Const/Mutable Extended Lambda Captures

Document	P2034R7
Date	2026-06-02
Audience	EWG
Project	ISO/IEC JTC1/SC22/WG21 14882: Programming Language – C++

Authors	Ryan McDougall <mcdougall.ryan@gmail.com>
	Lakshay Garg <lakshayg.xyz@gmail.com>

GitHub Issue	https://wg21.link/P2034/github
Source	https://github.com/lakshayg/wg21/tree/main/p2034

Contents

1 Background 6
2 Initial Motivation: Const-correctness 6
3 Subsequent Motivation: Symmetry and Simplicity 8
4 Design 9
- 4.1 Summary 9
- 4.2 Const Lambdas 9
- 4.3 Mutable Capture By-copy 9
- 4.4 Const Capture By-copy 10
- 4.5 Mutable Capture By-reference 11
- 4.6 Const Capture By-reference 11
- 4.7 Capture Defaults 12
- 4.8 Captures of `this` 13
- 4.9 Deducing the NSDM Type 13
- 4.10 Implementation Experience 14

5 Concerns 15
- 5.1 Consequences of Const Members 15
- 5.2 Teaching Const Capture 16
- 5.3 East v. West Const 16
- 5.4 Pointer to Const v. Const Pointer 16
- 5.5 Static Call Operator 16

6 Lambdas Are Syntactic Sugar for Function Objects 16
- 6.1 Remaining Gaps 18

Thanks 19
Proposed Wording 19
- [expr.prim.id.unqual] 20
- [expr.prim.lambda.general] 20
- [expr.prim.lambda.closure] 21
- [expr.prim.lambda.capture] 21
- Feature-Test Macro 23

Bibliography 24

Revision History

Changes from R6: EWG Discussion

– Restructured the motivation into an initial const-correctness case and a subsequent symmetry-and-simplicity case.
– Added the "Lambdas Are Syntactic Sugar for Function Objects" argument, with standard, compiler, reflection, and implementation evidence.
– Committed const capture to a genuine `const` member (option 3) and unified the mutable and const NSDM type deduction.
– Added the capture-defaults matrix (`[const =]`, `[mutable =]`, `[const&]`).
– Specified const-reference capture as logical const, consistent with the unspecified representation of reference captures.
– Added "Consequences of a `const` Member" and "Teaching `const` Capture".
– TODO: complete the proposed wording for the capture-defaults and reference cases.

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

2026-03 Croydon, R6

P2034R6 should include default mutable captures: *Strong Consensus in favor*

SF	F	N	A	SA
3	28	4	0	0

P2034R6 should explore making const-capture equivalent to a const member: *Strong Consensus in favor*

SF	F	N	A	SA
8	25	1	2	0

Encourage more work in the direction of P2034R6: *Strong Consensus in favor*

SF	F	N	A	SA
14	26	2	0	0

2025-11 Kona, R5

We [EWG] encourage further work on this paper towards C++29: *Strong Consensus*

SF	F	N	A	SA
21	27	5	0	0

2025-06 Sofia, R4

EWG encourages more work in the direction of Partially Mutable Lambda Captures: *Consensus*

SF	F	N	A	SA
1	10	4	2	1

EWG encourages more work in the direction of Partially Mutable Lambda Captures, including extensions: *Stronger consensus*

SF	F	N	A	SA
2	15	3	1	0

2024-03 Tokyo, R2

EWGI believes P2034R3 should include a `const` qualifier for lambda captures: *Barely consensus* (Comment: motivation could be better)

SF	F	N	A	SA

2	4	4	1	0

EWGI believes P2034R3 is sufficiently well developed, EWGI forwards it to EWG: *Consensus*

SF	F	N	A	SA
3	7	0	0	0

1 Background

Lambdas were introduced in N2550, and while previous drafts (N2529) 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.

`std::move_only_function` (P0288, C++23), and since then `std::copyable_function` (P2548) and `std::function_ref` (P0792) in C++26, improved on `std::function` by respecting the `const` qualifier on their call signature (e.g. `move_only_function<void(int) const>`). A `const`-qualified call type binds only to lambdas that are not marked `mutable`.

A type that is "logically const" is a type that has some members that do not fundamentally change the invariants of the object when mutated, even when it is const.

Taken together, this means the above standard types, and *any* other const-correct callable library, *cannot* work with logically const lambdas in the current form.

2 Initial Motivation: Const-correctness

Type erased callables like those above are the backbone of most asynchronous systems. Users of such systems enclose 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 their 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 workarounds, such as abandoning logical const correctness, abandoning ownership, or introducing intermediary {non-}const-propagating types. Strict ownership rules are important due to the asynchronous nature of the handler, and const correctness is important for memory- and thread-safety.

However if we expand lambdas to allow mutable capture, then only the logically mutable non-static data members become mutable, and the rest of the captures can remain const. The idea is illustrated below.

Before	After

`struct A {`
`  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] {`
	`  // ...`
	`  };`

The proposal 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.

Alternatively, if most of the values captured are modifiable, but one should be `const`, then the following would be similarly shorter and more readable. The alternative is to simply leave otherwise const captures modifiable, 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.

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

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

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

Allowing `const` captures is ergonomic and simple.

3 Subsequent Motivation: Symmetry and Simplicity

Our initial motivation only needs a handful of combinations of `const` or `mutable` extensions to lambda syntax in order to meet its use cases – but in subsequent meetings EWG has expressed interest in symmetry and simplicity for its own sake, and asked the authors to investigate the design space.

There seems to be a commonly shared feeling that there is a simpler language hiding in C++, and that the lambda syntax should be orthogonal to all the other ways of declaring callable types, not a microcosm unto itself. The authors agree and have investigated all possible capture possibilities involving `const` or `mutable` captures and operator qualification.

4 Design

Some of the combinations of `const` or `mutable` that fall out of this design may not have very obvious uses, but this paper pursues symmetry and conceptual simplicity for its own sake.

4.1 Summary

At a high level: we have found that it is simple and straightforward to extend both `const` and `mutable` keywords to lambda syntax in a way that closely mirrors most users' mental model of lambda as "a means of getting an object of a callable struct", that is easy to implement, and is in line with the ongoing evolution of the language:

– By-copy captures can be prefixed by `const` or `mutable`, and this results in the non-static data member (NSDM) ([expr.prim.lambda.capture] paragraph 10) being declared as `const` or `mutable` respectively, and initialized with the expression it captures. Specializing a capture with `const` or `mutable` this way is an opt-in request with predictable behavior (that is the same as if they had declared the callable type manually).

– By-reference captures do not necessarily generate non-static data members (NSDM), and are unaffected by the call operator qualification due to the shallow propagation of `const`. `const&` captures have utility as read-only views, but `mutable` references do not exist.

4.2 Const Lambdas

A lambda's function call operator is already `const` (unless the lambda is declared `mutable` ([expr.prim.lambda.closure] paragraph 7)), but today this default cannot be spelled – we propose allowing it to be stated explicitly.

This introduces *no new behavior* – it is the existing default made explicit – but it is self-documenting and completes the symmetry with `mutable`: both qualifiers on the call operator can be written, just as both can now qualify a capture.

4.2.1 Syntax

```
[]() const {}  // identical to []() {}
```

For clarity we prefer to use the explicit syntax below.

4.3 Mutable Capture By-copy

We propose a new form of by-copy capture called "mutable capture", which allows lambda captures to be `mutable` qualified, as shown below. The standard mandates that by-copy captures create a non-static data member (NSDM) in the closure type ([expr.prim.lambda.capture] paragraph 10). A mutable capture, in addition to defining a NSDM, would have the effect of declaring it `mutable`.

4.3.1 Syntax

Capture Syntax	Description
`[mutable x]() const {}`	simple capture of x by copy; the NSDM is `mutable`

`[mutable x...]() const {}`	simple capture of pack x; each NSDM is `mutable`
`[mutable x = init]() const {}`	init-capture initialized from `init`; the NSDM is `mutable`
`[mutable ...xs = init]() const {}`	init-capture pack (P0780); each NSDM is `mutable`

4.3.2 Applicability

A mutable capture is permitted on a mutable lambda: it is well-formed, although is redundant in effect. Note however, a `[x]() mutable {}` is a different type than a `[mutable x]() mutable {}`.

	`() const`	`() mutable`

`[x]`	`struct X {`
	`  int x;`
	`  void operator()() const;`
	`};`
		`struct X {`
		`  int x;`
		`  void operator()();`
		`};`

`[mutable x]`	`struct X {`
	`  mutable int x;`
	`  void operator()() const;`
	`};`
		`struct X {`
		`  mutable int x;`
		`  void operator()();`
		`};`

4.3.2.1 Interaction with `consteval` and `constexpr` Lambdas

Reading a `mutable` member during constant evaluation is not a constant expression – the lvalue-to-rvalue conversion on a `mutable` subobject is disallowed ([expr.const]). That is a restriction on *use*, not on declaration: a `mutable` member is otherwise fine, so a `mutable` capture would not by itself make a `constexpr` or `consteval` lambda ill-formed. But rather than pin down exactly when such a member may be read during constant evaluation, we conservatively disallow the combination until experience is accrued.

4.4 Const Capture By-copy

We propose a new form of by-copy capture called “const capture”, which allows lambda captures to be `const` qualified, as shown below. The standard mandates that by-copy captures create a non-static data member (NSDM) in the closure type ([expr.prim.lambda.capture] paragraph 10). A const capture, in addition to defining a NSDM, would have the effect of declaring it `const`.

4.4.1 Syntax

Capture Syntax	Description
`[const x]() mutable {}`	simple capture of x by copy; the NSDM is `const`
`[const x...]() mutable {}`	simple capture of pack x; each NSDM is `const`
`[const x = init]() mutable {}`	init-capture initialized from `init`; the NSDM is `const`
`[const ...xs = init]() mutable {}`	init-capture pack (P0780); each NSDM is `const`

4.4.2 Applicability

A const capture is permitted on a const lambda: it is well-formed, although is redundant in effect. Note however, a `[x]() const {}` is a different type than a `[const x]() const {}`, and const members create other concerns: see Section 5.1.

	`() const`	`() mutable`

`[x]`	`struct X {`
	`  int x;`
	`  void operator()() const;`
	`};`
		`struct X {`
		`  int x;`
		`  void operator()();`
		`};`

`[const x]`	`struct X {`
	`  const int x;`
	`  void operator()() const;`
	`};`
		`struct X {`
		`  const int x;`
		`  void operator()();`
		`};`

4.5 Mutable Capture By-reference

We explicitly disallow capture of the form `[mutable& x]`. This is because `mutable` references are not permitted by the language. Note that this is not the same as mutable capture of a reference type:

```
T& x = ...;
auto f = [mutable x]() { }; // closure type gets a `mutable T x;` member
```

4.6 Const Capture By-reference

Capture by reference is not implicitly `const`, unlike capture by copy – the `const`-qualified call operator is "shallow" for pointers and references – meaning it ensures that you don't modify members, saying nothing of what they point to. For example

```
int i = 5;
int *p = &i;
auto l = [p] () const { *p = 0; }; // ok
auto x = [p] () const { p = nullptr; }; // error
```

The same holds for a reference capture: [expr.prim.lambda.capture] says an odr-use of a reference capture names the original entity, so there is no member for `const` to apply to – and the standard leaves it unspecified whether reference captures are represented as members at all ([expr.prim.lambda.closure]).

Capturing by `const` reference is nonetheless useful – for read-only access to an object too large to copy – and today requires `std::cref` or `std::as_const`, which is not as concise, intuitive, or discoverable as `const&`.

We therefore specify `[const& x]` by its semantics rather than as a `const` reference member: within the call operator `x` names a `const` lvalue reference, and any nested lambda that re-captures it observes that `const`. The usual reference-lifetime caveats apply.

Unlike the by-copy cases, `[const& x]` has the same effect on a `const` or `mutable` lambda: the referent is `const` within the body either way, since the `const` is on the reference rather than supplied by the call operator.

4.6.1 Syntax

Capture Syntax	Description
`[const& x]() mutable {}`	simple capture of `x` by `const` reference
`[const& x...]() mutable {}`	simple capture of pack `x`, each by `const` reference
`[const& x = init]() mutable {}`	init-capture binding a `const` reference to `init`
`[const& ...xs = init]() mutable {}`	init-capture pack (P0780), each binding a `const` reference

4.7 Capture Defaults

The capture-defaults `[=]` and `[&]` may be qualified by `const` or `mutable`, applying the qualifier to every implicitly-captured entity. For symmetry with the explicit reference default `[const&]`, the copy defaults are spelled with an explicit `=`:

	= (by copy)	& (by reference)
(unqualified)	`[=]`	`[&]`
`const`	`[const =]`	`[const&]`
`mutable`	`[mutable =]`	ill-formed

`[const =]` captures every implicitly-captured entity by `const` copy, `[mutable =]` by `mutable` copy, and `[const&]` by `const` reference. `[mutable&]` is ill-formed, since `mutable` references do not exist.

The current grammar admits only `&` and `=` as a *capture-default* ([expr.prim.lambda.capture]). We extend it:

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

capture-default-qualifier:
    `const`
    `mutable`

This is unambiguous: `const` and `mutable` are keywords, and the trailing `=` or `&` fixes the capture kind.

Following the C++20 deprecation of implicitly capturing `*this` under `[=]` ([depr.capture.this]), a qualified capture-default does not implicitly capture `*this`; it must be captured explicitly:

```
struct X {
  int x;
  void f() {
    auto a = [mutable =] { return x; };        // error: *this is not captured
implicitly
    auto b = [mutable =, this] { return x; };  // OK: this captured explicitly
  }
};
```

A qualified capture-default applies its qualifier only to the entities it captures implicitly; an explicitly-listed `this` or `*this` is captured by its own rules and is unaffected by the default's qualifier. Combined

with Section 4.8 – where `this` and `*this` may not themselves be qualified – this fixes every combination: `[mutable =, this]`, `[const =, *this]`, `[const&, *this]`, and so on capture `this`/`*this` normally, while `[mutable =, const this]` and the like are ill-formed.

4.8 Captures of `this`

A capture may name either `this` or `*this`, and the two capture differently. `[this]` captures the enclosing object *by reference*: the closure conceptually holds the pointer, though the standard leaves it unspecified whether a member is actually declared for it ([expr.prim.lambda.closure]). `[*this]` captures the object *by copy*, declaring an unnamed non-static data member of the enclosing class type ([expr.prim.lambda.capture] paragraph 10). We recommend disallowing `const` and `mutable` on all four spellings – `[const this]`, `[mutable this]`, `[const *this]`, and `[mutable *this]` – until experience is accrued.

For `[const this]` and `[mutable this]`, recall that capture is bitwise `const`: a qualifier names the captured pointer, not its pointee. But `this` is a prvalue ([expr.prim.this]) captured by reference, so there is no by-copy member for the qualifier to attach to; and the captured pointer can never be reassigned, so the qualifier is either meaningless (under the bitwise rule) or inconsistent with it (if read as qualifying `*this`).

For `[const *this]` and `[mutable *this]` the obstacle is mechanical rather than semantic. Because `*this` is captured by copy, the qualifier *would* carry the same meaning it has on any by-copy capture: a `const` or `mutable` copy of the object. But that member is unnamed and reached implicitly through `this` – each odr-use of `*this` is rewritten to refer to it ([expr.prim.lambda.capture]) – rather than through a named *id-expression*, so the const-propagation wording this paper threads through [expr.prim.id.unqual] paragraph 4 and the nested re-capture rule ([expr.prim.lambda.capture] paragraph 14) does not reach it. Rather than special-case that machinery for a capture with no demonstrated demand, we defer it.

4.9 Deducing the NSDM Type

A by-copy capture requires a non-static data member ([expr.prim.lambda.capture] paragraph 10); the question is its type. The two existing capture forms deduce it by different rules.

A *simple-capture* keeps the captured entity's type, retaining its cv-qualifiers: the member type is the referenced type if the entity is a reference to an object, an lvalue reference to the referenced function type if it is a reference to a function, and the entity's type otherwise ([expr.prim.lambda.capture] paragraph 10). Capturing a `const T` therefore yields a `const` member – a deliberate choice (CWG756), so that `decltype`, overload resolution, and template argument deduction inside the lambda agree with the enclosing scope.

An *init-capture* instead behaves "as if it declares … a variable of the form `auto` *init-capture* `;`" ([expr.prim.lambda.capture] paragraph 6, N3610, N3648), so its type is deduced by `auto`, which strips top-level cv-qualifiers and references.

4.9.1 `mutable const T`

Applying `mutable` to the cv-preserving simple-capture type can instead yield an ill-formed `mutable const T` – consider for example the following (which is easy to reach in generic code or after a refactor):

```
const int x = 5;
auto f = [mutable x]() { x = 0; };  // ??
```

Because an init-capture deduces by `auto`, it is not affected by this problem: `[mutable x = e]` is `mutable auto x = e;`, and `[const x = e]` is `const auto x = e;`. `auto` never produces a top-level `const`, so `mutable` never collides with one.

Entity type of x	Capture	(1) preserve cv-qualifiers	(2) deduce by `auto`
`T`	`[mutable x]`	`mutable T`	`mutable T`
`T`	`[const x]`	`const T`	`const T`
`const T`	`[const x]`	`const T`	`const T`
`const T`	`[mutable x]`	`mutable const T` – ill-formed	`mutable T`

1. Simple-capture: *preserve the entity's cv-qualifiers, then add the requested qualifier*. Adding `mutable` over a `const` entity forms `mutable const T`, which is an error which we simply accept. This is faithful to cv-preservation but surprising and fragile: whether `[mutable x]` compiles depends on a cv-qualifier the author may not control.
2. Init-capture: *deduce by* `auto` *rules, then apply the requested qualifier*, exactly as an init-capture does. `mutable` drops any top-level `const`; `const` adds one. This is uniform across both qualifiers and both capture forms.

Here we adopt (2): A programmer who writes `mutable` or `const` on a capture is requesting a customization, so faithfully preserving the source cv-qualifiers – the simple-capture default – is neither expected nor useful; deducing as an init-capture does makes a qualified simple-capture and the corresponding init-capture produce the same member, and removes the `mutable const T` hazard.

For an entity of type `T`:

– `mutable` produces a `mutable` member of type `std::remove_const_t<std::remove_reference_t<T>>`, and
– `const` produces a member of type `const std::remove_reference_t<T>`.

An unqualified by-copy capture is unchanged. The qualifier is a genuine member qualifier, not an "as-if" treatment confined to the call operator: the closure is exactly the struct a programmer would hand-write, so `decltype`, overload resolution, and reflection (P2996) all observe the real member type.

4.10 Implementation Experience

Ville Voutilainen implemented this proposal, including its extensions, in GCC with regression tests, and reported:

> 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.

That the change reduces to adjusting capture-member types and storage-class-specifiers is itself evidence for Section 6: the closure is already a class, and the proposal only sets qualifiers on its members. The implementation is available on GitHub and can be tried on Compiler Explorer.

5 Concerns

5.1 Consequences of Const Members

Because the member is genuinely `const`, it carries the ordinary consequences of a `const` data member – nothing lambda-specific. A `const` member is copied rather than moved by the defaulted move constructor – you cannot move from a `const` object – so the closure's move constructor is `noexcept` only when the member's *copy* constructor is. Containers notice – `std::vector` reallocation uses `move_if_noexcept`, so a member with a throwing copy (e.g. `std::string`) is copied on every growth:

```
auto concatWith(const std::string x) {  // note the const
  return [x] (std::string y) {          // deduce NSDM as `const std::string`
    return x + y;
  };
}

int main() {
  using Concat = decltype(concatWith(""));
  std::vector<Concat> concats;
  concats.emplace_back(concatWith("A"));
  concats.emplace_back(concatWith("B")); // vector realloc: all elements are copied.
  concats.emplace_back(concatWith("C")); // if Concat had a nothrow move ctor, this
  concats.emplace_back(concatWith("D")); // would have been a move instead.
}
```

This regression is not introduced by the proposal: `[x]` of a `const std::string` already produces a `const` member with exactly this behavior today (CWG756). A const capture only makes the request explicit. Two further consequences follow from the same class rule:

– Assignment. A `const` member also deletes copy and move assignment. This is inert while lambdas delete assignment regardless, but P3963 (approved by EWG) restores it for ordinary captures; a const capture then correctly opts back out – exactly as a `const` member of a hand-written callable would.
– Move-only captures. For a move-only captured type the `const` member cannot be copied (the type is move-only) and cannot be moved (a `const` object can only be copied from, never moved) – so the closure is non-movable: a diagnosed error at the use site, not a silent pessimization.

For instance, const-capturing a `unique_ptr` yields a closure that cannot be stored in a `move_only_function`:

```
move_only_function<int()> f =
  [const p = std::make_unique<int>(42)] { return *p; };
  // error: the closure has a const std::unique_ptr<int> member, which can be
  //        neither moved (it is const) nor copied (unique_ptr is move-only),
  //        so the closure is non-movable and move_only_function cannot store it.
```

These are properties of a faithful model (Section 6), not defects to engineer around; the alternative – a non-`const` member behind a `const` spelling – would trade a teachable rule for a hidden one.

5.2 Teaching Const Capture

`const` capture behaves as a `const` member because it is one; the rules for using it well are the rules for any `const` member.

– Reach for `const` capture to make genuinely-immutable owned state immutable, not as a reflexive annotation. Its cost is the cost of a `const` member, no more and no less.
– A closure headed for a reallocating container, or one that must be assignable, pays for a `const` capture of an expensive-to-copy type on every move; if that cost matters, do not `const`-capture that member.
– Never `const`-capture a move-only type you must move out of – the closure becomes non-movable.
– To read an object without owning a copy, capture by `const` reference rather than by `const` value; there is then no member to move, subject to the usual reference-lifetime caveat.
– "`const` within the body but a movable member" is a *different* feature – logical versus physical `const` – and is not what `[const x]` means; spelling it `const` would give the keyword two meanings.

5.3 East v. West Const

In both East- and West-`const` styles the `const` appears before the identifier; this proposal does not change that.

5.4 Pointer to Const v. Const Pointer

Current lambda behavior mandates bitwise `const` – a `const` pointer, not a pointer to `const`. This proposal continues that rule and does not modify it.

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

5.5 Static Call Operator

A lambda whose call operator is `static` (P1169) has no object parameter and may not have a *lambda-capture*, so a `const` or `mutable` capture cannot co-occur with a `static` call operator; the combination is ill-formed.

6 Lambdas Are Syntactic Sugar for Function Objects

C++ has converged towards the reality that lambdas are just sugar for a hand-written function object; this proposal only lets the sugar express qualifications the desugared class already supports.

1. The standard specifies the closure as a class.
   – [expr.prim.lambda.closure] paragraph 1 – "a unique, unnamed non-union class type"
   – [expr.prim.lambda.capture] paragraph 10, CWG756 – by-copy captures are non-static data members that retain the entity's cv-qualifiers

- [expr.prim.lambda.closure] paragraph 7 – the call operator is a member; special members are "implicitly defined as usual"
- [expr.prim.lambda.capture] paragraph 6, N3610, N3648 – an init-capture is defined as an `auto` variable declaration
- N3649 – a generic lambda's call operator is a member template

2. Compilers represent it as a class.
   - Clang: the closure is a `CXXRecordDecl`, each capture a `FieldDecl`, the call operator a `CXXMethodDecl` (CXXRecordDecl, ASTLambda.h)
   - GCC: Ville Voutilainen's proof-of-concept for this proposal was "adjusting the types of the capture members … and the storage-class-specifier for mutable" – "a single afternoon" (branch)

3. Reflection exposes the captures as ordinary members.
   - P2996 – `nonstatic_data_members_of` enumerates a closure's captures; `type_of` and `is_mutable_member` report each member's type and `mutable`-ness
   - a capture spelled `const` whose member were not `const` would make reflection report a falsehood, so the member must be real

4. Each revision has closed a gap with ordinary classes, never opened one.
   - CWG756 – cv-faithful capture members (C++11)
   - N3649, N3610, N3648 – generic lambdas and init-captures (C++14)
   - P0428, P0780 – explicit template parameters for generic lambdas, and pack-expansion init-captures (C++20)
   - P0624 – captureless lambdas default-constructible and assignable (C++20)
   - P2996 – reflection over closure members (C++26)
   - P3963 – copy and move assignment for captured lambdas (EWG-approved, pending CWG)

5. It is the orthogonal design.
   - `const`, `mutable`, and reference qualifiers mean on a capture exactly what they mean on a member – not a microcosm with bespoke rules; the "simpler language hiding in C++" is the function object the lambda already lowers to

You can run this code today:

```cpp
#include <experimental/meta>
#include <iostream>

template <typename L, std::size_t I>
void print_capture() {
    constexpr auto ctx = std::meta::access_context::unchecked();
    constexpr auto m   = std::meta::nonstatic_data_members_of(^^L, ctx)[I];
    constexpr auto t   = std::meta::type_of(m);
    if constexpr (std::meta::is_mutable_member(m))
        std::cout << "mutable ";
    std::cout << std::meta::display_string_of(t) << '\n';
}

template <typename L>
void print_captures() {
    constexpr auto ctx  = std::meta::access_context::unchecked();
    constexpr auto size = std::meta::nonstatic_data_members_of(^^L, ctx).size();
    std::cout << "capture count: " << size << '\n';
    [&]<std::size_t... I>(std::index_sequence<I...>) {
        (print_capture<L, I>(), ...);
    }(std::make_index_sequence<size>{});
}

int main() {
    auto lam = [x = 42, y = 3.14, z = true]() { return x; };
    print_captures<decltype(lam)>();
    return 0;
}
```

(Compiler Explorer: x86-64 clang, `-freflection-latest -std=c++26`)

We are not proposing a new model of lambdas; we are completing one the language has converged on since C++11 – letting the programmer spell the `const` and `mutable` members the desugared function object could always have held.

6.1 Remaining Gaps

The standard deliberately withholds three structural guarantees an ordinary class would give ([expr.prim.lambda.closure]):

1. the declaration order of capture members is unspecified,
2. the implementation may vary their size, alignment, trivial-copyability, and standard-layout-ness, and
3. the closure type is not an aggregate.

None of this is in tension with `const` capture meaning a `const` member: the cv-qualification of a member is a semantic property, independent of where the member sits or whether the type is an aggregate.

Beyond those structural freedoms, a closure is not interchangeable with a hand-written function object in three further ways.

1. Special members, with captures. A closure with captures has no default constructor and a deleted copy assignment operator ([expr.prim.lambda.closure]); a hand-written struct would have both defaulted. These are gaps the language is closing on the same trajectory as everything else – captureless lambdas gained them in C++20 (P0624), and P3963 would restore assignment for captured lambdas – not properties this paper changes.
2. Anonymity. A closure type is unique and unnamable: it cannot be forward-declared, and a programmer cannot add data members, member functions, base classes, or constructors to it. This is inherent to a lambda being an *expression* rather than a class definition; the sugar generates a fixed shape.
3. The conversion a struct lacks. A captureless closure converts to a function pointer ([expr.prim.lambda.closure]) – a divergence in the opposite direction, an affordance no plain struct has.

The thesis is that the closure *is* a class with a function object's member semantics, and that `const` and `mutable` on a capture should mean what they mean on a member – not that a lambda is a way to write an arbitrary class. These residual differences are exactly what make a lambda worth having.

Thanks

Thanks to Patrick McMichael for suggesting the idea; to Nevin Liber and Matt Calabrese for important corrections; to Nevin Liber, Davis Herring, Barry Revzin, and Victoria Tsai for examples and suggestions; to Ville Voutilainen for the exploratory implementation; and to Daveed Vandevoorde for feedback on the wording.

Proposed Wording

Changes are relative to N5008, using the insert and ~~strike~~ convention. This wording is partial: it states the settled additions; the remainder is flagged for a CWG pass.

Editorial — still to be drafted: the interaction of the `const`, `const&`, *and* `mutable`capture-default
*s with explicit*simple-capture
s ([expr.prim.lambda.capture] paragraph 2); the by-copy `const` *form threaded through [expr.prim.id.unqual] paragraph 4 and the nested re-capture rule ([expr.prim.lambda.capture] paragraph 14); and the semantic (logical-*`const`*) specification of* `[const&]`*. A feature-test macro is noted below.*

[expr.prim.id.unqual]

Change [expr.prim.id.unqual] 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 [expr.prim.lambda.general]

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

Change [expr.prim.lambda.general] 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] paragraph 7

> … 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. …
>
> [*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 [expr.prim.lambda.capture]

> *capture-default:*
> *capture-default-qualifier*$_{opt}$ =
> `const`$_{opt}$ &
>
> *capture-default-qualifier:*
> `const`
> `mutable`
>
> *simple-capture:*
> `mutable`$_{opt}$ *identifier* ...$_{opt}$
> `const` *identifier* ...$_{opt}$
> `const`$_{opt}$ & *identifier* ...$_{opt}$
> *this*
> **this*
>
> *init-capture:*
> `mutable`$_{opt}$ ...$_{opt}$ *identifier initializer*
> `const` ...$_{opt}$ *identifier initializer*
> `const`$_{opt}$ & ...$_{opt}$ *identifier initializer*

Change [expr.prim.lambda.capture] 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 [expr.prim.lambda.capture] 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 variable 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 [expr.prim.lambda.capture] 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.
>
> An entity captured by copy is said to be *captured by const copy* if the *capture* begins with the `const` 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,
> – `const std::remove_reference_t<T>` if the entity is captured by const copy, 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 [expr.prim.lambda.capture] 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 [expr.prim.lambda.capture] 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 [expr.prim.lambda.capture] 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`.

Feature-Test Macro

On adoption, bump `__cpp_lambdas` in [cpp.predefined] to the value corresponding to this paper.

Bibliography

[N2529] *Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 3)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2529.pdf

[N2550] *Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4)*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf

[N2658] *Constness of Lambda Functions (Revision 1)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2658.pdf

[CWG756] *Dropping cv-qualification on members of closure objects*
https://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#756

[N3610] *Generic lambda-capture initializers, supporting capture-by-move*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3610.html

[N3648] *Wording Changes for Generalized Lambda-capture*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3648.html

[N3649] *Wording for Auto Generic Lambda Proposal (Generic (Polymorphic) Lambda Expressions, Revision 3)*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3649.html

[P0624] *Default constructible and assignable stateless lambdas*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0624r2.pdf

[P0428] *Familiar template syntax for generic lambdas*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0428r2.pdf

[P0780] *Allow pack expansion in lambda init-capture*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0780r2.html

[P0288] *move_only_function*
https://wg21.link/p0288

[P0792] *function_ref: a type-erased callable reference*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p0792r14.html

[P2548] *copyable_function*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2548r6.pdf

[N5008] *Programming Languages — C++, Working Draft*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/n5008.pdf

[P2996] *Reflection for C++26*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2996r12.html

[P3963] *Assignable lambdas with capture*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p3963r0.html