deduce

Document Number: P4338R0
Date: 2026-08-03
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG, SG18

Abstract

This paper proposes a utility for easily writing deduction guides which are `emplace_from`-aware [1].

Background

`emplace_from` is a proposed standard library class template which enables emplace functionality (e.g. `std::vector::emplace_back`, `std::optional::emplace`, et cetera) to be used to construct immovable types returned by factory functions (e.g. `std::execution::connect`).

The introduction of a proxy type, however, interferes with CTAD. Consider:

```
std::optional o(5);
```

`std::optional<int>` is deduced, which is what is expected. Now consider:

```
std::optional o(std::emplace_from{[&] {
  return std::execution::connect(sndr, rcvr);
}});
```

`std::optional` will be deduced as having an element type of `std::emplace_from<...>` rather than the intended operation state type.

An earlier proposal of functionality very similar to `emplace_from` proposed a core language change in an attempt to address the above [2].

Discussion

CTAD can not only avail itself of rules deduced by examining the class whose template arguments are being deduced, but also of rules provided by users through deduction guides.

Therefore to ameliorate the above the author of a type can provide a deduction guide:

```
template<typename T>
optional(emplace_from<T>) -> optional<call-result-t<T>>;
```

However this potentially increases the number of deduction guides which must be provided. Instead this paper proposes a template type alias, std::deduce_t<T>, which:

- Resolves to the type `emplace_from` converts to, if `T` is a possibly cv- and ref-qualified `emplace_from` specialization,
- `T` otherwise

Note that this discussion of deduction guides is not purely academic: Several types in the standard library would benefit from `emplace_from`-aware deduction guides. Therefore this paper also proposes adding them.

Wording

Note: The below assumes P4337 [1] has been applied.

[utility.syn]

```
namespace std {

  [...]

  struct cw-operators;                    // exposition only

  template<auto X>
    constexpr auto cw = constant_wrapper<X>{};

  // [utility.emplace_from], emplace construction support
  template<class Fun>
    struct emplace_from;

  // [utility.deduce], class template argument deduction support
  template<class T>
    struct deduce;
  template<class T>
    using deduce_t = deduce<T>::type;

  // [intseq], compile-time integer sequences
  template<class T, T...>
    struct integer_sequence;
  template<size_t... I>
```

```
    using index_sequence = integer_sequence<size_t, I...>;

  [...]

}
```

[utility.deduce]

Note: This is a new section.

```
template<class T>
  struct deduce {
    using type = see below;
  };
```

The member *typedef-name* `type` denotes *`call-result-t`*`<T>` if `remove_cvref_t<T>` denotes a specialization of `emplace_from` ([utility.emplace_from]), `T` otherwise.

[pairs.pair]

```
namespace std {

  [...]

  template<class T1, class T2>
    pair(T1&&, T2&&) -> pair<decay_t<deduce_t<T1>>, decay_t<deduce_t<T2>>>;
}
```

[tuple.tuple.general]

```
namespace std {

  [...]

  template<class... UTypes>
    tuple(UTypes&&...) -> tuple<decay_t<deduce_t<UTypes>>...>;
  template<class T1, class T2>
    tuple(pair<T1, T2>) -> tuple<T1, T2>;
  template<class Alloc, class... UTypes>
    tuple(allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;
  template<class Alloc, class T1, class T2>
    tuple(allocator_arg_t, Alloc, pair<T1, T2>) -> tuple<T1, T2>;
```

```
  template<class Alloc, class... UTypes>
    tuple(allocator_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>;
}
```

[optional.optional.general]

```
namespace std {

  [...]

  template<class T>
    optional(T&&) -> optional<decay_t<deduce_t<T>>>;
}
```

[expected.un.general]

```
namespace std {

  [...]

  template<class E> unexpected(E&&) -> unexpected<decay_t<deduce_t<E>>>;
}
```

[array.overview]

```
namespace std {

  [...]

  template<class T, class... U>
    array(T&&, U...) -> array<decay_t<deduce_t<T>>, 1 + sizeof...(U)>;
}
```

Implementation Experience

The author has deployed `elide` and `deduce_t` internally [3]. They are also provided by beman.emplace_from [4] (previously known as beman.elide).

References

[1] R. Leahy. emplace_from P4337R0

[2] T. Healy. std::elide P3288R3
[3] R. Leahy. An Adventure in Modern Library Design C++Now 2024
[4] https://github.com/bemanproject/emplace_from