Document: P2034R0
Author: Ryan McDougall <mcdougall.ryan@gmail.com>
Audience: EWG-I
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Partially Mutable Lambda Captures

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 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_;
  mutable 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

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 =`	`any_invocable<void() const> f =`

`  [s, b]() mutable {`
`    // ...`
`  };`
	`  [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]() {`
	`    // ...`
	`  };`

`struct B {`
`  const State& state;`
`  mutable std::mutex& mtx;`
`  void operator() const {`
`    // ...`
`  }`
`};`
 	
`// manual bespoke type`
`any_invocable<void() const> f = B{s, m};`
	`any_invocable<void() const> f =`
	`  [&s, mutable &m]() {`
	`    // ...`
	`  };`

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

`// manual non-const-propagating wrapper`
`any_invocable<void()> f =`
`  [&s, mtx = std::ref(m)]() mutable {`
`    // ...`
`  };`
	`any_invocable<void() const> f =`
	`  [&s, mutable &mtx = m]() {`
	`    // ...`
	`  };`

Possible Extensions

1. 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() {}
};
```

2. For full symmetry it should be allowed to declare the lambda call `const` -- just as you are able in a bespoke callable function object. Presumably the user would declare

the lambda `mutable` or `const` according to ideal semantics, and some minority of capture initialization would be the opposite, as an exception.

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

// equivalent to:

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

Thanks

Credit to my colleague Patrick McMichael for suggesting the idea and reviewing the draft.