Template `this`

(Divining the value category of `*this`)

Document #: DxxxxR0
Date: 2017-10-29
Audience: Evolution Working Group
Reply-to: Ben Deane, ben at elbeno dot com
Gašper Ažman, gasper dot azman at gmail dot com

Contents

1 Introduction

We propose a new mechanism for specifying the value category of an instance of a class, which is visible from inside a member function of that class. In other words, a way to tell from within a member function whether one's `this` points to an rvalue or an lvalue, and whether it is `const` or `volatile`.

2 Motivation

The existing mechanism for this is to add reference, rvalue-reference, const or volatile qualifier (*cv-ref qualifiers*) suffixes to a member function. It suffers from the following problems.

It is verbose.

A common task in writing a class is providing a *getter function* to access a contained member. Authors that care about performance want to provide getter functions that take advantage of move semantics in the case where the class instance is an rvalue, and so end up writing several of these functions, which only differ in the *cv-ref qualifiers*.

There is no way to write a reference qualifier for a lambda expression.

When writing a lambda expression, it is impossible to know whether it is safe to move from captured-by-copy members of the closure object.

This impacts the performance of lambda expressions. One scenario where this is important arises when using lambda expressions as parts of a chain of asynchronous continuations that carry a value through a computation.

Example:

```
struct optimal_carrier {
  std::string message;
  std::string const& operator()() const & { return message; }
  std::string&& operator()() && { return std::move(message); }
};
// no way to have both && and const& call operators...
auto suboptimal_lambda = [message = std::string(text)]() -> auto const&
{
  return message;
};
```

There is no way to refer to a lambda expression from within itself.

The rationale in [P0839] (*Recursive lambdas* by Richard Smith) is valid and we'd like to transclude it here.

This paper provides an alternative to the proposed solution in that paper.

3 Design Considerations

In addition to solving the existing problems, desirable properties of a solution are:

- It should work the same way for member functions and lambda expressions.
- It should work like existing practice as much as possible, while adding as little extra syntax as possible.
- It should avoid adding extraneous syntax to function declarations. In a world where every function is fast becoming tagged with attributes, `constexpr`, `noexcept` specifications, and (this proposal notwithstanding) reference qualifiers, readable, noise-free function declarations are to be prized.
- The design should provide more uniformity to the language. With the desires for uniform call syntax, the way we define functions and member functions should be converging as much as possible.

4 Proposed Solution

We propose the ability to add an optional first parameter to any member function of a class `T`, taking the form `T [const] [volatile] [&|&&] this`.

To facilitate use in generic lambda expressions, this may also be formulated as `auto [const] [volatile] [&|&&] this`.

In all cases, the value category of `this` inside the member function is exactly what the existing parameter rules would already imply. In other words, the *cv-ref qualifiers* that stand after the function signature now explicitly apply to the `this` parameter.

With this, the above "value carrier" example becomes:

```
struct optimal_source {
  std::string message;
  template <typename Self>
  decltype(auto) operator()(Self&& this)
  {
    return std::forward<Self>(this).message;
  }
};
auto optimal_lambda
    = [message = std::string(text)](auto&& this) mutable -> decltype(auto)
{
  return std::forward_like<decltype(this)>(message);
};
```

Getter functions are the most affected. Here is an example of slightly more common code:

```
struct Person {
  std::string name;

  template <typename U>
  decltype(auto) GetName(U&& this) {
    return std::forward<T>(this).name;
  }
};
```

The template above will likely result in the instantiation of the member functions:

```
  std::string const& GetName() const& ;
  std::string& GetName() &;
  std::string&& GetName() &&;
```

These are all member functions we currently write by hand, and have to manually maintain their currect semantics.

5 What does `this` in a parameter list mean?

The meaning of the different ways to pass `this` is the same as current general parameter handling.

The entries of this table should be read as if they are inside a class `T`:

```
class T { /* entry */ };
```

In other words, `T` is *not* a template parameter.

written as	C++17 signature	comments
`void f(T this)`	currently not available	[value]
`void f(T& this)`	`void f() &`	
`void f(T&& this)`	`void f() &&`	
`void f(T const this)`	currently not available	[value]
`void f(T const& this)`	`void f() const&`	
`void f(T const&& this)`	`void f() const&&`	
`void f(T volatile this)`	currently not available	[value]
`void f(T volatile& this)`	`void f() volatile&`	
`void f(T volatile&& this)`	`void f() volatile&&`	
`void f(T const volatile this)`	currently not available	[value]
`void f(T const volatile& this)`	`void f() const volatile&`	
`void f(T const volatile&& this)`	`void f() const volatile&&`	

Notes:

- *[value]*: whether passing by value should be allowed is debatable, but seems desired for completeness and parity with inline friend functions.
- The interpretation of `this` in the member function body differs, but only one definition for a given signature may be present, eg. one may define at most one of `void f()&`, or `void f(T& this)` or `void f()`, the first and last already being exclusive of one another.

How does templated `this` work?

Using existing deduction rules for template parameters, which will deduce the type of `this` to something in the above table.

What does `this` mean in the body of a member function?

It behaves exactly as a regular parameter declared in the same way.

Constructors

No exceptions to the above rules. If a particular constructor signature is not allowed by the language, it continues to be disallowed. We can already access already-initialized members in initialization lists, which means `this` is already available, even though it hasn't been completely constructed yet.

What about pass-by-value member functions?

We think they are a logical extension of the mechanism, and would go a long way towards making member functions as powerful as inline friend functions, with the only difference being the call syntax.

One implication of this is that the `this` parameter would be move-constructed in the case where the object is an rvalue, allowing you to treat chained builder member functions that return a new object uniformly without having to resort to templates.

Example:

```
class string_builder {
  std::string s;

  operator std::string (string_builder this) {
    return std::move(s);
  }
  string_builder operator*(string_builder this, int n) {
    assert(n > 0);

    s.reserve(s.size() * n);
    auto const size = s.size();
    for (auto i = 0; i < n; ++i) {
      s.append(s, 0, size);
    }
    return this;
  }
  string_builder bop(string_builder this) {
    s.append("bop");
    return this;
  }
};

// this is optimally efficient as far as allocations go
std::string const x = (string_builder{{"asdf"}} * 5).bop().bop();
```

Of course, implementing this example with templated `this` member functions would have been slightly more efficient due to also saving on move constructions, but we got rid of all references in the program!

Writing the function pointer types for such functions

Currently, we write member function pointers like so:

```
struct Y {
  int f(int a, int b) const &;
};
static_assert(std::is_same_v<decltype(&Y::f), int (Y::*)(int, int) const &>);
```

All the member functions that take references already have a function pointer syntax - they are just alternate ways of writing functions we can already write.

The only one that does not have such a syntax is the pass-by-value method.

We would like to ask for suggestions for syntax for these function pointers. We give our first pass here:

```
struct Z {
  int f(Z const& this, int a, int b);
  // same as 'int f(int a, int b) const&;'
  int g(Z this, int a, int b);
};
// f is still the same as Y::f
static_assert(std::is_same_v<decltype(&Z::f), int (Z::*)(int, int) const &>);
// but would this alternate syntax make any sense?
static_assert(std::is_same_v<decltype(&Z::f), int (*)(Z::const&, int, int)>);
// It allows us to specify the syntax for Z as a pass-by-value member function
static_assert(std::is_same_v<decltype(&Z::g), int (*)(Z::, int, int)>);
```

Such an approach unifies, to a degree, the member functions and the rest of the function type spaces, since it communicates that the first parameter is special, but also communicates its type and calling convention.

`this` as a reference

This paper turns `this` into a reference on an opt-in basis, which is in line with the existing guidance that never-null pointers should be references if at all possible, and in this case, it is possible.

We believe there would be no confusion, as in all cases, the value category of `this` is stated plainly in the parameter list, which is on the very same screen.

Teaching also becomes easier, as the meaning of what a "const member function" is becomes more obvious to students.

One can always obtain the address of the object by taking the address of the `this`.

`virtual` and `this` as value

Virtual member functions are always dispatched based on the type of the object the dot (or arrow – in case of pointer) operator is being used on. Once the member function is located, the parameter `this` is constructed with the appropriate (move or copy) constructor, and passed as the `this` parameter. This might incur slicing.

Effectively, we want to say that there is no change from current behavior, only a slight addition of a new overload that behaves the way a user would expect.

Teachability implications

Using `auto&& this` follows existing patterns for dealing with forwarding references.

Optionally adding `this` as the first parameter fits with many programmers' mental model of the `this` pointer being the first parameter to member functions "under the hood" and is comparable to usage in other languages, e.g. Python and Rust.

It also works as a more obvious way to teach how `std::bind` and `std::function` work with a member function pointer work by making the pointer explicit.

ABI implications for `std::function` and related

If references and pointers do not have the same representation for member functions, this effectively says "for the purposes of `this`, they do."

This matters because code written in the "`this` is a pointer" syntax with the '`->`' notation needs to be assembly-identical to code written with the '`.`' notation, as the two are just different ways to implement a function with the same signature.

6 Implications for lambdas

Generic lambdas, should they take an `auto&& this` parameter, work according to existing rewriting rules: the `auto&& this` is turned into a "forwarding reference" and deduced as if it were inside a `template <typename T> auto operator()(T&& this) { ... }`.

Do we allow `this` in lambdas that decay to a function pointer?

If the lambda would otherwise decay to a function pointer, `&this` shall have the value of that function pointer.

Does this allow recursion in lambdas?

Yup. You're allowed to call `this(...)`.

Expressions allowed for `this` in lambdas

```
this(...);      // call with appropriate signature
decltype(this); // evaluates to the type of the lambda with the appropriate
                // cv-ref qualifiers
&this;          // the address of either the closure object or function pointer
std::move(this) // you're allowed to move yourself into an algorithm...
/* ... and all other things you're allowed to do with the lambda itself. */
```

In the case of lambda expressions, the `this` parameter still does allow one to refer to the members of the closure object directly (as it has no defined storage or layout, and its members do not have names). What it does instead is allow one to deduce the value category of the lambda and access its members, including the various call operators, in the way appropriate for the value category.

Interplays with capturing [this] and [*this]

this is passed as a parameter, and parameters shadow. Therefore, this inside the lambda body points to the closure, and its type is the type of the lambda, with the appropriate cv-ref qualifiers.

One loses the ability to refer to the object captured by *this directly through its *this pointer, but the members are very much still in scope.

Example:

```
struct X {
  int operator()(int, int) const;
  void g(int n) const;

  auto f()
  {
    // capture *this by copy
    auto h = [*this](auto&& this, int n)
    {
      // decltype(this) is decltype(h) [const][&|&&], depending on which call
      // operator is invoked.
      g(n); // members of X are still accessible
      if (n % 2) {
        // calls this lambda recursively. Forward the cv-ref qualifiers to
        // sub-call.
        return std::forward<decltype(this)>(this)(n + 1);
      }
      return operator()(n, 2); // calls X::operator().
    };
    return h;
  }
};
```

That said, it's not possible to refer to the members of the closure using the this pointer of the lambda, since the closure has no defined layout, its members do not have names, and the members referenced may not even be inserted into the closure, if one exists at all. One refers to the members of the closure as now - they are just in scope.

7 Impact on the Standard

TBD: A bunch of stuff in section 8.1.5 [expr.prim.lambda].

TBD: A bunch of stuff in that this can appear as the first member function parameter.

8 Further Directions

In this section, we explore the space of options that this feature might lead to. Please consider every section as its own mini-proposal - each one goes further into unexplored territory. However, knowing where a feature may lead is important for its assessment, which is why we are including this section.

Unification with inline friend functions

This proposal also makes this far less special. In fact, it almost unifies inline friend functions and class member functions, with the differences being:

- the calling syntax (member function vs free function)
- member functions can be virtual

Basically, if the first parameter is called `this`, one can parse and instantiate the declaration with exactly the same rules as an `inline friend`, except with a calling convention for member functions.

Opt-in uniform call syntax

The interaction of a `this` parameter with `friend` functions raises the possibility of opt-in uniform call syntax. Consider:

```
class Foo
{
  friend auto ufcs(Foo this, int x)
  {
    // this function can be called two ways
  }
};
```

A reasonable interpretation for defining a member function with `friend` *and* a `this` parameter is that it may be called as either a regular member function, or as a friend function. That is, either syntax could work.

```
  Foo f;
  f.ufcs(42);  // member function call syntax
  ufcs(f, 42); // friend (free) function call syntax
```

In the past, there has been considerable discussion whether UFCS prefers the member or the free function. In this case, the member and the free function are the very same function, so that question is rendered irrelevant.

Free functions

Given the above, we might choose to extend uniform call syntax to free functions by allowing `this` to appear in their signatures as well. Such (non-friend) functions would become callable with both syntaxes.

Providing both a member and a free-function with the same signature makes the program ill-formed, to again avoid the issue of preference.

```
std::pair<int, int>& add(std::pair<int, int>& this,
                         std::pair<int, int> const& other) {
  // only public members are visible, since not a friend
  first += other.first;
  second += other.second;
  return this;
}

auto x = pair{1, 2};
x.add({3, 4});
// x.first == 4, x.second == 6
```

There are legitimate reasons for disallowing this use-case. In c++17, one can always find the member function definitions in one specific place. This would remove this last refuge of obvious discoverability.

However, we are already in the same situation with free functions. The reviewers of this paper have held positions both for and against allowing this. It is a question for the committee.

Why stop at the first parameter?

Now that we have uniform call syntax, we might reasonably ask ourselves what `this` would mean if included as another, *not the first*, parameter.

The authors of this paper would argue that functions that do that, when called with regular invocation syntax, look exactly as if the parameter were named something else, except for the fact that the public members are in scope.

When called with the member function invocation syntax, the `this` parameter is dropped from the signature regardless of its location.

For some predicates, the reverse positioning of parameters makes perfect sense, as in the following example.

```

template <typename T>
bool has_element(T const& element, std::list<T> const& this) {
  return std::find(begin(), end(), element) != end();
}
std::list<int> my_list = {1, 2, 3, 4, 5, 6, 8};
my_list.has_element(6); // true
has_element(7, my_list); // false
```

How `this` as any parameter would play with operator overloading

Operators are one of the parts of the language that already have UFCS! They go through less change than the rest of the language.

Should one define a plus operator like so:

```
template <typename T, typename U>
std::pair<T, U> operator+(std::pair<T, U> x, std::pair<T, U> this) {
  return {x.first + first, x.second + second};
}
```

From the point of view of the user of the operator, nothing at all changes, not even the ABI. The only thing that changes is how one writes the function.

If called as `pair1, 2.operator+(pair3, 4`, the order of parameters is reversed, according to the above rules. (the call is actually `operator+(pair3, 4, pair1, 2)`).

The authors of this paper do not expect that the practice of writing code that works in surprising ways for little reason iike the above example will catch on, but we see no reason to prohibit it specifically for operators.

Do we allow this-position swapping for member functions?

Only inline friends. The practice has no meaning for functions that do not opt in to UFCS.

9 Acknowledgments

Thanks to the following for their help and guidance:

- Louis Dionne
- Marshall Clow
- Andrew Bennieston
- Tristan Brindle

References

[P0839] Richard Smith, *Recursive lambdas*
http://wg21.link/p0839r0

[P0018r3] H. Carter Edwards, Daveed Vandevoorde, Christian Trott, Hal Finkel, Jim Reus, Robin Maffeo, Ben Sander, *Capturing *`this` by value*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0018r3.html

[P0637r0] Thomas Köppe, *Capture *`this` With Initializer*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0637r0.html

A `forward_like`

We propose a new library facility, `forward_like`, that acts as `std::forward<T>(t).member` does for members, with the syntax `forward_like<T>(t.member)`. This is done because in the case of lambdas, the closure members are not actually addressable using `this.member`.

The proposed semantics are essentially the same as `std::forward`. If `From` and `To` are the same type, `forward_like` and `forward` act exacty the same.

Proposed definition:

```cpp
template <typename Like, typename T>
constexpr decltype(auto) forward_like(T&& t) noexcept
{
  // first, get ‘t‘ back into the value category it was passed in
  // then, forward it as if its value category was ‘Like‘’s.
  // This prohibits rvalue -> lvalue conversions.
  return std::forward<like_t<Like, T>>(std::forward<T>(t));
}
```

To do that, we require another facility in the standard library, `like`:

```cpp
template <typename From, typename To>
class like {
  template <bool Condition, template <typename> class Function, typename T>
  using apply_if = std::conditional_t<Condition, Function<T>, T>;
  using base = std::remove_cv_t<std::remove_reference_t<To>>;
  using base_from = std::remove_reference_t<From>;

  static constexpr bool rv = std::is_rvalue_reference_v<From>;
  static constexpr bool lv = std::is_lvalue_reference_v<From>;
  static constexpr bool c = std::is_const_v<base_from>;
  static constexpr bool v = std::is_volatile_v<base_from>;

public:
  using type = apply_if<lv, std::add_lvalue_reference_t,
               apply_if<rv, std::add_rvalue_reference_t,
               apply_if<c, std::add_const_t,
               apply_if<v, std::add_volatile_t,
               base>>>>;
};

template <typename From, typename To>
using like_t = typename like<From, To>::type;
```

It merely copies the cv-ref qualifiers from `From` to `To`.

The entire listing of the code with all the tests is available at https://github.com/atomgalaxy/isocpp-template-this/blob/master/forward_like.cpp.