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 spelled as `forward<decltype(o)>(o).m` in C++20.

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

Therefore, `forward_like<decltype(o), decltype(o.m)>(o.m)` and `forward<decltype(o)>(o).m` *should* result in identical forwarding behavior in all cases where `o.m` is a valid expression. `std::tuple`'s `get`, however, disagrees with this model for rvalue-reference members, as the two yield different results.

We therefore have two models to select from:

- language (`std::move(s).rvalue` results in an lvalue)
- `std::tuple` (`std::get<0>(std::move(t))` results in an rvalue, unless element 0 is an lvalue reference type).

Both of these were implemented.

Notice that we require *two explicit template parameters** to be provided. This means that `forward_like` doesn't conflict with the current interface for `std::forward`; we can therefore choose to spell the feature `std::forward<decltype(o), decltype(m)>(m)` (see discussion in *naming*).

2.1 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);
}
```

2.2 Usage examples

2.2.1 Lambda that returns its capture

```
auto l = [x](this auto &&self) -> decltype(auto) {
  return std::forward<decltype(self), decltype(x)>(x);
};
sink(l()); // sink(lvalue)
sink(std::move(l)()); // sink(rvalue)
```

2.2.2 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 "";
  }
};
```

2.3 Corner cases

2.4 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.

2.5 As-Tuple (chosen)

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

3 Open Questions

Is LEWG is happy with the name `forward_like`?

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

4 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);
}
```

5 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.

6 References

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