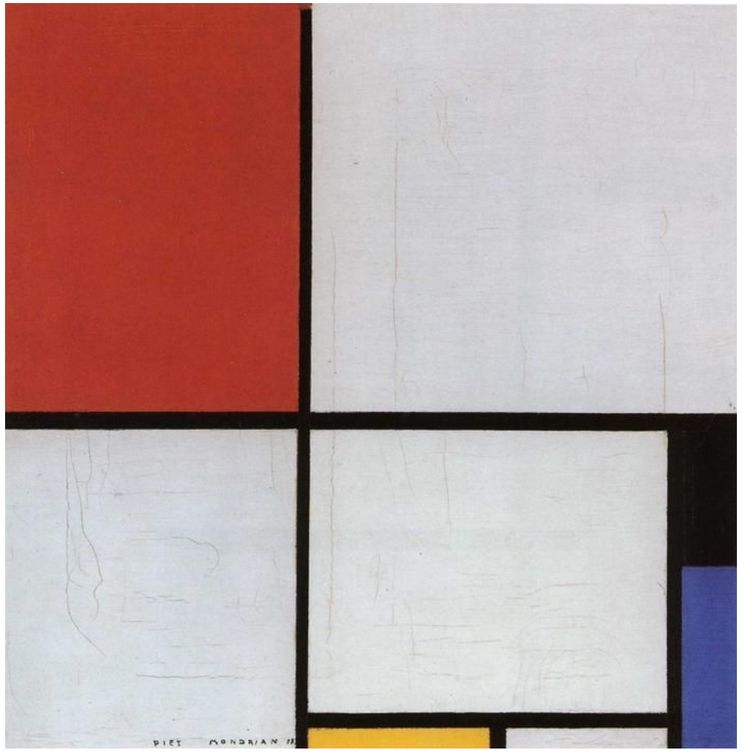
DEDUCING `this` PATTERNS



BEN DEANE / @ben_deane

CPPCON 2021

OVERVIEW

This presentation is about P0847, "Deducing `this`" (wg21.link/p0847)

Gašper Ažman, Sy Brand, Ben Deane, Barry Revzin

–

"We propose a new mechanism for specifying or deducing the value category of the expression that a member-function is invoked on. In other words, a way to tell from within a member function whether the expression it’s invoked on is an lvalue or an rvalue; whether it is const or volatile; and the expression’s type."

OVERVIEW

- A bit about the motivation and history of P0847
- What the feature is
- Existing patterns evolved by P0847
- New patterns not possible before P0847
- Miscellanea

Disclaimer: all code is exposition only and probably has some bugs
(but some has been tested using https://circle.godbolt.org/)

FIRST, AN ORIGIN STORY

(and timeline of sorts)

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.
- Late 2017: Gašper and I discover that Sy Brand and Barry Revzin are working on the same idea. We all join forces.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.
- Late 2017: Gašper and I discover that Sy Brand and Barry Revzin are working on the same idea. We all join forces.
- May 2018: Gašper talks at C++Now about the paper.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.
- Late 2017: Gašper and I discover that Sy Brand and Barry Revzin are working on the same idea. We all join forces.
- May 2018: Gašper talks at C++Now about the paper.
- June 2018: The paper is presented for the first time to WG21 in Rapperswil.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.
- Late 2017: Gašper and I discover that Sy Brand and Barry Revzin are working on the same idea. We all join forces.
- May 2018: Gašper talks at C++Now about the paper.
- June 2018: The paper is presented for the first time to WG21 in Rapperswil.
- July 2021: The paper is now at R7 and heading for standardization.

FIRST, AN ORIGIN STORY

(and timeline of sorts)

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.
- Late 2017: Gašper and I discover that Sy Brand and Barry Revzin are working on the same idea. We all join forces.
- May 2018: Gašper talks at C++Now about the paper.
- June 2018: The paper is presented for the first time to WG21 in Rapperswil.
- July 2021: The paper is now at R7 and heading for standardization.
- October 2021: P0847 passes a plenary vote: approved for C++23!

THE PROBLEM

- Mid 2013: I start playing with monadic futures. I run into a problem.

```
auto value = /* some heap thing */;

// capture-by-move in a callback
auto callback = [value = std::move(value)] (auto fn) {
  fn(value);
};

// pass the callback off to be executed in the future
pass_to_async_op(callback);
```

In general, how does the callback know if it's safe to `move` the captured thing?

WORKING THE PROBLEM

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.

  At this point, the response is: "we have bigger fish to fry."

  Fair enough; we did. The prevailing attitude then was the "tick - tock" release cycle.

- C++11: big
- C++14: small (ish?)
- C++17: big?

WORKING THE PROBLEM

So I go away and let the issue go. It stays in the back of my mind…

- Mid 2013: I start playing with monadic futures. I run into a problem.
- May 2014: I go to C++Now for the first time.
- May 2015: I go back to C++Now and tell folks about the problem.
- September 2017: I talk to Gašper Ažman at CppCon.

  This is where it picks up again.

INITIAL COMMITTEE RECEPTION

We talked to folks on the committee, and got interesting insights into how it works.

INITIAL COMMITTEE RECEPTION

We talked to folks on the committee, and got interesting insights into how it works.

Library folks: "I want this yesterday! So. Many. Places. To. Use. It."

INITIAL COMMITTEE RECEPTION

We talked to folks on the committee, and got interesting insights into how it works.

Library folks: "I want this yesterday! So. Many. Places. To. Use. It."

Language folks: "I don't really see the use case…"

INITIAL COMMITTEE RECEPTION

We talked to folks on the committee, and got interesting insights into how it works.

Library folks: "I want this yesterday! So. Many. Places. To. Use. It."

Language folks: "I don't really see the use case…"

(Arguably this is how a healthy committee should work:
language folks ought to be parsimonious with new features.)

THE FEATURE

Here's what we came up with:

```
struct S {
  template <typename Self>
  auto func(this Self&& self);
};
```

By the end of this talk, you will understand this slide and (more importantly) its implications.

One of the satisfying things about this feature is that it doesn't introduce new magic - as we shall see.

WHAT CHANGED, AND WHAT DIDN'T

We added `this` as an annotation (on the first parameter only) to make the "implicit first argument" *explicit* on non-static member functions.

Practically *everything else* is unchanged:

WHAT CHANGED, AND WHAT DIDN'T

We added `this` as an annotation (on the first parameter only) to make the "implicit first argument" *explicit* on non-static member functions.

Practically *everything else* is unchanged:

- template deduction rules

WHAT CHANGED, AND WHAT DIDN'T

We added `this` as an annotation (on the first parameter only) to make the "implicit first argument" *explicit* on non-static member functions.

Practically *everything else* is unchanged:

- template deduction rules
- overload resolution

WHAT CHANGED, AND WHAT DIDN'T

We added `this` as an annotation (on the first parameter only) to make the "implicit first argument" *explicit* on non-static member functions.

Practically *everything else* is unchanged:

- template deduction rules
- overload resolution
- name lookup (mostly)

WHAT CHANGED, AND WHAT DIDN'T

We added `this` as an annotation (on the first parameter only) to make the "implicit first argument" *explicit* on non-static member functions.

Practically *everything else* is unchanged:

- template deduction rules
- overload resolution
- name lookup (mostly)
- `this` is still a pointer, not a reference (you can be sad)

WHAT CHANGED, AND WHAT DIDN'T

We added `this` as an annotation (on the first parameter only) to make the "implicit first argument" *explicit* on non-static member functions.

Practically *everything else* is unchanged:

- template deduction rules
- overload resolution
- name lookup (mostly)
- `this` is still a pointer, not a reference (you can be sad)
- the meaning of `this` inside a lambda is unchanged

YOU ALREADY KNOW HOW THIS WILL WORK

We often think of the "this" parameter as the implicit first parameter to member functions (and languages like Python already make it explicit).

After this addition, all the usual rules of C++ can be applied.

YOU ALREADY KNOW HOW THIS WILL WORK

We often think of the "this" parameter as the implicit first parameter to member functions (and languages like Python already make it explicit).

After this addition, all the usual rules of C++ can be applied.

If there is a logical deduction from the rules you know, it's very probably how the feature works. In other words, this works as you would expect.

SOME QUICK EQUIVALENCES

Before:

```
struct S {
  auto func() const &;
  auto func() &;
  auto func() &&;
};
```

After:

```
struct S {
  auto func(this const S&);
  auto func(this S&);
  auto func(this S&&);
};
```

Note: both forms are still valid!

(But don't define both forms; that would be an error. They are morally *the same thing*.)

THE OBVIOUS WIN

The first obvious win is deduplicating
(triplicating? quadruplicating? octuplicating? etc?)

Before:

```
struct S {
  auto func() const &;
  auto func() &;
  auto func() &&;

  // plus all the overloads we seldom bother to write, yes you know the ones...
  // const rvalue refs? maybe...
  // volatile and const volatile, anyone?
};
```

THE OBVIOUS WIN

The first obvious win is deduplicating
(triplicating? quadruplicating? octuplicating? etc?)

After:

```
struct S {
  template <typename Self>
  auto func(this Self&& s);

  // and the compiler synthesizes all the overloads we ask for
};
```

GETTERS

Before P0847, you have several unpalatable choices:

GETTERS

Before P0847, you have several unpalatable choices:

- Neglect some overloads (probable)

GETTERS

Before P0847, you have several unpalatable choices:

- Neglect some overloads (probable)
- Write them all separately

GETTERS

Before P0847, you have several unpalatable choices:

- Neglect some overloads (probable)
- Write them all separately
- Delegate to a particular one (`const_cast` etc)

GETTERS

Before P0847, you have several unpalatable choices:

- Neglect some overloads (probable)
- Write them all separately
- Delegate to a particular one (`const_cast` etc)
- Delegate to a template helper

GETTERS

Before P0847, you have several unpalatable choices:

- Neglect some overloads (probable)
- Write them all separately
- Delegate to a particular one (`const_cast` etc)
- Delegate to a template helper

All these choices involve copy-pasting that is *required* for member functions.

GETTERS

With P0847:

```
struct S {
  std::string str;

  template <typename Self>
  auto get(this Self&& s) -> decltype(auto) /* or see next slide */ {
    return std::forward<Self>(s).str;
  }
};
```

One function template does it all.

GETTER RETURN TYPES

We may need to employ a little recipe for the return type.

```
struct S {
  std::string str;

  template <typename Self>
  auto get(this Self&&) -> like_t<Self, std::string>&&;
};
```

`like_t` is a metafunction that works as follows:

- `like_t<int&, double>` -> `double&`
- `like_t<const int&&, double>` -> `const double&&`
- etc.

With this recipe, we can write a single, optimal, template method.
(Think of value wrappers like `optional`.)

`like_t` AND `forward_like`

- `like_t<int&, double>` -> `double&`
- `like_t<const int&&, double>` -> `const double&&`
- etc.

`forward_like<T>(u)` -> `forward<like_t<T, decltype(u)>>(u)`

These are generally useful in writing wrappers, etc with deduced `this`.

wg21.link/p2445 (published this week by Gašper Ažman) is `forward_like`

OK, THAT WAS A SIMPLE WIN

There's more. Template deduction is unchanged.

```
struct Base {
  auto func1(this const Base& self);

  template <typename Self>
  auto func2(this Self&& self);
};

struct Derived : Base {};

void example() {
  Base b{};
  b.func1(); // self has type const Base&
  b.func2(); // self has type Base&

  Derived d{};
  d.func1(); // self has type const Base&
  d.func2(); // self has type Derived&
}
```

CRTP

Pre P0847, CRTP is a case of following the pattern.

```
template <typename T>
struct NumericalFunctions {
  void scale(double multiplicator) {
    T& underlying = static_cast<T&>(*this);
    underlying.setValue(underlying.getValue() * multiplicator);
  }
};

struct Sensitivity : NumericalFunctions<Sensitivity> {
  double getValue() const;
  void setValue(double value);
};
```

(From https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/)

CRTP WITH P0847

```
struct NumericalFunctions {
  template <typename Self>
  void scale(this Self&& self, double multiplicator) {
    self.setValue(self.getValue() * multiplicator);
  }
};

struct Sensitivity : NumericalFunctions {
  double getValue() const;
  void setValue(double value);
};
```

No class template, no recursion, it's just… er… P I guess?

MORE SIMPLIFICATIONS OF CRTP

Consider a builder pattern:

```
struct Builder {
  Builder& a() { /* ... */; return *this; }
  Builder& b() { /* ... */; return *this; }
};

Builder{}.a().b().a();
```

Now, what if we want to make a specialized builder that knows how to do extra stuff?

MORE CRTP

```
template <typename D=void>
class Builder {
  using Derived = conditional_t<is_void_v<D>, Builder, D>;
  Derived& self() {
    return *static_cast<Derived*>(this);
  }

public:
  Derived& a() { /* ... */; return self(); }
  Derived& b() { /* ... */; return self(); }
};

struct Special : Builder<Special> {
  Special& c() { /* ... */; return *this; }
  Special& d() { /* ... */; return *this; }
};

Builder{}.a().b().a();
Special{}.a().c().d().a();
```

CURIOUSER AND CURIOUSER

Now, what if we want a super-special-builder?

```
// Builder is as before

// Special needs to do more metaprogramming
template <typename D=void>
struct Special
  : Builder<conditional_t<is_void_v<D>, Special<D>, D> {
  using Derived = typename Special::Builder::Derived;
  Derived& c() { /* ... */; return this->self(); }
  Derived& d() { /* ... */; return this->self(); }
};

struct Super : Special<Super> {
  Super& e() { /* ... */; return *this; }
};

Builder{}.a().b().a();
Special{}.a().c().d().a();
Super{}.a().d().e();

// phew... getting really complicated
```

WITH P0847, IT'S SIMPLER

```
struct Builder {
  template <typename Self>
  Self& a(this Self&& self) { /* ... */; return self; }

  template <typename Self>
  Self& b(this Self&& self) { /* ... */; return self; }
};

struct Special : Builder {
  template <typename Self>
  Self& c(this Self&& self) { /* ... */; return self; }

  template <typename Self>
  Self& d(this Self&& self) { /* ... */; return self; }
};

struct Super : Special {
  template <typename Self>
  Self& e(this Self&& self) { /* ... */; return self; }
};
```

All the same flat pattern.

SO WE HAVE IMPROVEMENTS

SO WE HAVE IMPROVEMENTS

- more functionality for less code

SO WE HAVE IMPROVEMENTS

- more functionality for less code
- more comprehensive functions (compiler makes the overloads for us)

SO WE HAVE IMPROVEMENTS

- more functionality for less code
- more comprehensive functions (compiler makes the overloads for us)
- no explosion of complexity with CRTP layering

SO WE HAVE IMPROVEMENTS

- more functionality for less code
- more comprehensive functions (compiler makes the overloads for us)
- no explosion of complexity with CRTP layering

These are all great, but fundamentally they aren't new.

SO WE HAVE IMPROVEMENTS

- more functionality for less code
- more comprehensive functions (compiler makes the overloads for us)
- no explosion of complexity with CRTP layering

These are all great, but fundamentally they aren't new.

Let's look at some things that aren't possible yet, but P0847 makes possible.

BY-VALUE `this`

If we can deduce the type of the object argument, why does it have to be a reference?

```
struct less_than {
  template <typename T, typename U>
  bool operator()(this less_than, const T& lhs, const U& rhs) {
    return lhs < rhs;
  }
};

less_than{}(4, 5);
```

Nothing new here: you know how this works. It's regular pass-by-value.

BY-VALUE `this`: SMALL OBJECTS

We actively *want* to pass small (or empty!) types by value.

But pre-P0847, we *cannot* pass them by value to their own member functions.

The result is leaving perf on the table. Imagine:

```
template <class...>
class basic_string_view {
private:
  const_pointer data_;
  size_type size_;
public:
  constexpr const_iterator begin(this basic_string_view self) {
    return self.data_;
  }

  // etc for other members
};
```

BY-VALUE `this`: MOVE CHAINS

Similar to the builder pattern idea, sometimes we want to move rvalues through computations.

```
struct my_vector : vector<int> {
  using vector<int>::vector;

  auto sorted_by(this my_vector self, auto comp) -> my_vector {
    sort(self.begin(), self.end(), comp);
    return self;
  }
};

// here's an rvalue that is moved through the computation
my_vector{3,1,4,1,5,9,2,6,5}.sorted_by(less_than);
```

We can do this today with free functions; we can't do it with member functions.

BY-VALUE `this`: LIFETIME MANAGEMENT

Lifetime of coroutines can be tricky:

```
struct C {
  int val;

  auto always() const -> std::generator<int> {
    for (;;) { co_yield val; }
  }
};

// range-for init is needed here, but not obvious
for (int i : C{42}.always()) { ... }
```

BY-VALUE `this`: LIFETIME MANAGEMENT

With P0847's by-value `this`:

```
struct C {
  int val;

  auto always(this const C c) -> std::generator<int> {
    for (;;) { co_yield c.val; }
  }
};

// C is taken by value, so doesn't dangle
for (int i : C{42}.always()) { ... }
```

BY-VALUE `this` IMPLICATIONS

Nothing special here. We reasoned through the implications of how regular C++ rules interact with deduced `this`.

Passing `this` by value works as we expect.

BY-VALUE `this` IMPLICATIONS

Nothing special here. We reasoned through the implications of how regular C++ rules interact with deduced `this`.

Passing `this` by value works as we expect.

- Better perf for small objects (including captureless lambdas) - no need to form a reference and pass it

BY-VALUE `this` IMPLICATIONS

Nothing special here. We reasoned through the implications of how regular C++ rules interact with deduced `this`.

Passing `this` by value works as we expect.

- Better perf for small objects (including captureless lambdas) - no need to form a reference and pass it
- New ability to move values through computations without templates

BY-VALUE `this` IMPLICATIONS

Nothing special here. We reasoned through the implications of how regular C++ rules interact with deduced `this`.

Passing `this` by value works as we expect.

- Better perf for small objects (including captureless lambdas) - no need to form a reference and pass it
- New ability to move values through computations without templates
- New options for lifetime safety, now that we can copy the object

THE ORIGINAL PROBLEM

```
auto value = /* some heap thing */;

// capture-by-move in a callback
auto callback = [value = std::move(value)] (this auto&& self, auto fn) {
  fn(forward_like<decltype(self)>(value));
};

// pass the callback off to be executed in the future
pass_to_async_op(callback);
```

Now the callback knows how to treat the captured object!

WHAT ELSE?

Let's talk a bit about SFINAE-friendly callables.

(Warning, more template code ahead)

CALLABLE WRAPPERS

```
template <typename F>
struct not_wrapper {
  F f;

  template <typename... Args>
  auto operator()(Args&&...) &
    -> decltype(not declval<invoke_result_t<F&, Args...>>());

  template <typename... Args>
  auto operator()(Args&&...) const &
    -> decltype(not declval<invoke_result_t<F const&, Args...>>());

  /// etc...
};

template <typename F>
auto not_fn(F&& f) { return not_wrapper<decay_t<F>>{forward<F>(f)}; }
```

PATHOLOGICAL CALLABLES, PART 1

```
struct sfinae_unfriendly {
  template <typename T>
  auto operator()(T) {
    static_assert(is_same_v<T, int>);
  }

  template <typename T>
  auto operator()(T) const {
    static_assert(is_same_v<T, double>);
  }
};

not_fn(sfinae_unfriendly{})(1); // hits static_assert
// even though the non-const overload is the best match,
// all the overloads get instantiated... boom
```

PATHOLOGICAL CALLABLES, PART 2

```
struct trying_to_be_safe {
  template <typename T>
  auto operator()(T&&) = delete;

  template <typename T>
  auto operator()(T&&) const { return true; }
};

not_fn(trying_to_be_safe{})(1); // returns false (i.e. not true)
// even though the non-const overload is the best match,
// and we want that case to give us "deleted"
```

CALLABLE WRAPPERS WITH P0874

```
template <typename F>
auto not_fn(F&& f) {
  return [f = forward<F>(f)] (this auto&& self, auto&&... args)
    BOOST_HOF_RETURNS(
      not invoke(forward_like<decltype(self)>(f),
                 forward<decltype(args)>(args)...));
}
```

BOOST_HOF_RETURNS is a macro that avoids the "you must repeat it 3 times" problem.
(trailing return type, noexcept clause, body)

MORE ON LAMBDAS

A few things to cover here:

MORE ON LAMBDAS

A few things to cover here:

- `this` inside the lambda

MORE ON LAMBDAS

A few things to cover here:

- `this` inside the lambda
- recursive lambda

MORE ON LAMBDAS

A few things to cover here:

- `this` inside the lambda
- recursive lambda
- deriving from lambdas, overload sets

MORE ON LAMBDAS

A few things to cover here:

- `this` inside the lambda
- recursive lambda
- deriving from lambdas, overload sets
- caveats

MORE ON LAMBDAS

We must use language a bit carefully to distinguish between:

MORE ON LAMBDAS

We must use language a bit carefully to distinguish between:

- a lambda expression (the code that we write)

MORE ON LAMBDAS

We must use language a bit carefully to distinguish between:

- a lambda expression (the code that we write)
- a closure object (the object the compiler generates by evaluating the lambda expression)

DEDUCING `this` AND LAMBDAS

First, uniformity of syntax between regular member functions and lambda expressions was one major factor deciding syntax.

```
struct S {
  int i;

  // P0847 syntax
  auto get(this S& self) { return s.i; }

  // alt 1: first parameter must be named "this"
  auto get(S& this) { return this.i; }
  // alt 2: add a type marker & name to the existing cv-ref-qualifier place
  auto get() S& self { return self.i; }
  // alt 3: add just a type marker to the existing place, use this
  auto get() S& { return this->i; }
};

// consistent syntax
auto lambda = [] (this auto& self) { ... };
```

`this` INSIDE LAMBDA

Importantly, `this` inside a lambda expression still means what it always has.

You get to name the explicit object parameter, and you obviously can't call it a keyword, so `this` still means a captured `this`.

`this` INSIDE LAMBDA

Importantly, `this` inside a lambda expression still means what it always has.

You get to name the explicit object parameter, and you obviously can't call it a keyword, so `this` still means a captured `this`.

(You can still be sad that `this`-capture is a grandfathered-in special case wart in C++ lambda expressions, but at least it won't break…)

RECURSIVE LAMBDAS (BASIC)

At last, this is what it's all about, right?

```
auto fib = [] (this auto self, int n) {
  if (n < 2) return n;
  return self(n-1) + self(n-2);
}
```

- The lambda must be generic (fairly obviously; you can't spell the type)
- Don't actually implement Fibonacci this way; see *Elements of Programming* §3.6

RESURSIVE LAMBDAS (FANCIER)

Remember, normal type deduction will deduce the derived type…

```
struct Leaf {};
struct Node;
using Tree = variant<Leaf, Node*>;
struct Node { Tree left; Tree right; };

template <typename... Fs> struct overloaded : Fs... { using Fs::operator()...; };

int num_leaves(const Tree& tree) {
  return visit(overloaded{
      [] (const Leaf&) { return 1; },
      [] (this const auto& self, const Node* n)-> int {
        return visit(self, n->left) + visit(self, n->right);
      }
    }, tree);
}
```

BY-VALUE LAMBDAS

With P0847, maybe non-capturing lambdas should conventionally be by-value?

```
auto lam = [] (this auto self, int x, int y) { ... };
```

The compiler can completely optimize away the reference-forming and passing of the empty object.

This achieves the same goal as part of wg21.link/p1169 (static `operator()`).

Many people have a mental model that the `operator()` of a non-capturing lambda should be static. But it *isn't* static…

LAMBDA CAVEATS

When we write this:

```
int x = 42;
auto l = [&] (int n) { return x + n; }
```

The compiler generates (approximately) this, right?

```
int x = 42;

class __lambda {
  int& x;
public:
  __lambda(int &_x) : x{_x} {}
  constexpr int operator()(int n) const { return x + n; }
} l{x};
```

LAMBDA CAVEATS

The (standard) truth is…

No!

(Despite what cppinsights tells you…)

CLOSURE OBJECTS, PER THE STANDARD

"An implementation may define the closure object differently…"

In particular, closure objects have:

CLOSURE OBJECTS, PER THE STANDARD

"An implementation may define the closure object differently…"

In particular, closure objects have:

- no defined layout

CLOSURE OBJECTS, PER THE STANDARD

"An implementation may define the closure object differently…"

In particular, closure objects have:

- no defined layout
- no defined size/alignment

CLOSURE OBJECTS, PER THE STANDARD

"An implementation may define the closure object differently…"

In particular, closure objects have:

- no defined layout
- no defined size/alignment
- no defined member init order (no, it's not the capture order)

CLOSURE OBJECTS, PER THE STANDARD

"An implementation may define the closure object differently…"

In particular, closure objects have:

- no defined layout
- no defined size/alignment
- no defined member init order (no, it's not the capture order)
- no defined members (you aren't using member variables, just aliases…)

CLOSURE OBJECTS, PER THE STANDARD

"An implementation may define the closure object differently…"

In particular, closure objects have:

- no defined layout
- no defined size/alignment
- no defined member init order (no, it's not the capture order)
- no defined members (you aren't using member variables, just aliases…)

This is by design: anticipating that sufficiently clever implementations can make optimizations in this area (e.g. capturing several contiguous stack variables by reference with just one pointer).

WHAT DOES THAT MEAN FOR P0847?

We can do two things with an explicit object parameter on a lambda expression:

- use `decltype`
- call it

We cannot access "member variables" with `self.member`.

```
int x = 42;

auto l = [&x] (this auto&& self, int n) {
  using T = std::remove_cvref_t<decltype(self)>; // fine
  if (n > 0) {
    return self(n - 1); // also fine
  } else {
    return self.x + n; // nope, no such thing as self.x
  }
};
```

Then again, there should be no need to (just say plain `member`).

MISCELLANEA

Remaining observations, a few minutiae, etc.

WE COULDN'T DO THIS BEFORE...

Lambda extra: we can now constrain the closure object's value category?

```
// C++20 - how do I write this as a lambda?
// I only want to use l-values...
struct lam_20 {
  auto operator()(int) & { ... };
};

// With P0847, I can
auto lam_23 = [] (this auto& self, int) { ... };

lam_20{}(42);            // compile error
decltype(lam_23){}(42); // also compile error with P0847
```

WE COULDN'T DO THIS BEFORE...

Anywhere we can put a template, we can put a concept.

Methods can be made safer with P0847, because they can constrain the derived type of the explicit object parameter.

```
struct Base {
  auto func(this std::regular auto&& self) {
    // whatever derives from me needs to be regular to call func!
  }
};
```

CONSIDER "TEMPLATE METHOD"

Recall the template method pattern
(*Design Patterns*, Gamma/Helm/Johnson/Vlissides pp 325-330)

```
struct Base {
  auto TemplateMethod() {
    PrimitiveOperation1();
    PrimitiveOperation2();
    ...
  }

  virtual void PrimitiveOperation1() = 0;
  virtual void PrimitiveOperation2() = 0;
};

struct Derived : Base {
  void PrimitiveOperation1() override { ... };
  void PrimitiveOperation2() override { ... };
};
```

TEMPLATE METHOD WITH P0847

Recall the template method pattern
(*Design Patterns*, Gamma/Helm/Johnson/Vlissides pp 325-330)

```
template <typename T>
concept tm_concept = has_primop_1<T> and has_primop_2<T>;

struct Base {
  template <tm_concept Self>
  auto TemplateMethod(this const Self& self) {
    self.PrimitiveOperation1();
    self.PrimitiveOperation2();
  }
};

struct Derived : Base {
  auto PrimitiveOperation1() { ... }
  auto PrimitiveOperation2() { ... }
};
```

Notice: this is morally equivalent to CRTP.

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated
5. generically usable

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated
5. generically usable

Before P0847, we couldn't get all of these properties.

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated
5. generically usable

Before P0847, we couldn't get all of these properties.

- CTRP is not simple

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated
5. generically usable

Before P0847, we couldn't get all of these properties.

- CTRP is not simple
- Virtual functions are not statically optimized

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated
5. generically usable

Before P0847, we couldn't get all of these properties.

- CTRP is not simple
- Virtual functions are not statically optimized
- Free functions are not best encapsulated

TEMPLATE METHOD

With P0847, the Template Method pattern is finally all of:

1. simply expressed
2. statically optimized
3. safely constrained
4. appropriately encapsulated
5. generically usable

Before P0847, we couldn't get all of these properties.

- CTRP is not simple
- Virtual functions are not statically optimized
- Free functions are not best encapsulated
- `friend` functions are not "UFCS-callable"

WHAT IS IT REALLY?

A method with an explicit object parameter is perhaps more like a free function than a member function.

So to prevent confusion, name lookup changes (just a bit).

```
struct S {
  auto f(this const S& self) {
    return i;       // ill-formed: no implicit access to members
    return this->i; // ill-formed: no this access
    return self.i;  // fine: you named it, you use it
  }

  int i;
};
```

WHAT IS IT REALLY?

Deduction-to-derived type means sometimes we need to defend against shadowing…

```
struct Base {
  int i;

  auto f(this auto&& self) {
    return self.Base::i;
  }
};

struct Derived : Base {
  double i;
};
```

Of course such badly-behaved classes are ill-advised. But this is the sort of thing that comes up in standardization…

HOW DOES ACCESS WORK?

```
struct Base {
  int i;

  auto f(this auto&& self) {
    return self.Base::i; // oh dear...
  }
};

struct Derived : private Base {
  double i;
};
```

More bad behaviour…

POINTER-TO-MEMBER-FUNCTION

New and old *do* the same thing, but aren't quite the same…

```
struct S {
  void f(int) const &;
  void g(this const S&, int);
};

S s;
s.f(42); // OK today
s.g(42); // OK with P0847

auto pf = &S::f;        // type: auto (S::*)(int) -> void
(s.*pf)(42);            // OK today
std::invoke(pf, s, 42) // OK today

auto pg = &S::g;        // type: auto (*)(const S&, int) -> void
pg(s, 42);              // OK with P0847
std::invoke(pg, s, 42) // still OK
```

ADT STYLE

The question arises: Why not write all member functions this way?

Why not "Always Deduce `this`"?

ADT STYLE ADVANTAGES

ADT STYLE ADVANTAGES

- Code deduplication/correctness-by-compiler

ADT STYLE ADVANTAGES

- Code deduplication/correctness-by-compiler
- Completeness-by-compiler

ADT STYLE ADVANTAGES

- Code deduplication/correctness-by-compiler
- Completeness-by-compiler
- Class interface gets "simpler & safer on the outside"
    - Constrained descendants
    - Simpler member function pointers

ADT STYLE ADVANTAGES

- Code deduplication/correctness-by-compiler
- Completeness-by-compiler
- Class interface gets "simpler & safer on the outside"
  - Constrained descendants
  - Simpler member function pointers
- Easier patterns

ADT STYLE ADVANTAGES

- Code deduplication/correctness-by-compiler
- Completeness-by-compiler
- Class interface gets "simpler & safer on the outside"
  - Constrained descendants
  - Simpler member function pointers
- Easier patterns
- Sometimes you'll want it, so why not be consistent?

ADT STYLE ADVANTAGES

- Code deduplication/correctness-by-compiler
- Completeness-by-compiler
- Class interface gets "simpler & safer on the outside"
  - Constrained descendants
  - Simpler member function pointers
- Easier patterns
- Sometimes you'll want it, so why not be consistent?
- Teachability?

ADT STYLE DISADVANTAGES

ADT STYLE DISADVANTAGES

- Templates
  - Implementation complexity
  - Compile time? (if you would avoid completeness)

ADT STYLE DISADVANTAGES

- Templates
  - Implementation complexity
  - Compile time? (if you would avoid completeness)
- Over-deduction complexity
  - You really just want const/mutable, but you get the whole value category enchilada

ADT STYLE DISADVANTAGES

- Templates
  - Implementation complexity
  - Compile time? (if you would avoid completeness)
- Over-deduction complexity
  - You really just want const/mutable, but you get the whole value category enchilada
- DT methods can't be `virtual` (yet…?)

GOOD OR BAD?

No `this` or implicit member access in DT methods - is this good or bad?

- More to type, but…
- No dependent base trip-ups?

CONCLUSIONS

A relatively small language addition, with effects and implications taken to their logical ends according to existing language rules. It works as you expect.

It will make lots of code easier, especially library code.

New patterns will become possible (we don't know what they all are yet).

https://wg21.link/P0847 / try it at https://circle.godbolt.org/

Thanks Barry, Gašper, Sy!