Document: D2034R1
Author: Ryan McDougall <rmcdougall@aurora.tech>
Patrick McMichael <patrick@aurora.tech>
Audience: EWG-I
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Partially Mutable Lambda Captures

Or

A More Uniform Const for Lambdas

Revision History

Changes from R0: Concerns from EWG-I

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

Background

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

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

A type that is "logically const" is a type that has some mutable members that do not fundamentally change the invariants of the object, even when it is const. This means `any_invocable`, and *any* other const-correct library, *cannot* work with logically const lambdas.

Motivation

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

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

  void operator()(Timestamp t) const {
    callback_(state_, accumulator_, t);
  }
};

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

Moreover, a classic use for mutable members in bespoke classes is `std::mutex`.

```
struct MyThreadedAnalyzer {
  const State& state_;
  std::mutex& mtx_;

  void operator()(Slice slice) const {
    std::lock_guard<std::mutex> lock{mtx_};
    analyze(state_, slice);
  }
};

concurrent::queue<any_invocable<void(Slice) const> queue;
queue.push(MyThreadedAnalyzer{s, m});
```

Lambdas in such cases require work-arounds, such as abandoning logical const correctness, or using intermediary types (such as `std::ref`) that do not propagate constness.

Proposal

Mutable Capture By Value

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

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

// equivalent to:
```

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

Before	After

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

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

`// **loss of regular value type**`
`any_invocable<void()> f =`
`  [s, buf_ptr = &b]() mutable {`
`    // ...`
`  };`
	`any_invocable<void() const> f =`
	`  [s, mutable buf = b] {`
	`    // ...`
	`  };`

Possible Extensions

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

1. Const Capture on Mutable Call Operator

If lambda capture initialization can be modified by `mutable` and lambda call can be modified by `mutable`, then lambda calls modified by `mutable` should be able to declare some of their captures `const`.

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

// equivalent to:

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

Before	After

struct A {
  const float *iter;
  const float *const end;
  void operator()() {
    for(; iter != end; ++iter) {
      // end never modified...
    }
  }
};
 	
// **manual bespoke type**
any_invocable<void()> f = A{
  a.cbegin(), a.cend()
};
	any_invocable<void()> f = [
	   iter = a.cbegin(),
	   const end = a.cend()
	  ] () mutable {
	    for(; iter != end; ++iter) {
	      // end never modified...
	    }
	  };

// **extraneous mutable copy**
any_invocable<void()> f = [
   iter = a.cbegin(),
   end = a.cend()
  ] () {
    auto copy = iter;
    for(; copy != end; ++copy) {
      // end never modified...
    }
  };
	any_invocable<void()> f = [
	   iter = a.cbegin(),
	   const end = a.cend()
	  ] () mutable {
	    for(; iter != end; ++iter) {
	      // end never modified...
	    }
	  };

2. Const Capture by Reference

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

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

// equivalent to:

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

Before	After

struct A {
  const Huge &huge;
  void operator()() const {
    // huge.mutate(); is error
  }
};
 	
// **manual bespoke type**
any_invocable<void() const> f = A{huge};
	any_invocable<void() const> f =
	  [const &huge] {
	    // huge.mutate(); is error
	  };

// **extraneous cast**
any_invocable<void() const> f = [
  &huge = static_cast<const Huge&>(huge)] {
    // huge.mutate(); is error
  };
	any_invocable<void() const> f =
	  [const &huge] {
	    // huge.mutate(); is error
	  };

X a, b, c;
a = foo();
b = bar();
c = baz();
{
  // **manual redeclaration and assignment**
  const X& const_a = a;
  const X& const_b = b;
  const X& const_c = c;
  // ... enter const context
}
	X a, b, c;
	a = foo();
	b = bar();
	c = baz();
	
	[const &] {
	  // ... const context
	}();

3. Const Call Operator

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

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

// equivalent to:

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

4. Const Capture on Const Call Operator

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

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

// equivalent to:
```

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

5. Mutable Capture on Mutable Call Operator

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

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

// equivalent to:

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

Benefits of Consistency and Symmetry

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

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

For example:

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

// equivalent to:

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

Concerns

1. East v. West Const

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

2. Pointer to Const v. Const Pointer

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

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

3. Interactions with `this`

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

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

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

Whereas the following would compile and work:

```
struct B {
    void mutate() {}
};

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

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

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

4. Interactions with init-capture Packs

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

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

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

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

That said, this proposal is easily separable.

Thanks

Thanks Patrick McMichael for suggesting the idea. Thanks to Matt Calabrese for offering important corrections.