Mitigation strategies for P2036 ”Changing scope for lambda trailing-return-type”

Document #:	P2579R0
Date:	2022-07-01
Programming Language C++	
Audience:	EWG,CWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

P2036R3 [3] was adopted for C++23 and as a Defect Report, affecting C++11 and greater. After implementing this paper in Clang, we observed the proposed changes make ill-formed previously valid and relied upon code.

While this paper explores multiple options, the provided wording is for option E which was chosen by EWG.

The problem

Previously, in the *lambda-declarator* of a lambda expression, lookup would find the names in the enclosing scope of the lambda, even if the name would refer to an entity that would be captured explicitly or implicitly. This defies user expectations which should be allowed to expect:

- Name lookup would ”find the first thing of that name” (formulation stolen from P1753R0 [1], thanks, Walter!)
- The type of an expression would be consistent throughout a lambda, i.e., be the same in the *lambda-declarator* and in the *compound statement* of a lambda.

And this problem was solved by P2036R3 [3]. Great.

P2036R3 [3], however, faced a problem. The type of an expression involving a capture cannot be known until we know whether the lambda is mutable. And the ‘mutable‘ keyword, is not lexically close from the capture (as it is after the *template-parameter-list*, *requires-clause*, *attribute-specifier-seq* and *parameter-declaration-clause*.

> What do we want to do about a case like this?
>
> ```
> double x;
> [x=1](decltype((x)) y){ return x; }
> ```
>
> There are four options for what this lambda could mean:
>
> - this is a lambda that takes a `double&` (status quo).

> - this is a lambda that takes an `int&` (lookup could be changed to find the init-capture but not do any member access transformation - even though this lambda ends up being not mutable)
> - this is a lambda that takes an `int const&` (would require lookahead, highly undesirable)
> - this is ill-formed

P2036R3 [3] draws the following conclusions:

> While there's a lot of motivation for the trailing-return-type, I have never seen anybody write this and do not know what the motivation for such a thing would be. (1) isn't very reasonable since the init-capture is lexically closer to use, and it's just as surprising to find ::x in the parameter-declaration-clause as it is in the trailing-return-type.
>
> The advantage of (4) is that it guarantees that all uses of x in the lambda-expression after the lambda-introducer mean the same thing — we reject the cases up front where we are not sure what answer to give without doing lookahead. If motivation arises in the future for using captures in these contexts, we can always change the lookup in these contexts to allow such uses — rejecting now doesn't cut off that path.
>
> This paper proposes (4).

I think P2036R3 [3] is fundamentally correct that 'decltype((x))' should behave the same anywhere in the lambda, and the motivation that applies to the trailing return type applies equally to the parameter list. Making it ill-formed is a more desirable outcome than giving it different meanings in different places. However, making it ill-formed proved to have a large blast radius.

P2036R3 [3] breaks existing code

We first found an issue in LLVM, which we patched.

```
if (Seg->end <= *Idx) {
    Seg = std::upper_bound( ++Seg, EndSeg, *Idx,
    [=](std::remove_reference_t<decltype(*Idx)> V, const std::remove_reference_t<decltype(*Seg)> &S) {
        return V < S.end;
    });
    if (Seg == EndSeg)
    break;
}
```

But we then found another issue in libstdc++:

```
void _M_invalidate_locals() {
    auto __local_end = _M_cont()._M_base().cend(0);
    this->_M_invalidate_local_if(
    [__local_end](__decltype(__local_end) __it)
```

```
    { return __it != __local_end; });
}
```

This was fixed by implementing CWG2569 [**?**] however, soon after, the following 2 code reductions were reported as broken.

```
template <typename It, typename MapFn>
auto MapJoin(It first, It last, MapFn map_fn) {
    return std::accumulate(
    first, last, map_fn(*first),
    // a new diagnostic: error: captured variable 'first' cannot appear here
    [=](typename std::result_of<MapFn(decltype(*first))>::type result) {  });
}
```

and

```
void foo() {
    int x = [x](int y[sizeof x]){return sizeof x;}(0);
}
```

As all of these breakages were reported in a non-shipping compiler in the days following the deployment of the change, leading us to believe that they would break a lot more code if it were more widely deployed. The patch was then reverted, so we were not able to gather more data, but we think this proposal would be too disruptive in its current form to be implemented in shipping compilers.

What about these snippets

While I have no evidence of that, it seems that one of the reasons for the amount of breakage we are seeing is that C++11 did not support generic lambdas. And so, code that might use `auto` in more recent versions of C++ would have had to name capture in the parameter list in previous versions and C++. We could argue that introducing type aliases might be more elegant. Still, these code snippets are not, by any stretch of the imagination, outlandish enough that they should no longer work.

Mitigation strategies

A: CWG2569's proposed resolution

After noticing an issue with the compilation of libstdc++ (Bug report), Barry created CWG2569.

This proposes to allow `decltype(capture_id)`, `noexcept(capture_id)`, `sizeof(capture_id)` as we know these expressions will not be affected by the lambda's mutability.

However, this fix proves problematic:

- It does not address the breadth of expressions referring to captures seen in the wild and is, as such insufficient.

- It is rather arbitrary and hard to make sense of. Why is `sizeof(a)` fine by example but `sizeof(+a)` isn't? How do we teach that
- Even with these seemingly-arbitrary looking restrictions, this proposed resolution is a non-trivial implement for at least some implementation.

Implementation experience tells us that any attempt to find a subset of expressions we could allow in a lambda's parameter declaration clause such that it would not affect existing code is a futile pursuit.

B: Before the mutable keyword, lookup find the variable in the enclosing scope

This was the status quo in C++20 and earlier. However, the motivation for P2036R3 [3] still remains. Having `decltype(expr)` give a different result before and after the mutable keyword is not better than giving different results before and after the start of the compound statement of the lambda expression. If we are able to name a capture (explicitly or not), it should give a consistent result. The only possible justification for inconsistency there is implementation difficulty and backward compatibility, which are fine arguments, but they would be incoherent and surprising for users.

C: Making lambdas referring to captures in its parameter declaration clause ill-formed if it is later found to be mutable

A compiler could assume that the lambda call-operator is const, such that captures referred to would be assumed to be const. If later the lambda is found to be mutable, the program could be made ill-formed on the basis that the assumed type of a capture is not consistent in the entire scope of the lambda expression. This may be confusing and does not entirely eliminate the risk of breakage. There exist some `mutable` lambdas that capture by copy and then refer to captured variables in their parameter declaration clause, but certainly much less. A very crude search found a couple results.

This solution does have some benefits:

- It does not require a lookahead
- It would allow the use of capture anywhere after the capture list - including in the template parameter list and require clause.
- It has an evolution path to support mutable captures with P2034R0 [2]: if the mutable keyword is placed in the capture-list, this whole problem goes away.

It is not perfect. First, it reduces but does not eliminate the breakage entirely. And second, it introduces an oddity that can be taught and justified, but it is still an oddity and a novelty that lookup would behave differently depending on whether a mutable keyword is found later.

D: Looking ahead for the existence of the mutable keyword

If the compiler could determine before parsing the lambda whether it will be mutable or not, it could give a consistent answer without arbitrary restriction. P2036R3 [3] flat out rejected

this option with the assumption of implementation burden.

There are four places where a captured entity can appear before the mutable keyword (all of these parts are optional):

```
[=]</*#1*/> requires /*#2*/ (/*4*/) mutable
```

Some (Or at the very least clang) cannot do arbitrary parses and revert it.

The template parameter list (#1) and the following `requires` clause (#2) would need to be fully analyzed to find where they end.

For example, consider:

```
[]<auto N = 1 < 0>() requires std::same_as<decltype(N>0), int>{};
```

P2036R3 [3] is correct to say that this would put a lot of burden on implementations.

However, the `attribute-specifier-seq` and the `parameter-declaration-clause` need only to be lexed (and the parentheses balanced) to be skipped, and lexing ahead is a more tractable problem, that implementations are more likely to be able to implement with a reasonable effort.

The pros of this approach are:

- It does not break existing code referring to a capture in a parameter declaration
- It offers a behavior for the parameter-clause that is consistent with the trailing return type and, hopefully, the user's expectations

It's not a perfect approach: captures cannot be referred to in the template parameter list or first requires clause. The risk of breaking existing code is not null but very limited by virtue of lambda template parameter lists being a C++20 feature, and C++20 not being widely deployed yet - but it is an issue that would get worse with time.

And we still have this problem of that limitation being seemingly arbitrary from the users' perspective.

We should also note that looking ahead is not free in terms of compile-time, but the impact should be minimal.

Still, of all the options presented, D seems to offer the best mitigation strategy while being coherent with P2036R3 [3], and avoiding adding too many new gotchas and corner cases.

We should note that clang already had to implement a form of lookahead to support GNU attributes, which can appear between the lambda parameter declaration clause and the `mutable` keyword, for example:

```
int y;
[=]() __attribute__((diagnose_if(!is_same<decltype((y)), const int &>, "wrong type", "warning"))){}();
```

E: identifiers refered to captured variables but do not take the mutable keyword into account

This option was choosen by EWG, wording is provided below.

Barry observed on the reflector thar, for regular class member functions, refering to a member of that class in the parameter list doesn't take a const qualiffier into account

```
struct F {
    float x;
    void mem1(decltype((x)) p3);       // p3 is a float&
    void mem2(decltype((x)) p4) const; // p4 is a float&
};
```

We could argue that doing the same thing for lambdas would make sense as it would be consistent with regular classes:

```
int x;
[x=42.0]<decltype(x) a> // float
(decltype((x)) b)       // float&
-> decltype((x))        // const float&
```

There are multiple advantages to this approach:

- We can find a capture variable anywhere, including in the template parameter list, so we avoid exceptions and inconsistencies
- Consistent with member functions
- No lookahead

It has however one drawback: `decltype((foo))` can change meaning after the end of the function parameter list.

We also have 2 options here:

- Before the end of the parameter list access to a captured variables are considered mutable, which is consistent with trying to access the member variable outside of a const call operator, and consistent with the behavior of regular classes with member function.
- Before the end of the parameter list access to a captured variables are considered const because const is the default for lambda call operators. I would argue that this is less consistent, and add more special cases.

I find myself whether the behavior of member function is the one we would want if we could change it, but they would suffer the same problem: it's not generally implementable for their template parameter list.

In all, I still slightly prefer option D, but option E looks better than all other options.

Should P2036R3 still be a DR?

From previous wiki discussions, it is not clear how much the different scenarios for breakage and semantic changes have been discussed. I think there are three interesting things to discuss here, in addition to the parameter declaration clause addressed by the rest of this paper.

Template parameters

```
int x;
[x]<auto N = x> {}
```

This is made ill-formed by P2036 and the proposed mitigation. Given that template parameter lists are a c++20 features, and C++20 is not widely deployed yet, not applying that behavior change retroactively would create some churn that benefits no one.

Init capture visibility

```
auto f() {
    float x;
    return [x = 0] -> decltype(x){ return 1.2;}();
}
```

In C++20 `f` would produce a float; it now produces an int. However, this sort of shadowing is ill-advised and probably rare enough that it isn't a large problem in practice. I have no data to back that up.

In general, this paper is a semantic change, and code that relies on the semantics changed by the paper will be affected whether by updating the toolchain or by changing standard mode, and there aren't a lot of opportunities to diagnose that change.

Given the very strong consensus established in Prague and subsequently in 2021 by EWG that this was a defect, I do not see a big motivation to revisit that discussion if a sufficient mitigation strategy - such as option D - is adopted for C++23.

However, in the absence of such mitigation, this paper would not be implemented as a DR, and might not be implemented at all, given the potential damage it causes.

Lookup for non-standard C++ attributes

Lookup for implementation-specific attributes needs to be specified to support existing use cases:

```
int main() {
    int asp = 5;
    [=] [[gnu::regparm(asp)]] () {};
}
```

Implementation experience

Both **A: CWG2569's proposed resolution** and option **D: Looking ahead for the existence of the mutable keyword** have been implemented in clang. Option A was deployed and still broke a lot of production code. Option D was able to successfully compile the synthetic code examples submitted by users after deployment of the original paper implementation and the CWG2569 proposed resolution implementation.

Wording for Option E

[Editor's note: Add the following section immediately after [basic.scope.param].]

§ Lambda scope [basic.scope.lambda]

A *lambda-expression* E introduces a *lambda scope* that starts immediately after the *lambda-introducer* of E and extends to the end of the *compound-statement* of E.

§ Lambda expressions [expr.prim.lambda]

§ Captures [expr.prim.lambda.capture]

[Editor's note: Paragraph 5. The changes to this paragraph have not been explicitely discussed by EWG, and may be better treated as a core issue or an NB comment.]

If an *identifier* in a ~~*simple-capture*~~ appears as the *declarator-id* of a parameter of the *lambda-declarator*'s *parameter-declaration-clause* or as the name of a template parameter of the *lambda-expression*'s *template-parameter-list* the program is ill-formed.

[*Example:*

```
void f() {
    int x = 0;
    auto g = [x](int x) { return 0; };    // error: parameter and ~~simple-~~capture have the same name
    auto h = [y = 0]<typename y>(y) { return 0; }; // error: template parameter and capture have the same name
}
```

— *end example*]

[Editor's note: Paragraph 6]

An *init-capture* inhabits the ~~function parameter scope of the *lambda-expression*'s *parameter-declaration-clause*~~ 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* ;", 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.

🔲 Unqualified names [expr.prim.id.unqual]

[...]

The result is the entity denoted by the *unqualified-id*. If the *unqualified-id* appears in a *lambda-expression* at program point P and the entity is a local entity or a variable declared by an *init-capture*, then let S be the *compound-statement* of the innermost enclosing *lambda-expression* of P. If naming the entity from outside of an unevaluated operand within S would refer to an entity captured by copy in some intervening *lambda-expression*, then let E be the innermost such *lambda-expression* ~~, and:~~ .

- If there is such a *lambda-expression* and if P is in E's function parameter scope but not its *parameter-declaration-clause*, 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:* If E is not declared `mutable`, the type of such an identifier will typically be `const` qualified. — *end note*]
- Otherwise (if there is no such *lambda-expression* or if P either precedes E's function parameter scope or is in E's *parameter-declaration-clause*), the ~~program is ill-formed.~~

~~Otherwise, the~~ type of the expression is the type of the result. [*Note:* If the entity is a template parameter object for a template parameter of type `T`, the type of the expression is `const T`. — *end note*] [*Note:* The type will be adjusted as described in **??** if it is cv-qualified or is a reference type. — *end note*] The expression is an lvalue if the entity is a function, variable, structured binding, data member, or template parameter object and a prvalue otherwise; it is a bit-field if the identifier designates a bit-field. [*Example:*

```
    void f() {
        float x, &r = x;
        [=]() -> decltype((x)) {      // lambda returns float const& because this lambda is not
mutable and
            // x is an lvalue
            decltype(x) y1;             // y1 has type float
            decltype((x)) y2 = y1;      // y2 has type float const&
            decltype(r) r1 = y1;        // r1 has type float&
            decltype((r)) r2 = y2;      // r2 has type float const&
            return y2;
        };

        ~~[=]<decltype(x) P>; // error: x refers to local entity but precedes the lambda's function
parameter scope~~
```

```
        [=](decltype((x)) y) {
              decltype((x)) z = x;
        };       // error: x refers to local entity but is in the lambda's
parameter-declaration-clause Ok, y has type float&, z has type float const&
        [=]{
            []<decltype(x) P>; // OK, x is in the outer lambda's function parameter scope
            [](decltype((x)) y){};    // OK, lambda takes a parameter of type float const&
            [x=1](decltype((x)) y){
                decltype((x)) z = x;
            }; // error: x refers to init-capture but is in the lambda's parameter-declaration-clause
ok, y has type int&, z has type int const&
        };
    }
```

— end example]

Acknowledgments

Thanks to Barry Revzin and Davis Herring for their help with the wording, and to the many people who gave feedback on this issue.

References

[1] Walter E Brown. P1753R0: Name lookup should “find the first thing of that name”. https://wg21.link/p1753r0, 6 2019.

[2] Ryan McDougall. P2034R0: Partially mutable lambda captures. https://wg21.link/p2034r0, 1 2020.

[3] Barry Revzin. P2036R3: Changing scope for lambda trailing-return-type. https://wg21.link/p2036r3, 9 2021.