```
auto b = std::begin(std::forward<decltype(r)>(r));
auto e = std::end  (std::forward<decltype(r)>(r));  // BAD IDEA:
                                                    // r might be moved from.
```

Forwarding `r` only in the initialization of `e` might avoid issues caused by moving an object twice but might result in inconsistent behavior with `b`:

```
auto b = std::begin(r);
auto e = std::end(std::forward<decltype(r)>(r));  // BAD IDEA: e might have
                                                  // a different type than b.
```

The `std::forward` utility

The final piece of the forwarding reference infrastructure is the `std::forward` utility function. Since the expression naming a forwarding reference `x` is always an *lvalue* due to its reachability by either name or address and since our intention is to move `x` in case it was originally an *rvalue*, we need a conditional *move* operation that will move `x` only in that case and otherwise let `x` pass through as an *lvalue*.

The Standard Library provides two overloads of the `std::forward` function in the `<utility>` header:

```
namespace std {
template <class T> T&& forward(typename remove_reference<T>::type&  t) noexcept;
template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
}
```

Note that, to avoid ambiguity, the second overload will be deliberately removed from the overload set if `T` is an *lvalue* reference type.

Recall that the type `T` associated with a forwarding reference is deduced as a reference type if given an *lvalue* reference and as a nonreference type otherwise. So for a forwarding reference `forRef` of type `T&&`, we have two cases.

1. An *lvalue* of type `U` was used for initializing `forRef`, so `T` is `U&`; thus, the first overload of `forward` will be selected and will be of the form `U& forward(U& u) noexcept`, thus just returning the original *lvalue* reference. Notice the effect of reference collapsing in the return type: `(U&)&&` becomes simply `U&`.
2. An *rvalue* of type `U` was used for initializing `forRef`, so `T` is `U`, and the second overload of `forward` will be selected and will be of the form `U&& forward(U&& u) noexcept`, essentially equivalent to `std::move`.

Note that, in the body of a function template accepting a forwarding reference `T&&` named `x`, `std::forward<T>(x)` could be replaced with `static_cast<T&&>(x)` to achieve the same effect. Due to reference collapsing rules, `T&&` will resolve to `T&` whenever the original value category of `x` was an *lvalue* and to `T&&` otherwise, thus achieving the *conditional move* behavior elucidated in *Description* on page 377. Using `std::forward` over `static_cast`, however, expresses the programmer's intent explicitly.