Document number: P2196R0
Revises:
Date: 2020-07-06
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: LEWGI
Reply to: Vincent Reverdy
École Normale Supérieure, rue d'Ulm, Paris, France
vince.rev@gmail.com

A lifetime-extending forwarder

Abstract

We propose an addition to the `std::forward` mechanism under the form of a type capable of properly capturing an argument and its cvref-qualifiers at construction, extending its lifetime when necessary, and perfectly forwarding it when called. The functionality can be thought of as a generalization of the existing utility to the case for which the location of "capture" and the location of "restitution" are different.

Contents

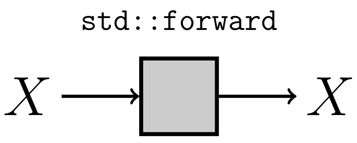
1 Proposal [proposal]

1.1 History [proposal.history]

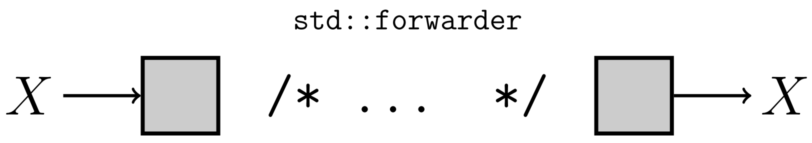
— P2196R0: this version, the original proposal submitted to LEWGI.

1.2 Introduction [proposal.intro]

This proposal aims at solving the following problem: how to correctly forward an entity when the cvref-qualification that needs to be remembered and the actual use of the entity do not happen at the same location? For a lot of users `std::forward` remains a magic tool that makes things work when one want to transfer an entity with its correct cvref-qualification. But being able to distinguish the place where the cvref-qualification need to be saved, from the place where the entity is used can bring another layer of flexibility. This can be useful, for example, to allow users to call the correct function when wrapping callables with cvref-qualified overloads. To solve this problem, this proposal introduces a `std::forwarder` class template whose functionality can be seen as the following: remembering the cvref-qualification of an entity at construction site, correctly extending the lifetime of temporary objects when necessary, and forwarding the entity when called. The idea can be summarized below:



and for the proposed `std::forwarder`:



To make it happen, the proposal also introduces two concepts with a wide range of use: `std::similar` mimicking the definition of *similar* types in the standard (see `[conv.qual]`), and `std::alike` to handle similarity in the context of cvref-qualified types and forwarding references. We believe these two concepts can have a wide range of use, one of them being to restrain the scope of constructors taking forwarding references as inputs. An illustration of the functionality provided by `std::forwarder` is given below:

Illustration of `std::forwarder` functionality

```
// Preamble
#include <iostream>

// An object with an overloaded member function
struct object {
    void test() const & {std::clog << "void test() const &\n";}
    void test() const && {std::clog << "void test() const &&\n";}
};

// A function taking a forwarding reference as an input
template <class T>
auto function(T&& t) {
    auto fwd = forwarder<T>(t);
    /* ... */
    return fwd;
}

// Main function
int main(int argc, char* argv[]) {
    object x;
    auto fwd0 = function(x);
    auto fwd1 = function(object{});
    /* ... */
    fwd0().test(); // void test() const &
    fwd1().test(); // void test() const &&
    return 0;
}
```

In practical applications, `std::forwarder` can be used as a class data member that captures the cvref-qualifiers of an argument passed to one of the class function members, and allow the class to reuse this argument in another function member with the correct cvref-qualification. This use case has been encountered by the author of this proposal when writing utilities to handle overload sets.

1.3 Impact on the standard [proposal.impact]

This proposal is a pure library extension. It does not require any changes in the core language and does not depend on any other library extensions. The proposed extensions are added to the `<type_traits>`, `<concepts>`, and `<utility>` headers.

1.4 Motivation [proposal.motivation]

The need for a lifetime-extending `forwarder` became clear when working on the implementation of a generic *perfect forwarding call wrapper* (see `[func.def]` and `[func.require]`), itself needed to implement an updated revision of P1772R1: Variadic overload sets and overload sequences. Since the functionality and the related concepts were meeting some of the criteria required by LEWGI to be relevant candidates for standardization (see below), the tools were extracted to build this independent proposal. This proposal therefore constitutes a dependency to build a generic `std::overload_set` and `std::overload_sequence`, which are proposed separately. However since the proposed `std::similar`, `std::alike`, and `std::forwarder` are more widely applicable, they have been isolated from the rest and are therefore proposed in this paper.

	`std::similar`	`std::alike`	`std::forwarder`
Widely applicable use	✓✓	✓✓✓	✓
Encapsulates nonportability			
Difficult to implement correctly	✓✓	✓	✓✓✓
Requires language support			

The three items are proposed together because the later, namely `std::forwarder`, depends on `std::alike`, which itself depends on `std::similar`. The `similar` concept just reflects in code something that is already perfectly defined in the standard: the notion of type similarity (see `[conv.qual]`). Since its implementation is believed to not be straightforward for developers who are not used to formal wording, providing it in the standard library is believed to be a nice addition. In practice, the `alike` concept will probably be used more often since it basically expresses the notion of "decaying to the same type" or more exactly "decaying to similar types", which is particularly useful when dealing with forwarding references.

Finally, the core of this proposal, `std::forwarder`, can be thought as an extension of `std::forward` when, as described in introduction, the location of the thing that needs to be forwarded and the location of where it needs to be forwarded to are different. A typical use case of the `forwarder`, as illustrated below, is to serve as a data member of a class, in which an argument needs to be perfectly captured in a function member, and used in a different one. In terms of lines of code, the implementation of the `forwarder` is not that complicated: however implementing it requires a good understanding of the forwarding mechanism in C++, which remains black magic for most users. Therefore, we see significant value in adding it to the standard as an extension of `std::forward`.

Example of `std::forwarder` application

```cpp
// Preamble
#include <utility>
#include <iostream>
#include <type_traits>

// A function object class
struct function_object {
    template <class... Args>
    void operator()(Args&&...) const & {
        std::clog << "void operator()(Args&&...) const &\n";
    }
    template <class... Args>
    void operator()(Args&&...) const && {
        std::clog << "void operator()(Args&&...) const &&\n";
    }
};

// A function object wrapper
template <class F>
struct function_wrapper {
    template <alike<F> G>
    constexpr function_wrapper(G&& g)
    : fwd(std::forward<G>(g)) {
    }
    template <class... Args>
    void operator()(Args&&... args) {
        fwd()(std::forward<Args>(args)...);
    }
    forwarder<F> fwd;
};

// Its deduction guide
template <class F>
function_wrapper(F&&) -> function_wrapper<F>;

// Main function
int main(int argc, char* argv[]) {
    function_object function0;
    function_wrapper wrapper0(function0);
    function_wrapper wrapper1(function_object{});
    wrapper0(1, 2, 3); // void operator()(Args&&...)  const &
    wrapper1(4, 5, 6); // void operator()(Args&&...)  const &&
    return 0;
}
```

1.5 Design choices [proposal.design]

The space of possible designs for this functionality is rather small.

First, the notion of similar types is already fully specified in the standard: the proposed `std::similar` concept just materializes it.

Second, the functionality provided by the `std::alike` concept is, to our opinion, very much needed in the context of generic function templates taking forwarding references as inputs. Instead of making `std::alike` depend on `std::same_as`, we believe that making it depend on `std::similar` leads to a more accurate definition of what users actually mean in their code. From a purely subjective standpoint, we find expressions like `template <alike<T> U>` very readable and explicit in code.

Third, regarding `std::forwarder`. The name fully describes what it is: it is a type whose role is to forward. The capture of the parameter to be forwarded is done at construction, and we found it natural to use the function call `operator()` as the way to execute the forwarder and make it forward. We chose to provide a deduction guide because we believe users of this functionality will have a good grasp of the forwarding mechanism. In the context of forwarding references, we find it easy to explain to users that `std::forwarder<Arg>(arg)` can be thought of as a shorthand for `std::forwarder(std::forward<Arg>(arg))`. Finally, in this version, we decided to ban reassignment and direct access to the underlying entity via a getter because we think it would only obfuscate the functionality. It could also lead more easily to unwanted or confusing results. In its current form the functionality remains very simple to explain: `std::forwarder` captures the cvref-qualification of an entity at construction, properly extends the lifetime of temporary objects when needed, and actually forwards its contents when called through its function call operator. As for `std::forward`, it wraps a not-so-easy-to-understand functionality of the C++ language, and "does the right thing". In its current form, only unqualified and `const`-qualified versions of `operator()` are provided: whether their `volatile`-qualified counterpart would be useful is left as an open question. Any reader with a use case in mind for such `volatile`-qualified `std::forwarder::operator()` is encouraged to contact the author of this proposal.

1.6 Technical specification [proposal.specs]

See the proposed wording and an illustrative implementation at the end of this paper.

1.7 Open questions [proposal.questions]

- Bikeshedding: `similar` vs `similar_to` vs *alternative names*
- Bikeshedding: `alike` vs *alternative names*
- Bikeshedding: `forwarder` vs *alternative names*
- Should both the `is_similar` type trait and the `similar` concept be exposed, or just the concept?
- Should both the `is_alike` type trait and the `alike` concept be exposed, or just the concept?
- Should `volatile` and `const volatile`-qualified `forwarder::operator()` members be provided?
- Should `forwarder` be copy and move assignable?

1.8 Acknowledgements [proposal.acks]

This work has been made possible thanks to *PSL Research University*.

1.9 References [proposal.bibliography]

— A forwarder implementation, Vincent Reverdy, *Github* (July 2020)

— N4860: Draft International Standard - Programming Languages - C++, Richard Smith, *ISO / IEC – JTC1 / SC22 / WG21* (April 2020)

2 Wording [wording]

2.1 Concepts library [concepts]

2.1.1 Header `<concepts>` synopsis [concepts.syn]

1 Add the following after the concept `same_as` and before the concept `derived_from`:

```
// 2.1.2, concept similar
template <class T, class U>
concept similar = see below;

// 2.1.3, concept alike
template <class T, class U>
concept alike = see below;
```

2.1.2 Concept `similar` [concept.similar]

```
template<class T, class U>
concept similar-impl = is_similar_v<T, U>;          // exposition only

template<class T, class U>
concept similar = similar-impl<T, U> && similar-impl<U, T>;
```

1 [*Note*: `similar<T, U>` subsumes `similar<U, T>` and vice versa. — *end note*]

2.1.3 Concept `alike` [concept.alike]

```
template<class T, class U>
concept alike-impl = similar<decay_t<T>, decay_t<U>>;          // exposition only

template<class T, class U>
concept alike = alike-impl<T, U> && alike-impl<U, T>;
```

1 [*Note*: `alike<T, U>` subsumes `alike<U, T>` and vice versa. — *end note*]

2.2 General utilities library [utilities]

2.2.1 Utility components [utility]

2.2.1.1 Header `<utility>` synopsis [utility.syn]

1 Add the following at the end of the code section `// forward/move`:

```
template <class T>
class forwarder;
```

2.2.1.2 Forward/move helpers [forward]

2.2.1.2.1 Class template `forwarder` [forwarder]

```
namespace std {
template <class T>
class forwarder {
public:
    using type = T&&;

    // Constructors
    template <alike<T> U>
    constexpr forwarder(U&& arg) noexcept(
        std::is_nothrow_constructible_v<T, U&&>
    );
    constexpr forwarder(const forwarder<T>& other) noexcept(
        std::is_nothrow_constructible_v<T, T>
    ) = default;
    constexpr forwarder(forwarder<T>&& other) noexcept(
        std::is_nothrow_constructible_v<T, T&&>
    );
    template <alike<T> U>
    constexpr forwarder(const forwarder<U>& other) noexcept(
        std::is_nothrow_constructible_v<T, U>
    );
    template <alike<T> U>
    constexpr forwarder(forwarder<U>&& other) noexcept(
        std::is_nothrow_constructible_v<T, U&&>
    );

    // Accessors
    constexpr T&& operator()() noexcept;
    constexpr T&& operator()() const noexcept;

    // Implementation details
    private:
    T entity; // exposition only
};

// Deduction guide
template <class T>
requires see below
forwarder(T&&) -> forwarder<T>;
```

2.2.1.2.1.1 Constructors [forwarder.con]

```
template <alike<T> U>
constexpr forwarder(U&& arg) noexcept(
    std::is_nothrow_constructible_v<T, U&&>
);
```

1 *Effects:* Constructs entity from `std::forward<U>(arg)`.

```
constexpr forwarder(const forwarder<T>& other) noexcept(
    std::is_nothrow_constructible_v<T, T>
) = default;
```

2 *Effects:* Copies `entity`.

```
constexpr forwarder(forwarder<T>&& other) noexcept(
    std::is_nothrow_constructible_v<T, T&&>
);
```

3 *Effects:* Constructs entity from `std::forward<T>(other.entity)`.

```
template <alike<T> U>
constexpr forwarder(const forwarder<U>& other) noexcept(
    std::is_nothrow_constructible_v<T, U>
);
```

4 *Effects:* Constructs entity from `other.entity`.

```
template <alike<T> U>
constexpr forwarder(forwarder<U>&& other) noexcept(
    std::is_nothrow_constructible_v<T, U&&>
);
```

5 *Effects:* Constructs entity from `std::forward<U>(other.entity)`.

2.2.1.2.1.2 Accessors [forwarder.access]

```
constexpr T&& operator()() noexcept;
constexpr T&& operator()() const noexcept;
```

1 *Returns:* `std::forward<T>(entity)`.

2.2.1.2.1.3 Deduction guide [forwarder.deduction]

```
template <class T>
requires see below
forwarder(T&&) -> forwarder<T>;
```

1 *Constraints:* T is not an instance of `forwarder`.

2.2.2 Metaprogramming and type traits [meta]

2.2.2.1 Header <type_traits> synopsis [meta.type.synop]

1 Add the following in code section `// type relations` right after the line corresponding to the declaration of `is_same`:

```
template <class T, class U> struct is_similar;
template <class T, class U> struct is_alike;
```

2 Add the following in code section `// type relations` right after the line corresponding to the definition of `is_same_v`:

```
template <class T, class U>
inline constexpr bool is_similar_v = is_similar<T, U>::value;
template <class T, class U>
inline constexpr bool is_alike_v = is_alike<T, U>::value;
```

2.2.2.2 Relationships between types [meta.rel]

1 Add the following to the table "Type relationship predicates `[tab:meta.rel]`", right after the row corresponding to `is_same`:

Table 1: Type relationship predicates [tab:meta.rel]

Template	Condition	Comments
`template<class T, class U> struct is_similar;`	`T` and `U` name similar types.	See `[conv.qual]` for the definition of similarity.
`template<class T, class U> struct is_alike;`	`is_similar_v<decay_t<T>, decay_t<U>>`.	

3 Implementation [implementation]

A working implementation is provided here: https://github.com/vreverdy/forwarder. It is detailed below:

3.1 Similar concept [implementation.similar]

std::similar concept implementation

```
// ================================ SIMILAR ================================ //
// Checks if two types are similar:  cv-qualified types
template <class T, class U>
struct is_similar
: conditional_t<
    is_const_v<T> || is_volatile_v<T> ||
    is_const_v<U> || is_volatile_v<U>,
    is_similar<remove_cv_t<T>, remove_cv_t<U>>,
    is_same<T, U>
> {};

// Checks if two types are similar:  pointer types
template <class T, class U>
struct is_similar<T*, U*>: is_similar<T, U> {};

// Checks if two types are similar:  pointer to member types
template <class T, class U, class C>
struct is_similar<T C::*, U C::*>: is_similar<T, U> {};

// Checks if two types are similar:  arrays
template <class T, class U, size_t N>
struct is_similar<T[N], U[N]>: is_similar<T, U> {};

// Checks if two types are similar:  unknown bound arrays
template <class T, class U>
struct is_similar<T[], U[]>: is_similar<T, U> {};

// Checks if two types are similar:  left-hand side unknown bound array
template <class T, class U, size_t N>
struct is_similar<T[], U[N]>: is_similar<T, U> {};

// Checks if two types are similar:  right-hand side unknown bound array
template <class T, class U, size_t N>
struct is_similar<T[N], U[]>: is_similar<T, U> {};

// Variable template
template <class T, class U>
inline constexpr bool is_similar_v = is_similar<T, U>::value;

// Implementation details:  concept
template <class T, class U>
concept _similar = is_similar_v<T, U>;

// Concept
template <class T, class U>
concept similar = _similar<T, U> && _similar<U, T>;
// ========================================================================= //
```

3.2 Alike concept [implementation.alike]

std::alike concept implementation

```
// ================================= ALIKE ================================= //
// Checks if two types are alike:  their decayed types are similar
template <class T, class U>
struct is_alike: is_similar<decay_t<T>, decay_t<U>> {};

// Variable template
template <class T, class U>
inline constexpr bool is_alike_v = is_alike<T, U>::value;

// Implementation details:  concept
template <class T, class U>
concept _alike = similar<decay_t<T>, decay_t<U>>;

// Concept
template <class T, class U>
concept alike = _alike<T, U> && _alike<U, T>;
// ======================================================================== //
```

3.3 Forwarder class template [implementation.forwarder]

std::forwarder class template implementation

```
// ================================ FORWARDER =============================== //
// A class that holds an entity and forwards it
template <class T>
class forwarder
{
    // Friendship
    template <class U>
    friend class forwarder;

    // Types
    public:
    using type = T&&;

    // Lifecycle
    public:
    template <alike<T> U>
    constexpr forwarder(U&& arg) noexcept(
        is_nothrow_constructible_v<T, U&&>
    ): _arg(forward<U>(arg)) {
    }
    constexpr forwarder(const forwarder<T>& other) noexcept(
        is_nothrow_constructible_v<T, T>
    ) = default;
    constexpr forwarder(forwarder<T>&& other) noexcept(
        is_nothrow_constructible_v<T, T&&>
    ): _arg(forward<T>(other._arg)) {
    }
    template <alike<T> U>
    constexpr forwarder(const forwarder<U>& other) noexcept(
        is_nothrow_constructible_v<T, U>
    ): _arg(other._arg) {
    }
    template <alike<T> U>
    constexpr forwarder(forwarder<U>&& other) noexcept(
        is_nothrow_constructible_v<T, U&&>
    ): _arg(forward<U>(other._arg)) {
    }

    // Forwarding
    public:
    constexpr T&& operator()() noexcept {
        return forward<T>(_arg);
    }
    constexpr T&& operator()() const noexcept {
        return forward<T>(_arg);
    }

    // Implementation details:  data members
    private:
    T _arg;
};

// Implementation details:  checks if a type is a forwarder:  default
template <class T>
struct _is_forwarder: false_type {};

// Implementation details:  checks if a type is a forwarder:  specialization
template <class T>
struct _is_forwarder<forwarder<T>>: true_type {};

// // Implementation details:  variable template
template <class T>
inline constexpr bool _is_forwarder_v = _is_forwarder<T>::value;

// Deduction guide
template <class T>
requires !_is_forwarder_v<remove_cvref_t<T>>
forwarder(T&&) -> forwarder<T>;
// ========================================================================== //
```