P, but not in an unevaluated operand, would refer to an entity captured ~~by copy~~ in some intervening *lambda-expression*, and
– P is in the function parameter scope, but not the *parameter-declaration-clause*, of the innermost such *lambda-expression E*,

then the type of the expression is ~~the type of a class member access expression naming the non-static data member that would be declared for such a capture in the object parameter of the function call operator of *E*.~~:
– the type of a class member access expression naming the non-static data member that would be declared for such a capture in the object parameter of the function call operator of *E* if some intervening *lambda-expression* captures the entity by copy,
– the type of the entity if all the intervening *lambda-expression*s capture the entity by non-const reference, or
– the const qualified type of the entity if all intervening *lambda-expression*s capture the entity by reference, and at least one captures the entity by const reference.

[*Note 3:* If *E* is not declared `mutable` and the entity is not captured mutably (expr.prim.lambda.capture) by *E*, the type of such an identifier will typically be `const` qualified. — *end note*]

expr.prim.lambda.general

Change in expr.prim.lambda.general (7.5.6.1)

lambda-specifier:
`consteval`
`constexpr`
`const`
`mutable`
`static`

Change in expr.prim.lambda.general (7.5.6.1) paragraph 4

A *lambda-specifier-seq* shall contain at most one of each *lambda-specifier* and shall not contain both `constexpr` and `consteval`. If the *lambda-declarator* contains an explicit object parameter, then no *lambda-specifier* in the *lambda-specifier-seq* shall be `const`, `mutable`, or `static`. The *lambda-specifier-seq* shall ~~not contain both `mutable` and `static`~~ contain at most one of `const`, `mutable`, or `static`. If the *lambda-specifier-seq* contains `static`, there shall be no *lambda-capture*.

expr.prim.lambda.closure

Add a note to expr.prim.lambda.closure (7.5.6.2) paragraph 7

The function call operator or operator template is a static member function of static member function template if the *lambda-expression*'s *parameter-declaration-clause* is followed by `static`. Otherwise, it is a non-static member function or member function template that is declared `const` if and only if the *lambda-expression*'s *parameter-declaration-clause* is not followed by `mutable` and the *lambda-declarator* does not contain an explicit object parameter. It is neither virtual nor declared `volatile`. Any *noexcept-specifier* or *function-contract-specifier* specified on a *lambda-expression* applies to the corresponding function call operator or operator template. An *attribute-specifier-seq* in a *lambda-declarator* appertains to the type of the corresponding function call operator or operator template. An *attribute-specifier-seq* in a *lambda-expression* preceding a *lambda-declarator* appertains to the corresponding function call operator or operator template. The function call operator or any given operator template specialization is a constexpr function if either the corresponding *lambda-expression*'s *parameter-declaration-clause* is followed by `constexpr` or `consteval`, or it is constexpr-suitable. It is an immediate function if the corresponding *lambda-expression*'s *parameter-declaration-clause* is followed by `consteval`.

[*Note*: The `const` *lambda-specifier* has no additional effect; the function call operator is declared `const` if and only if `mutable` and `static` are not specified, regardless of whether `const` is present. – *end note*]

expr.prim.lambda.capture

Change in expr.prim.lambda.capture (7.5.6.3)

lambda-capture:
capture-default
capture-list
capture-default, capture-list

capture-default:
`const`$_{opt}$ *&*
=

capture-list:
capture
capture-list, capture

capture:
simple-capture
init-capture

simple-capture:
`mutable`$_{opt}$ *identifier* …$_{opt}$
`const`$_{opt}$ *& identifier* …$_{opt}$
this
**this*

init-capture:
`mutable`$_{opt}$ …$_{opt}$ *identifier initializer*
`const`$_{opt}$ *&* …$_{opt}$ *identifier initializer*

Change in expr.prim.lambda.capture (7.5.6.3) paragraph 2

If a *lambda-capture* includes a *capture-default* that is ~~&~~not `=`, no ~~identifier in a~~ *simple-capture* of that *lambda-capture* shall ~~be preceded by &~~ begin with that *capture-default*. If a *lambda-capture* includes a *capture-default* that is `=`, each *simple-capture* of that *lambda-capture* shall be of the form "`&` *identifier* …$_{opt}$" , "`const &` *identifier* …$_{opt}$", "`this`", or "`* this`".

Change in expr.prim.lambda.capture (7.5.6.3) paragraph 6

An *init-capture* inhabits the lambda scope of the *lambda-expression*. An *init-capture* without ellipsis behaves as if it declares and explicitly captures a variables of the form "`auto` *init-capture* `;`" ignoring any leading `mutable` keyword, except that:

– if the capture is by copy (see below), the non-static data member declared for the capture and the variable are treated as two different ways of referring to the same object, which has the lifetime of the non-static data member, and no additional copy and destruction is performed, and
– if the capture is by reference, the variable's lifetime ends when the closure object's lifetime ends.

Change in expr.prim.lambda.capture (7.5.6.3) paragraph 10

An entity is *captured by copy* if
– it is implicitly captured, the *capture-default* is `=`, and the captured entity is not `*this`, or
– it is explicitly captured with a capture that is not of the form `this`, `&` *identifier* …$_{opt}$, `const &` *identifier* …$_{opt}$ ~~or~~, `&` …$_{opt}$ *identifier initializer* or `const &` …$_{opt}$ *identifier initializer*.

An entity captured by copy is said to be *captured mutably* if the *capture* begins with the `mutable` keyword.

For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member ~~is the referenced type if the entity is a reference to an object, an lvalue reference to the referenced function type if the entity is a reference to a function, or the type of the corresponding captured entity otherwise.~~corresponding to a captured entity of type `T` is:
– an lvalue reference to the referenced function type if the entity is a reference to a function,
– `std::remove_const_t<std::remove_reference_t<T>>` if the entity is captured mutably, or
– `std::remove_reference_t<T>` otherwise.

The data member is declared `mutable` if the entity is captured mutably. A member of an anonymous union shall not be captured by copy.

Change in expr.prim.lambda.capture (7.5.6.3) paragraph 12

An entity is *captured by reference* if it is implicitly or explicitly captured but not captured by copy. An entity captured by reference is *captured by const reference* if it is either explicitly captured with a `const &` capture, or it is implicitly captured and the *capture-default* is `const &`. It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference. If declared, such non-static data members shall be of literal type.

Change in expr.prim.lambda.capture (7.5.6.3) paragraph 13

An *id-expression* within the *compound-statement* of a *lambda-expression* that is an odr-use of a reference captured by reference refers to the entity to which the captured reference is bound and not to the captured reference. If the entity is captured by const reference, the type of such an id-expression is const-qualified.

Change in expr.prim.lambda.capture (7.5.6.3) paragraph 14

If a *lambda-expression* `m2` captures an entity and that entity is captured by an immediately enclosing *lambda-expression* `m1`, then `m2`'s capture is transformed as follows:

– If `m1` captures the entity by copy, `m2` captures the corresponding non-static data member of `m1`'s closure type; if `m1` is not `mutable` and the entity is not captured mutably, the non-static data member is considered to be const-qualified.
– If `m1` captures the entity by reference, `m2` captures the same entity captured by `m1`.