std::forward for members (forward_like)

Document #: D2445R0
Date: 2021-10-22
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 &&`	`&&`	`&`

n	Owner	Member	'merge'	'tuple'	'language'
27	`const &`	`&&`	`const &`	`&`	`&`
28	`const &&`	`&&`	`const &&`	`&&`	`&`
29		`const &`	`const &&`	`const &`	`const &`
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 in all cases (25, 28, 34, 36) 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, uses the value category of Owner. Needs only Owner to be an explicit template parameter.

3 Interface

In the **merge** model, the interface is:

```
template <typename T>
[[nodiscard]] constexpr
auto forward_like(auto&& x) noexcept -> __forward_like_t<T, decltype(x)> {
  return static_cast<__forward_like_t<T, decltype(x)>>(x);
}
```

In the **tuple** and **language** models, we need both to be explicit:

```
template <typename T, typename M>
[[nodiscard]] constexpr
auto forward_like(__similar<M> auto&& x) noexcept -> __forward_like_t<T, M, decltype(x)> {
  return static_cast<__forward_like_t<T, decltype(x)>>(x);
}
```

(`__similar<T, U>` is a concept that is satisfied by the two types if they are equal up to cv-ref qualifiers.)

However, because we need two explicit template parameters, the definition is compatible with calling it just `forward`, so we could use

```
std::forward<decltype(o), decltype(m)>(m)
```

instead of the longer `forward_like<decltype(o), decltype(m)>(m)` in these cases. This orthogonalizes the interface, which eases teaching. *If forwarding members, just supply both!*

The *language* and *tuple* models have bigger problems with the use cases, however, so this is just silver lining on a very dark cloud.

4 Use cases

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

4.1 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 with optimal efficiency.

With the *merge* model:

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

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

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

Note that *tuple* and *language* models have *significant problems* when applied to reference captures - see the section on that below.

4.2 Returning "far" owned state

This is a family of cases where we are forwarding a member "owned" by the Owner, but perhaps not directly contained by it.

With the **merge** model:

```
struct fwd {
  std::unique_ptr<std::string> ptr;
  std::optional<std::string> opt;
  std::deque<std::string> container;

  auto get_ptr(this auto&& self) -> std::string {
    if (ptr) { return std::forward_like<decltype(self)>(*ptr); }
    return "";
  }

  auto get_opt(this auto&& self) -> std::string {
    if (opt) { return std::forward_like<decltype(self)>(*m); }
    return "";
  }

  auto operator[](this auto&& self, size_t i) -> std::string {
    return std::forward_like<decltype(self)>(container[i]);
  }
};
```

and so on.

- The **language** and **tuple** models fail here - we need an alternative way to cast the far state into an rvalue (they both leave lvalue arguments as lvalues).
- In the `optional` case, we are lucky, and notice `optional` provides an rvalue accessor, which means we could spell the line as `*std::forward<decltype(self)>(self).opt`.
- However, `deque` does not provide an rvalue subscript operator (though it *could*);

— but `unique_ptr`'s `operator*()` *will never* provide the appropriate cast, as pointers have shallow semantics.

***merge* is the only model that satisfies this use case.**

4.3 Forwarding reference captures

There is another significant gotcha with the language and tuple models.

In lambdas with reference captures, find an unsolvable problem: [&] and [=] captures do not produce a distinguishing `decltype`. (notice lines (a) and (c) are the same!)

```
int x;
int z;
[&x, &y=x, z](this auto&& self) {                        /* move-invoked */
  forward_like<decltype(self), decltype(x)>(x); /* a: int&&            */
  forward_like<decltype(self), decltype(y)>(y); /* b: int& (!)         */
  forward_like<decltype(self), decltype(z)>(z); /* c: int&&            */
  forward_like<decltype(self), decltype(y)>(x); /* d: int& (and typo!) */
}();
```

The inconsistency here is dangerous.

- With the **language** and **tuple** models, we get inconsistent behavior between (a) and (b), which is extremely surprising, especially if one considers [&]-style captures.
- We also get *consistent* behavior between lines (a) and (c), which is a surprise in this case.
- (d) also exposes the brittle nature of typos with this model; we must reference the parameter twice so we run into problems with typos. This is impossible with the *merge* model, which is orthogonalized.
- With the **merge** model, we get consistent behavior - rvalue if invoked as an rvalue, lvalue if invoked as lvalue. Simple, predictable, obvious.

5 Open Questions

Is LEWG is happy with the name `forward_like`?

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

6 Proposal

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

```
template <typename T>
[[nodiscard]] constexpr
auto forward_like(auto&& x) noexcept -> __forward_like_t<T, decltype(x)> {
    return static_cast<__forward_like_t<T, decltype(x)>>(x);
}
```

where `__forward_like_t<T, U>` is a metafunction defined with the *merge* model table; or, more succinctly:

```
template <typename T, typename U>
using __override_ref_t = std::conditional_t<std::is_rvalue_reference_v<T>,
                                            std::remove_reference_t<U> &&, U &>;

template <typename T, typename U>
using __copy_const_t =
    std::conditional_t<std::is_const_v<std::remove_reference_t<T>>,
                       U const, U>;
```

```
template <typename T, typename U>
using __forward_like_t = __override_ref_t<
    T &&,
    __copy_const_t<T, std::remove_reference_t<U>>>;
```

7 Proposed Wording

Notes on wording: should we endeavor to define the `U` parameter as not-explicitly-specifiable by the user, as above, or do it old-style as now?

Add the following function to the `<utility>` header:

Synopsis:

```
template<class T, class U>
[[nodiscard]] constexpr auto forward_like(U&& x) noexcept -> /* see below */;
```

Let `OVERRIDE_REF(From, To)` denote the type `std::remove_reference_t<To> &&` if `From` is an rvalue reference, and `To &` otherwise.

Let `COPY_CONST(From, To)` denote the type `const To` if `std::remove_reference_t<From>` is `const`-qualified, and `To` otherwise.

Let `V` be the type denoted by `OVERRIDE_REF(O&&, COPY_CONST(O, std::remove_reference_t<decltype(m)>))`.

Returns: `static_cast<V>(x)`.

Remarks: The return type is `V`.

8 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.
- *Yunlan Tang*, who did some of the research for an early version of this paper.
- *Barry Revzin*, *Sy Brand* and *Ben Deane*, my dear co-authors of [P0847R7], without whom this paper would be irrelevant.
- *Vittorio Romeo*, who tried writing this paper first a few years ago.
- *Jens Maurer*, who wrote the initial wording, and *Corentin Jabot* also writing the wording. The current is a merge between both.
- *Tomasz Kamiński*, for pointing out typos.

9 Appendix: code listing for implementation and tables

```
#include <type_traits>
#include <utility>
#include <tuple>
#include <memory>
#include <string>

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

```cpp
template <typename T, typename U>
using _copy_ref_t = std::conditional_t<
    std::is_rvalue_reference_v<T>, U &&,
    std::conditional_t<std::is_lvalue_reference_v<T>, U &, U>>;

template <typename T, typename U>
using _override_ref_t = std::conditional_t<std::is_rvalue_reference_v<T>,
                                           std::remove_reference_t<U> &&, U &>;

template <typename T, typename U>
using _copy_const_t =
    std::conditional_t<std::is_const_v<std::remove_reference_t<T>>,
                       _copy_ref_t<U, std::remove_reference_t<U> const>, U>;

template <typename T>
constexpr bool _is_reference_v =
    std::is_lvalue_reference_v<T> || std::is_rvalue_reference_v<T>;

template <typename T, typename U>
using _copy_cvref_t = _copy_ref_t<T &&, _copy_const_t<T, U>>;

// test utilities

#define FWD(...) std::forward<decltype((__VA_ARGS__))>(__VA_ARGS__)

template <typename Expected, typename Actual> constexpr void is_same() {
  static_assert(std::is_same_v<Expected, Actual>);
}

namespace ftpl {
using std::forward;

template <typename T, typename U>
using _fwd_like_tuple_t =
    std::conditional_t<_is_reference_v<U>, _copy_ref_t<T, U>,
                       _copy_cvref_t<T, U>>;

// implementation
template <typename T, typename M, _similar<M> U>
auto forward_like_tuple(U &&x) noexcept -> decltype(auto) {
  return static_cast<_fwd_like_tuple_t<T, M>>(x);
}

template <typename T, typename M, _similar<M> U>
auto forward(U &&x) noexcept -> decltype(auto) {
  return forward_like_tuple<T, M>(static_cast<U &&>(x));
}
} // namespace ftpl

namespace flang {
using std::forward;
template <typename T, typename U>
using _fwd_like_lang_t =
    std::conditional_t<_is_reference_v<U>, U &,
```

```
                       _copy_ref_t<T, _copy_const_t<T, U>> &&>;
template <typename T, typename M, _similar<M> U>
auto forward(U &&x) noexcept -> decltype(auto) {
  return static_cast<_fwd_like_lang_t<T, _copy_const_t<U, M>>>(x);
}
} // namespace flang

namespace fmrg {
template <typename T, typename U>
using _copy_const_t =
    std::conditional_t<std::is_const_v<std::remove_reference_t<T>>, U const, U>;

template <typename T, typename U>
using _fwd_like_merge_t =
    _override_ref_t<T &&, _copy_const_t<T, std::remove_reference_t<U>>>;

template <typename T, typename U>
auto forward_like(U &&x) noexcept -> decltype(auto) {
  return static_cast<_fwd_like_merge_t<T, U>>(x);
}
} // namespace fmrg

struct probe {};

template <typename M> struct S {
  M m;
  using value_type = M;
};

template <typename T, typename Merge, typename Tuple, typename Lang>
void test() {
  using value_type = typename std::remove_cvref_t<T>::value_type;

  using mrg = decltype(fmrg::forward_like<T>(std::declval<value_type>()));
  using tpl_model = decltype(std::get<0>(
      std::declval<_copy_cvref_t<T, std::tuple<value_type>>>()));
  using tpl =
      decltype(ftpl::forward<T, value_type>(std::declval<value_type>()));
  using lng_model = decltype((std::forward<T>(std::declval<T>()).m));
  using lng =
      decltype(flang::forward<T, value_type>(std::declval<value_type>()));

  is_same<Merge, mrg>();
  is_same<Tuple, tpl>();
  is_same<Lang, lng>();
  // sanity checks
  is_same<Tuple, tpl_model>();
  is_same<Lang, lng_model>();
}

void test() {
  using p = probe;
  // clang-format off
  //   TEST TYPE             ,'merge'    ,'tuple'    ,'language'
```

```cpp
  test<S<p         >         , p &&      , p &&      , p &&      >();
  test<S<p         > &       , p &       , p &       , p &       >();
  test<S<p         > &&      , p &&      , p &&      , p &&      >();
  test<S<p         > const   , p const &&, p const &&, p const &&>();
  test<S<p         > const & , p const & , p const & , p const & >();
  test<S<p         > const &&, p const &&, p const &&, p const &&>();
  test<S<p const   >         , p const &&, p const &&, p const &&>();
  test<S<p const   > &       , p const & , p const & , p const & >();
  test<S<p const   > &&      , p const &&, p const &&, p const &&>();
  test<S<p const   > const   , p const &&, p const &&, p const &&>();
  test<S<p const   > const & , p const & , p const & , p const & >();
  test<S<p const   > const &&, p const &&, p const &&, p const &&>();
  test<S<p &       > &       , p &       , p &       , p &       >();
  test<S<p &&      > &       , p &       , p &       , p &       >();
  test<S<p const & > &       , p const & , p const & , p const & >();
  test<S<p const &&> &       , p const & , p const & , p const & >();
  test<S<p const & > const & , p const & , p const & , p const & >();
  test<S<p const &&> const & , p const & , p const & , p const & >();

  test<S<p &       >         , p &&      , p &       , p &       >();
  test<S<p &       > &&      , p &&      , p &       , p &       >();
  test<S<p &       > const   , p const &&, p &       , p &       >();
  test<S<p &       > const & , p const & , p &       , p &       >();
  test<S<p &       > const &&, p const &&, p &       , p &       >();
  test<S<p &&      >         , p &&      , p &&      , p &       >();
  test<S<p &&      > &&      , p &&      , p &&      , p &       >();
  test<S<p &&      > const   , p const &&, p &&      , p &       >();
  test<S<p &&      > const & , p const & , p &       , p &       >();
  test<S<p &&      > const &&, p const &&, p &&      , p &       >();
  test<S<p const & >         , p const &&, p const & , p const & >();
  test<S<p const & > &&      , p const &&, p const & , p const & >();
  test<S<p const & > const   , p const &&, p const & , p const & >();
  test<S<p const & > const &&, p const &&, p const & , p const & >();
  test<S<p const &&>         , p const &&, p const &&, p const & >();
  test<S<p const &&> &&      , p const &&, p const &&, p const & >();
  test<S<p const &&> const   , p const &&, p const &&, p const & >();
  test<S<p const &&> const &&, p const &&, p const &&, p const & >();
  // clang-format on
}

void test_lambdas() {
  probe x;
  probe z;
  auto l = [x, &y = x, z](auto &&self) mutable {
    // correct, this is what we *meant*, consistently
    // If we didn't mean to forward the capture, we wouldn't have used
    // forward_like.
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(fmrg::forward_like<decltype(self)>(y))>();
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(fmrg::forward_like<decltype(self)>(x))>();
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(fmrg::forward_like<decltype(self)>(z))>();
```

```
    // x and y behave differently with the tuple model (problem)
    is_same<probe &, decltype(ftpl::forward<decltype(self), decltype(y)>(y))>();
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(ftpl::forward<decltype(self), decltype(x)>(x))>();
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(ftpl::forward<decltype(self), decltype(z)>(z))>();

    // x and y behave differently with the language model (problem)
    is_same<probe &,
            decltype(flang::forward<decltype(self), decltype(y)>(y))>();
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(flang::forward<decltype(self), decltype(x)>(x))>();
    is_same<_override_ref_t<decltype(self), probe>,
            decltype(flang::forward<decltype(self), decltype(z)>(z))>();
  };
  l(l);             // lvalue-call emulation
  l(std::move(l)); // sortish like a this-auto-self with a && call operator
}

struct owns_far_string {
  std::unique_ptr<std::string> s;
};

void test_far_objects() {
  // problem is that *unique_ptr returns a reference
  owns_far_string fs;
  auto l = [](auto &&fs) {
    using mrg = decltype(fmrg::forward_like<decltype(fs)>(*fs.s));
    using tpl = decltype(ftpl::forward<decltype(fs), decltype(*fs.s)>(*fs.s));
    using lng = decltype(flang::forward<decltype(fs), decltype(*fs.s)>(*fs.s));

    // fit for purpose
    is_same<_override_ref_t<decltype(fs), std::string>, mrg>();
    // these are not fit for purpose
    is_same<std::string &, tpl>();
    is_same<std::string &, lng>();
  };
  l(fs);            // lvalue call
  l(std::move(fs)); // rvalue call - we want to move the string out
}

int main() {
  test();
  test_lambdas();
  test_far_objects();
}
```

10 References

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