

Const/Mutable Extended Lambda Captures

Document [P2034R9](#)
Date 2026-08-19
Audience CWG
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

Revision History	3
Polls	6
1 Background	8
2 Initial Motivation: Const-correctness	8
3 Subsequent Motivation: Symmetry and Simplicity	10
4 Design	11
4.1 Summary	11
4.2 Const Lambdas	11
4.3 Mutable Capture By-copy	11
4.4 Const Capture By-copy	12
4.5 Mutable Capture By-reference	13
4.6 Const Capture By-reference	13
4.7 Capture Defaults	14
4.8 Captures of this	16
4.9 Deducing the NSDM Type	16
4.10 Recaptures	18
4.11 Reference Captures of Temporaries	19
4.12 Interaction with consteval and constexpr Lambdas	20
4.13 Implementation Experience	21
5 Concerns	21
5.1 Consequences of Const Members	21
5.2 Teaching Const Capture	22
5.3 East v. West Const	23
5.4 Pointer to Const v. Const Pointer	23
5.5 Static Call Operator	23
6 Lambdas Are Syntactic Sugar for Function Objects	23
6.1 Remaining Gaps	25
7 Wording Design	25
7.1 The by-copy member carries most of the feature	26
7.2 The const specifier and the specifier constraints	26
7.3 const& has no member to qualify	26
7.4 Capture-defaults reduce to one redundancy rule	27

7.5 What needed no wording	27
Proposed Wording	28
[expr.prim.id.unqual]	28
[expr.prim.lambda.general]	28
[expr.prim.lambda.closure]	29
[expr.prim.lambda.capture]	29
Feature-Test Macro	32
Thanks	32
Bibliography	33

Revision History

Changes from R8

- Added Section 4.11, “Reference Captures of Temporaries”: what happens when a `const&` capture binds to a temporary, how copies of the closure behave, and the loss of the temporary when the closure is returned. Notes the reading of [\[expr.prim.lambda.capture\] paragraph 6](#) and [\[class.temporary\] paragraph 6](#) that this raises, for CWG’s view.
- Gave a second reason `[mutable&]` is unnecessary: the call operator’s `const` never reaches through a reference, so a plain `&` capture is already modifiable (Section 4.5).
- Stated that `[mutable& x]` and `[mutable&]` are ill-formed because the grammar admits no such production, with [\[dcl.stc\] paragraph 8](#) as the reason for omitting it rather than the rule that rejects it.
- Cited specific paragraphs throughout, and corrected citations naming the wrong subclause: the claims that a reference capture need not create a member, and that this is captured by reference, are in [\[expr.prim.lambda.capture\]](#), not [\[expr.prim.lambda.closure\]](#).
- Updated the [\[expr.const\]](#) citations for its split into [\[expr.const.core\]](#) and [\[expr.const.init\]](#).
- Rebased the proposed wording from N5008 onto N5054. Paragraph numbering in [\[expr.prim.lambda.closure\]](#) had shifted, and P3847 moved the declaration-order sentence out of [\[expr.prim.lambda.capture\] paragraph 10](#) and narrowed it: members introduced for explicit captures now follow the order of those captures. Narrowed the corresponding claim in Section 6.1 and added P3847 to the list of gaps the language has closed.

Changes from R7: [EWG Discussion](#)

- Changed audience to CWG: EWG forwarded the paper to CWG for inclusion in C++29 (2026-06 Brno).
- Recorded the 2026-06 Brno polls.
- Retained the capture-default syntax `[mutable =]`, `[const&]`, and `[const =]`: EWG reached no consensus either to additionally allow or to substitute the `[=mutable]`, `[&const]`, and `[=const]` spellings.
- Completed the proposed wording for CWG review, drafting the parts previously deferred:
 - implicit capture under the qualified capture-defaults `[mutable =]` and `[const =]`, and which explicit captures may accompany a qualified default ([\[expr.prim.lambda.capture\] paragraph 2](#));
 - the logical-`const` specification of `[const&]`, in [\[expr.prim.id.unqual\]](#) and the nested re-capture rule ([\[expr.prim.lambda.capture\] paragraph 14](#)); and
 - the non-implicit capture of `*this` under a qualified capture-default.
- Unified the qualified-by-copy member type with auto deduction, stripping both `const` and `volatile`.
- Added a “Wording Design” section explaining the structure of the normative changes.
- Expanded the design discussion: added “Recaptures” and “Redundant Default Captures”, and rewrote “Const Capture By-reference”.
- Dropped the proposed restriction making a mutable capture ill-formed on a `constexpr` or `constexpr` lambda: it was unnecessary, since [\[expr.const\]](#) already governs when such a member may be read. Broke “Interaction with `constexpr` and `constexpr` Lambdas” out into its own section and rewrote it accordingly.

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

2026-06 Brno, R7

D2034R7 should be modified to also allow [=mutable], [&const], and [=const] in addition to [mutable=], [const&], and [const=], with the same semantics respectively: *Not consensus*

SF	F	N	A	SA
1	2	6	8	2

D2034R7 should be modified to replace [mutable=], [const&], and [const=] with [=mutable], [&const], and [=const] respectively: *Not consensus*

SF	F	N	A	SA
2	0	9	7	1

Forward D2034R7 (as on the wiki) to CWG for inclusion in C++29: *Consensus*

SF	F	N	A	SA
4	13	2	1	2

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 ([\[func.wrap.move.ctor\]](#)).

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
--------	-------

<pre> struct A { 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; } }; // 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>

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

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
<code>[mutable x]() const {}</code>	simple capture of <code>x</code> by copy; the NSDM is <code>mutable</code>

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

	<code>() const</code>	<code>() mutable</code>
<code>[x]</code>	<pre>struct X { int x; void operator()() const; };</pre>	<pre>struct X { int x; void operator()(); };</pre>
<code>[mutable x]</code>	<pre>struct X { mutable int x; void operator()() const; };</pre>	<pre>struct X { mutable int x; void operator()(); };</pre>

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
<code>[const x]() mutable {}</code>	simple capture of x by copy; the NSDM is const
<code>[const x...]() mutable {}</code>	simple capture of pack x; each NSDM is const
<code>[const x = init]() mutable {}</code>	init-capture initialized from init; the NSDM is const
<code>[const ...xs = init]() mutable {}</code>	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.

	<code>() const</code>	<code>() mutable</code>
<code>[x]</code>	<pre>struct X { int x; void operator()() const; };</pre>	<pre>struct X { int x; void operator()(); };</pre>

<code>[const x]</code>	<pre>struct X { const int x; void operator>() const; };</pre>	<pre>struct X { const int x; void operator>(); };</pre>
------------------------	--	--

4.5 Mutable Capture By-reference

We explicitly disallow capture of the form `[mutable& x]`: the grammar admits no such *simple-capture*, since mutable references are not permitted by the language ([\[dcl.stc\] paragraph 8](#)). It would also have nothing to do: the call operator's `const` never reaches through a reference, so `[&x]` already modifies `x` on a `const` lambda.

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 copy is made `const` by the call operator; capture by reference is not. Two things stand in the way:

First, the `const` on the call operator is *shallow*: it stops you from reassigning a captured pointer or reference, but says nothing about what that pointer or reference binds to. Capturing a pointer declares a member of that pointer type ([\[expr.prim.lambda.capture\] paragraph 10](#)), so `const` qualifies the pointer, not its pointee.

```
int i = 5;
int* p = &i;
auto l = [p]() const { *p = 0; }; // ok: const does not reach the pointee
auto x = [p]() const { p = nullptr; }; // error: the captured pointer is const
```

Second, a reference capture need not produce a member at all: the standard leaves unspecified whether one is declared ([\[expr.prim.lambda.capture\] paragraph 12](#)), and only by-copy captures are rewritten into member accesses ([\[expr.prim.lambda.capture\] paragraph 11](#)) – so there is nothing for `const` to attach to.

Capturing by `const` reference is nonetheless useful – read-only access to an object too large to copy – but today it takes `std::cref` or `std::as_const`, neither as concise nor as discoverable as `const&`.

We therefore depart from analogy with struct members briefly, and define the meaning `[const& x]` directly: within the body, `x` is a `const` lvalue reference, and any nested lambda that re-captures it observes that `const` (see Section 4.10). The usual reference-lifetime caveats apply (Section 4.11). The call operator's `const` is shallow on references, so it never reaches the referent – which is also why `[mutable&]` would add nothing (Section 4.5).

4.6.1 Syntax

Capture Syntax	Description
<code>[const& x]() mutable {}</code>	simple capture of <code>x</code> by <code>const</code> reference
<code>[const& x...]() mutable {}</code>	simple capture of pack <code>x</code> , each by <code>const</code> reference

<code>[const& x = init]() mutable {}</code>	init-capture binding a const reference to <code>init</code>
<code>[const& ...xs = init]() mutable {}</code>	init-capture pack (P0780), each binding a const reference

4.6.2 Applicability

Unlike the by-copy forms, a `const&` capture is never redundant. On a `const` lambda `[x]` already yields a `const` member, so `[const x]` adds nothing; but `[&x]` does not yield a `const` view, because the call operator's `const` does not reach the referent. `[const& x]` is the only way to ask for one, and it asks for the same thing on either kind of lambda. The cells below give the meaning of `x` in the body rather than a desugared `struct`, since a reference capture need not declare a member to show.

	<code>() const</code>	<code>() mutable</code>
<code>[&x]</code>	<code>x</code> is a modifiable lvalue	<code>x</code> is a modifiable lvalue
<code>[const& x]</code>	<code>x</code> is a const lvalue	<code>x</code> is a const lvalue

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 `=`:

	<code>=</code> (by copy)	<code>&</code> (by reference)
(unqualified)	<code>[=]</code>	<code>[&]</code>
<code>const</code>	<code>[const =]</code>	<code>[const&]</code>
<code>mutable</code>	<code>[mutable =]</code>	ill-formed

`[const =]` captures every implicitly-captured entity by `const` copy, `[mutable =]` by `mutable` copy, and `[const&]` by `const` reference. `[mutable&]` is ill-formed – the grammar admits no such *capture-default* – since `mutable` references do not exist ([\[dcl.stc\] paragraph 8](#)).

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.

4.7.1 Redundant Default Captures

Today, `[=, x]` is ill-formed: under the `=` default `x` is already captured by copy, so naming it again by copy is redundant, and the language rejects it ([\[expr.prim.lambda.capture\] paragraph 2](#)). We keep that principle and extend it to the qualified defaults.

The default governs everything captured *implicitly*; an explicit capture in the same list overrides it for one named entity and carries its own qualifier; and the single thing you cannot write is an explicit capture that does *exactly* what the default already does. Under each default the one redundant – and therefore ill-formed – form is:

Capture-default	Redundant (ill-formed) explicit capture
<code>[=]</code>	<code>[=, x]</code>
<code>[mutable =]</code>	<code>[mutable =, mutable x]</code>
<code>[const =]</code>	<code>[const =, const x]</code>
<code>[&]</code>	<code>[&, &x]</code>
<code>[const&]</code>	<code>[const&, const& x]</code>

Pairing each of the five capture-defaults with the five ways to write a single named capture (`x`, `mutable x`, `const x`, `&x`, `const& x`) gives twenty-five combinations: the five above are redundant, and the other twenty compose – an explicit capture overrides the default for its own entity, carrying its own qualifier. A few:

Capture	Effect
<code>[mutable =, const x]</code>	implicit captures mutable; <code>x</code> a const member
<code>[const =, mutable x]</code>	implicit captures const; <code>x</code> mutable
<code>[=, &x]</code>	captured by copy; <code>x</code> by reference
<code>[&, x]</code>	captured by reference; <code>x</code> a copy
<code>[const&, x]</code>	const-reference views; <code>x</code> a copy
...	...

So a default carries the common case while one entity is named as the exception – the case raised in Section 2 (most captures modifiable, one `const`; or the reverse) – without spelling out every capture by hand.

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 ([\[expr.prim.lambda.capture\] paragraph 12](#)). `[*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. The obstacle differs between the two, and neither depends on which qualifier is written:

	unqualified	const or mutable
<code>[this]</code>	captures the enclosing object by reference	ill-formed: no by-copy member for the qualifier to attach to
<code>[*this]</code>	captures the object by copy, as an unnamed member	ill-formed: that member is unnamed, so the const-propagation wording does not reach it

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 ([\[expr.prim.this\] paragraph 4](#)) 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 ([\[expr.prim.lambda.capture\] paragraph 11](#)) – 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. This proposal changes neither, and leaves an unqualified by-copy capture exactly as it is today; it settles only which of the two rules a *qualified* capture should follow.

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](#)), so its type is deduced by `auto`, which strips top-level cv-qualifiers and references ([\[dcl.type.auto.deduct\] paragraph 3](#), [\[temp.deduct.call\] paragraph 2.3](#)).

The two rules, side by side:

Entity type of x	[x] (simple-capture)	[y = x] (init-capture)
T	T	T
const T	const T	T
volatile T	volatile T	T
T&	T	T
const T&	const T	T
T(&)()	T(&)()	T(*)()

The `const T&` row is the rule Section 4.10 depends on: a copy taken of a `const` view is itself `const`.

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.

That leaves two candidate rules for a qualified simple-capture:

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
volatile T	[mutable x]	mutable volatile T	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. `auto` deduction drops any top-level cv-qualifiers, so `mutable` never collides with a `const`; `const` then adds one. This is uniform across both qualifiers and both capture forms.

4.9.2 The Adopted Rule

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_cvref_t<T>`, and
- `const` produces a member of type `const std::remove_cvref_t<T>`.

Stripping all top-level cv-qualifiers – not just `const` – is exactly what auto deduction does, so a qualified simple-capture and the corresponding init-capture deduce the same member type even for a `volatile` entity. The one exception is a reference to a function, which the type rule settles before the qualified branches apply: the simple-capture keeps a reference, while the init-capture decays to a pointer.

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 Recaptures

`[const& x]` gives the body a `const` view of the original object. What nesting raises is the question of what an *inner* lambda sees when it re-captures `x`:

```
auto outer = [const& x] {           // x is a const view of the original
    auto inner = [x] { /* ... */ }; // inner makes its own copy of x
};
```

`inner` copies `x`, and because it copies from a `const` view the copy is itself `const` – the same long-standing rule (CWG756) that makes `[x]` of a `const` variable produce a `const` member. The consequence shows when the inner lambda is `mutable`:

```
auto outer = [const& x] {
    auto inner = [x]() mutable { x = 0; }; // ill-formed: inner's copy is const
};
```

Letting `inner` mutate its own copy would quietly contradict the `const` view `[const& x]` promised one line above. (The original object is untouched either way – but a *modifiable* copy silently appearing from a `const` capture is the kind of surprise `const`-correctness exists to prevent.) The `const` carries through any depth of nesting, including through intervening plain-reference captures: a `[const& x]` several levels out still yields a `const` copy at the bottom.

None of the following are special cases; each follows from two facts – a `const`-reference view is `const`, and a copy of a `const` view is `const`. The first column is the nesting, read outermost-to-innermost; the second is what the innermost `x` is.

Nesting	Innermost x	Why
<code>[const& x]{ x; }</code>	<code>const lvalue</code>	the <code>const</code> -reference view

<code>[const& x]{ [&x]{ x; } }</code>	const lvalue	the view stays const through nesting
<code>[const& x]{ [x]{ x; } }</code>	const copy	a copy of a const view is const
<code>[const& x]{ [x]() mutable { x = 0; } }</code>	ill-formed	the const copy cannot be made mutable
<code>[const& x]{ [&x]{ [x]() mutable {} } }</code>	const copy	const carries through the plain-& middle
<code>[&x]{ [x]() mutable { x = 0; } }</code>	OK	no const& anywhere; an ordinary modifiable copy

`[const& x]` behaves the same on a const or a mutable lambda: the const belongs to the view of the object, not to a member supplied by the call operator. And because there is no by-copy member, there is nothing to move or copy when the closure is moved – subject to the usual reference-lifetime caveat.

4.11 Reference Captures of Temporaries

`[const& x = init]` can bind to a temporary. The language already permits this, but it is newly easy to write: today it takes an initializer that is already const – a function returning const T, say – since `auto&` will not bind to a non-const prvalue, which is why `[&x = f()]` is normally an error. `[const& x = bar()]` works with any ordinary `bar()`. This proposal turns an obscure corner into an idiomatic spelling, so the lifetime question deserves an answer.

The answer follows from the existing rules. An *init-capture* behaves as if it declares a variable of the form `auto init-capture ;`, and for a capture by reference “the variable’s lifetime ends when the closure object’s lifetime ends” ([\[expr.prim.lambda.capture\] paragraph 6](#)); a temporary bound to that reference persists for the lifetime of the reference ([\[class.temporary\] paragraph 6](#)). The temporary therefore lives exactly as long as the closure object whose *init-capture* created it.

```
{
    auto f = [const& v = bar()] { use(v); };
    f();                      // OK: the temporary is alive for as long as f is
}                             // f destroyed, then the temporary
```

Three consequences follow, and they settle the questions this form raises:

- **The temporary belongs to one closure object.** It is tied to the closure whose *init-capture* created it – not to the closure type, and not to any copy.
- **Copying extends nothing.** A copy binds its own reference directly to the same object. Lifetime extension applies where a reference binds to a *temporary* ([\[class.temporary\] paragraph 6](#)); binding a further reference to an already-bound object is not such a case and does not lengthen anything.
- **A copy that outlives the original dangles.** The temporary dies with the original closure, leaving any surviving copy referring to a destroyed object – the ordinary consequence of a reference outliving its referent.

The sharp edge is returning such a closure:

```

auto make() {
    return [const& v = bar()] { use(v); }; // the temporary does not survive the
return
}

auto g = make();
g(); // dangling

```

By guaranteed copy elision the closure object is the caller's, but the temporary was materialized in the callee's frame and is destroyed when `make` returns. This is diagnosable, and is diagnosed: for the spelling reachable today, Clang reports “*returning address of local temporary object*” with the note “*captured by reference via initialization of lambda capture*”. GCC does not accept that spelling at all – it rejects `[&x = g()]` for a `const`-returning `g`, though it accepts the equivalent `auto& x = g();` – so the existing form is barely exercised. The same happens without lambdas, though: a returned aggregate with a reference member bound to a temporary loses that temporary at the return, on both GCC and Clang, so this is how reference lifetime behaves across a return generally, not something particular to closures.

We would welcome CWG's view on how the wording is meant to be read here. Taken literally, it appears to say something else: if the returned closure object's lifetime is the caller's, then so is the *init-capture* variable's ([\[expr.prim.lambda.capture\] paragraph 6](#)), and the temporary bound to it persists for the lifetime of that reference ([\[class.temporary\] paragraph 6](#)) – which would require the temporary to outlive the frame it was materialized in. We may be reading it wrongly. If we are not, it seems better handled as a core issue than as part of this paper. Either way the behavior is reachable in C++26 and is not introduced by this proposal; we raise it because this proposal makes it much easier to encounter.

4.12 Interaction with `constexpr` and `constexpr` Lambdas

A mutable capture raises no new question for a `constexpr` or `constexpr` lambda. Mutating and reading local state during constant evaluation has been allowed since C++14 (N3652), and a mutable data member is part of that – a function object with a mutable member can be evaluated at compile time today:

```

struct Counter {
    mutable int n = 0;
    constexpr int operator()() const { return ++n; }
};

constexpr int f() {
    Counter c;
    return c() + c(); // reads and writes the mutable member at compile time
}
static_assert(f() == 3, ""); // OK since C++14

```

The lambda spelling has worked since lambdas became usable in constant expressions in C++17: an ordinary mutable lambda already holds state it mutates and reads.

```
constexpr int g() {
    auto counter = [n = 0]() mutable { return ++n; };
    return counter() + counter();
}
static_assert(g() == 3);    // OK since C++17
```

Writing `[mutable n]` declares the same kind of member as the `mutable` lambda above and behaves the same way. The one case that does not work – reading a `mutable` member of an object that already existed before the evaluation began – is not specific to lambdas: it fails for a hand-written `mutable` member in exactly the same way. So a `mutable` capture needs no rule of its own, and this paper adds none.

4.13 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. The move constructor initializes each member from the corresponding member of an xvalue referring to its parameter ([\[class.copy.ctor\] paragraph 15](#)), so a `const M` member yields a `const M` xvalue, which cannot bind to `M(M&&)` ([\[class.copy.ctor\] paragraph 9](#)); the copy constructor is selected instead. 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 ([\[class.copy.assign\] paragraph 7](#)). This is inert while lambdas delete assignment regardless ([\[expr.prim.lambda.closure\] paragraph 17](#)), 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. The captured member has the captured entity’s type ([\[expr.prim.lambda.capture\] paragraph 10](#)), so qualifying a captured pointer applies `const` to the pointer, not its pointee. 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* ([\[expr.prim.lambda.general\] paragraph 4](#)), 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)
- P3847 – explicit captures declared in lexical order (C++29)
- 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:

```
#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:

1. the declaration order of capture members is unspecified, except that members introduced for explicit captures follow the order of those captures ([\[expr.prim.lambda.capture\] paragraph 15](#)),
2. the implementation may vary their size, alignment, trivial-copyability, and standard-layout-ness ([\[expr.prim.lambda.closure\] paragraph 4](#)), and
3. the closure type is not an aggregate ([\[expr.prim.lambda.closure\] paragraph 4](#)).

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 ([\[expr.prim.lambda.closure\] paragraph 17](#)); 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 ([\[expr.prim.lambda.closure\] paragraph 11](#)) – 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.

7 Wording Design

The feature is small, and the wording is mostly small with it: in the common case it sets a cv-qualification and a storage-class-specifier on members the closure already declares. This section is a guide to how the normative changes are organized and why they take the shape they do – the design of the *wording*, as distinct from the design of the feature above. It is written for readers following the proposed wording closely.

The changes touch five subclauses:

- [\[expr.prim.id.unqual\]](#) – the type of a name that resolves to a `const`-reference capture;
- [\[expr.prim.lambda.general\]](#) – the `const` *lambda-specifier*;
- [\[expr.prim.lambda.closure\]](#) – a note that the `const` *lambda-specifier* has no effect;

- [\[expr.prim.lambda.capture\]](#) – the bulk: grammar, the qualified members, the capture-default rules, and nested re-capture; and
- [\[cpp.predefined\]](#) – the feature-test macro.

7.1 The by-copy member carries most of the feature

By-copy capture already declares a non-static data member for each capture ([\[expr.prim.lambda.capture\]](#) [paragraph 10](#)); all this proposal adds there is that member’s cv-qualification and whether it is mutable. That one paragraph does most of the work.

The member type is stated as prose rather than as `std::remove_cvref_t<T>`, because [\[expr\]](#) cannot depend on the library. The two qualified cases reduce to a single helper: from the existing cv-faithful captured type U , form V by removing top-level cv-qualifiers; a mutable capture yields V (declared mutable), a const capture yields const-qualified V . V is exactly what auto deduction produces, which is why a qualified simple-capture and the corresponding init-capture agree – except for a reference to a function, which the first sentence of the type rule settles before V is reached.

Two terms – *captured mutably* and *captured by const copy* – are defined there rather than inlined, because two later places refer to them: the member’s mutable storage class and the nested re-capture rule ([\[expr.prim.lambda.capture\]](#) [paragraph 14](#)). Each term is defined over both the explicit capture (the *capture* begins with the keyword) and the implicit capture (the qualified *capture-default*), so one term serves both `[mutable x]` and `[mutable =]`.

Function references are the one by-copy capture whose member is a reference rather than a value. A reference member can be neither const-qualified nor mutable ([\[basic.type.qualifier\]](#) [paragraph 1](#), [\[dcl.stc\]](#) [paragraph 8](#)), so the qualifier is simply inert there. The type rule needs no exception, since it settles the function-reference case before reaching the qualified branches; the mutable storage class excludes it explicitly, and a note records why.

7.2 The const specifier and the specifier constraints

The const *lambda-specifier* introduces no behavior – the call operator is already const unless mutable or static is present – so the closure-type wording gains only a note saying so ([\[expr.prim.lambda.closure\]](#) [paragraph 7](#)), and the specifier constraints in [\[expr.prim.lambda.general\]](#) [paragraph 4](#) gain only the entry forbidding const alongside an explicit object parameter and the mutual exclusion of const, mutable, and static.

7.3 const& has no member to qualify

A reference capture need not create a member at all ([\[expr.prim.lambda.capture\]](#) [paragraph 12](#)), so const has nothing to attach to. The const is therefore a property of the name’s *type*, and the one place that already determines the type of a captured name is [\[expr.prim.id.unqual\]](#) [paragraph 4](#). We add a paragraph there: when a name resolves to an entity captured by const reference anywhere in the enclosing chain of lambdas, its type is const-qualified.

We add a separate paragraph rather than widen the existing by-copy rule. Widening was tried – by changing that rule’s trigger from “captured by copy” to “captured” – and it breaks: the rule names “the member that the capture would create” in the innermost capturing lambda, and a reference capture creates no member,

so the rule contradicts itself whenever that innermost lambda captures by reference. Keeping the by-copy rule untouched and adding a parallel rule for the reference case avoids the contradiction.

Re-capture is where this needs care. The behavior – a copy of a `const` view is itself `const`, to any nesting depth – is described in Section 4.10; the wording achieves it in the re-capture rule ([\[expr.prim.lambda.capture\] paragraph 14](#)), which propagates the `const` by asking whether the entity would have `const`-qualified type within the enclosing lambda, a question it answers through [\[expr.prim.id.unqual\]](#). That phrasing is what carries the `const` through any number of plain-reference intermediaries, but it also makes the two paragraphs refer to each other. The reference is well-founded: each step moves one lambda outward and terminates at the outermost. Still, it is the part of the wording most worth a second look, and CWG may prefer to restate it as a single inductive definition.

7.4 Capture-defaults reduce to one redundancy rule

The existing [\[expr.prim.lambda.capture\] paragraph 2](#) forbids an explicit capture that matches the *capture-default*, phrased two ways: a prohibition for `&`, a whitelist for `=`. With qualifiers, the clean generalization is a single prohibition – an explicit *simple-capture* may not capture an entity in exactly the way the default already would. This reproduces today’s diagnostics, treats all five defaults uniformly, and leaves combinations like `[mutable =, const x]` well-formed – which a whitelist would not.

The qualified defaults also do not implicitly capture `*this` ([\[expr.prim.lambda.capture\] paragraph 7](#)), continuing the direction of the C++20 deprecation; the unqualified `=` and `&` are unchanged.

7.5 What needed no wording

Some cases are handled by omission. The grammar offers no production for a qualified `this` or `*this`, so `[const this]` and the like are ill-formed with no constraint required. The representation of reference captures remains unspecified, so `[const&]` declares no member and needs no member wording. And the `const` *lambda-specifier*, being inert, changes no rule beyond the note noted above.

A mutable capture on a `constexpr` or `constexpr` lambda likewise needs no constraint of its own; the existing rules in [\[expr.const\]](#) already settle it. The lvalue-to-rvalue conversion that reads a member is permitted, among other cases, on “a non-volatile glvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of *E*” ([\[expr.const.core\] paragraph 2.10.3](#)) – an allowance not conditioned on the mutable qualifier – so a mutable member is readable whenever the closure was constructed within the evaluation, the usual case for a closure built and called in one constant expression. A mutable subobject is excluded only from the *other* allowance ([\[expr.const.core\] paragraph 2.10.2](#)), for a glvalue referring to an object “usable in constant expressions”: the definition of *potentially usable in constant expressions* admits only a “non-mutable subobject” ([\[expr.const.init\] paragraph 8.5](#)), so a mutable member of a closure that already existed before the evaluation cannot be read. Both outcomes match a hand-written mutable member, so an unmarked but `constexpr`-suitable lambda is left to fail naturally at use rather than at declaration.

The lifetime of a temporary bound by a `const&` capture also needs no wording of its own (Section 4.11). An *init-capture* is already specified as a variable declaration ([\[expr.prim.lambda.capture\] paragraph 6](#)), so the ordinary rule for a reference bound to a temporary ([\[class.temporary\] paragraph 6](#)) reaches it unchanged; this proposal widens what such a capture can bind to, not how long the bound object lives. One reading of those two rules together, for the case where the closure is returned, is put to CWG in that section.

Proposed Wording

Changes are relative to N5054, using the **insert** and **strike** convention.

[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.

[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]

Add a paragraph after [\[expr.prim.id.unqual\]](#) paragraph 4

Otherwise, 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 const reference ([\[expr.prim.lambda.capture\]](#)) 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*,**

then the type of the expression is the const-qualified type of the entity.

[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 *constexpr*. 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-qualifieropt` =
`constopt` &

capture-default-qualifier:

`const`
`mutable`

simple-capture:

`mutableopt identifier ...opt`
`const identifier ...opt`
`constopt & identifier ...opt`
`this`
`*this`

init-capture:

`mutableopt ...opt identifier initializer`

`const ...opt identifier initializer`
`constopt & ...opt identifier initializer`

Change [\[expr.prim.lambda.capture\] paragraph 2](#)

If a *lambda-capture* includes a *capture-default* that is `&`, no identifier in a *simple-capture* of that *lambda-capture* shall be preceded by `&`. If a *lambda-capture* includes a *capture-default* that is `=`, each *simple-capture* of that *lambda-capture* shall be of the form “`& identifier ...opt`”, “`this`”, or “`* this`”. If a *lambda-capture* includes a *capture-default*, no *simple-capture* of that *lambda-capture* shall be of the form

- “`identifier ...opt`” if the *capture-default* is `=`,
- “`mutable identifier ...opt`” if the *capture-default* is `mutable =`,
- “`const identifier ...opt`” if the *capture-default* is `const =`,
- “`& identifier ...opt`” if the *capture-default* is `&`, or
- “`const & identifier ...opt`” if the *capture-default* is `const &`.

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 7](#)

... the entity is said to be *implicitly captured* by each intervening *lambda-expression* with an associated *capture-default* that does not explicitly capture it, except that `*this` is not implicitly captured by a *lambda-expression* whose *capture-default* is `const =`, `mutable =`, or `const &`. The implicit capture of `*this` is deprecated when the *capture-default* is `=`; see [\[depr.capture.this\]](#). ...

Change [\[expr.prim.lambda.capture\]](#) paragraph 10

An entity is *captured by copy* if

- it is implicitly captured, the *capture-default* is `==`, `mutable` `=`, or `const` `=`, 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` ~~`⊗`~~, `& ...opt identifier initializer` or `const & ...opt identifier initializer`.

An entity captured by copy is *captured mutably* if it is explicitly captured by a *capture* that begins with `mutable`, or it is implicitly captured and the *capture-default* is `mutable` `=`.

An entity captured by copy is *captured by const copy* if it is explicitly captured by a *capture* that begins with `const`, or it is implicitly captured and the *capture-default* is `const` `=`.

For each entity captured by copy, an unnamed non-static data member is declared in the closure type. 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. The type of such a data member is an lvalue reference to the referenced function type if the entity is a reference to a function. Otherwise, letting *U* be the referenced type if the entity is a reference to an object and the type of the entity otherwise, and *V* be *U* with any top-level cv-qualifiers removed, it is

- *V*, if the entity is captured mutably,
- `const`-qualified *V*, if the entity is captured by const copy, or
- *U* otherwise.

If the entity is captured mutably and is not a reference to a function, the data member is declared *mutable*. [Note: For an entity that is a reference to a function, the data member is a reference, which can be neither `const`-qualified nor *mutable* ([\[basic.type.qualifier\]](#), [\[dcl.stc\]](#)); a `const` or *mutable* capture of such an entity therefore has no effect on the data member. — end note] 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 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*. If an *id-expression* naming the entity within the *compound-statement* of *m1* would have const-qualified type ([\[expr.prim.id.unqual\]](#)), then the entity is considered to be const-qualified for the determination of the type of any non-static data member declared for *m2*'s capture ([\[expr.prim.lambda.capture\]](#) paragraph 10).

Feature-Test Macro

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

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 Hana Dušíková and Ville Voutilainen for observing that the `constexpr/constexpr` restriction was unnecessary; to Yihan Wang for raising the lifetime of a `const&` capture bound to a temporary, which became Section 4.11; to Ville Voutilainen for the exploratory implementation; and to Daveed Vandevoorde for feedback on the wording.

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>
- [N3652] *Relaxing constraints on constexpr functions / constexpr member functions and implicit const*
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3652.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>
- [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>

- [P3847] *Lexical order for lambdas*
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p3847r1.html>
- [N5054] *Programming Languages — C++, Working Draft*
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/n5054.pdf>