std::forward for members (forward_like)

Document #:	P1361R0
Date:	2021-10-19
Project:	Programming Language C++
Audience:	Library Evolution Working Group Library Working Group

Reply-to:	Gašper Ažman <gasper.azman@gmail.com>

Contents

1 Introduction

Deducing This [P0847R7] is expected to land in C++23.

Its examples use a hypothetical `std::forward_like<decltype(self)>(variable)` facility because `std::forward<decltype(v)>(v)` is insufficient. This paper proposes an additional overload of `std::forward` to cater to this scenario.

2 Design Discussion

As `forward`, `forward_like` is a type cast that only influences the value category of an expression.

`forward_like` is a facility for forwarding the value category of an object-expression `m` (usually a member) based on the value category of the owning object-expression `o`.

When `m` is an actual member and thus `o.m` a valid expression, this is usually spelled as `forward<decltype(o)>(o).m` in C++20 code.

When `o.m` is not a valid expression, *i.e.* members of lambda closures, one needs `forward_like</*see below*/>(m)`.

This leads to three possible models, called **merge**, **tuple**, and **language**.

— **merge**: we merge the `const` qualifiers, and adopt the value category of the Owner
— **tuple**: what `std::get<0>(tuple<Member> Owner)` does.
— **language**: what `std::forward<decltype(Owner)>(o).m` does.

2.1 The common parts

All the models agree on the following table:

n	Owner	Member	Forwarded
1			&&
2	&		&
3	&&		&&
4	const		const&&
5	const&		const&
6	const&&		const&&
7		const	const&&
8	&	const	const&
9	&&	const	const&&
10	const	const	const&&
11	const&	const	const&
12	const&&	const	const&&
13	&	&	&
14	&	&&	&
15	&	const&	const&
16	&	const&&	const&
17	const&	const&	const&
18	const&	const&&	const&

Commentary:

— For value-type members, we follow the forwarding category of the parent.
— If the parent is an lvalue, the result is an lvalue even for references.
— `const` is merged for these cases

2.2 The differing parts

The models differ in the following cases:

n	Owner	Member	'merge'	'tuple'	'language'
19		&	&&	&	&
20	&&	&	&&	&	&
21	const	&	const &&	&	&
22	const &	&	const &	&	&
23	const &&	&	const &&	&	&
24		&&	&&	&&	&
25	&&	&&	&&	&&	&
26	const	&&	const &&	&&	&
27	const &	&&	const &	&	&
28	const &&	&&	const &&	&&	&
29		const &	const &&	const &	const &

n	Owner	Member	'merge'	'tuple'	'language'
30	&&	const &	const &&	const &	const &
31	const	const &	const &&	const &	const &
32	const &&	const &	const &&	const &	const &
33		const &&	const &&	const &&	const &
34	&&	const &&	const &&	const &&	const &
35	const	const &&	const &&	const &&	const &
36	const &&	const &&	const &&	const &&	const &

Commentary:

— **language** is obviously wrong on all cases where both are rvalues - those should be rvalues. In addition, it requires both Owner and Member types to be explicit template parameters.
— **tuple**: collapses the value category of Owner and Member, inherits `const` from member. Plausible, but has problems with use-cases, and needs both Owner and Member types to be explicit template parameters.
— **merge**: merges the `const` from Owner and Member, overrides value category from Owner. Needs only Owner to be an explicit template parameter.

3 Use cases

In order to decide between the three models, let's look at use-cases.

3.0.1 A lambda that forwards its capture

This was the very first use-case for *deducing this*: a callback lambda that can be used in either "retry" (lvalue) or "try or fail" (rvalue, use-once) algorithms optimal efficiency.

With the *merge* model:

```
std::string message = get_message();
auto callback = [m=std::move(message), &scheduler](this auto &&self) -> bool {
  return scheduler.submit(std::forward<decltype(self)>(m)); // success-fail
};
callback(); // retry(callback)
std::move(callback)(); // try-or-fail(rvalue)
```

Or, with the **tuple** or **language** models:

```
std::string message = get_message();
auto callback = [m=std::move(message), &scheduler](this auto &&self) -> bool {
  return scheduler.submit(std::forward<decltype(self), decltype(m)>(m)); // success-fail
};
callback(); // retry(callback)
std::move(callback)(); // try-or-fail(rvalue)
```

3.0.2 Returning "far" owned state

```
struct S {
  std::unique_ptr<std::string> m;
  auto get(this auto&& self) -> std::string {
    if (m) {
      return std::forward<decltype(self), decltype(*m)>(*m);
    }
    return "";
```

```
  }
};
```

3.1 Interface

This results in the following interface:

```
template <typename T, typename U>
concept _similar = std::is_same_v<std::remove_cvref_t<T>,
                                  std::remove_cvref_t<U>>;

template <typename T, typename M, _similar<M> U>
auto forward(U &&x) noexcept -> _forward_like_t<T, M, U&&> {
  return static_cast<_forward_like_t<T, M, U>>(x);
}
```

3.2 Usage examples

3.3 As-Language (Not Chosen)

(see appendix A for the code listing)

with the usage looking like `std::forward_like<decltype(o), decltype(o.m)>(o.m)`, or `std::forward_like<Self, declty` for lambdas.

In lambdas, we actually get into a further problem that is not really solvable:

```
int x;
[&x](this auto&& self) { forward_like<decltype(self), decltype(x)>(x); /* int&& */ }
[&y=x](this auto&& self) { forward_like<decltype(self), decltype(y)>(y); /* int& */ }
```

In either case, the lambda does not own `x`, but in the common reference-capture case, *it would move*! This is unacceptable.

3.4 As-Tuple (chosen)

This facility chooses to follow the model of `std::tuple`.

4 Open Questions

Is LEWG is happy with the name `forward_like`?

Some alternative names: `forward_member`, (feel free to suggest more).

5 Proposal

Add the `forward_like` function template to the `utility` header.

```
template <typename T, typename U>
auto forward_like(U&& x) noexcept -> decltype(auto) {
    return static_cast</* see below */>(x);
}
```

6 Thank-yous and Acknowledgements

— Sarah from the #include discord for pointing out `std::tuple`'s `get` has a better view on how to treat reference members than the language does, thus saving the facility from being a mess that duplicates the language.

7 References

[P0847R7] Barry Revzin, Gašper Ažman, Sy Brand, Ben Deane. 2021-07-14. Deducing this.
https://wg21.link/p0847r7