`emplace_from`

Document Number: P4337R0
Date: 2026-08-03
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG, SG18

Abstract

This paper proposes a bona fide, user-facing version of the exposition-only *`emplace-from`* helper employed by `std::execution` [1].

Background

Enabled by variadic templates and forwarding references, C++11 brought emplace construction. Containers, wrappers, et cetera could provide variadic constructors or functions which forwarded arguments through to a constructor of some wrapped or contained type thereby creating an instance of that type without intervening temporaries (which would be copied or moved from).

C++17, with its reformulation of prvalues and guaranteed RVO changed the landscape. Emplace construction was no longer the most general way to defer responsibility for selecting the modality by which some wrapped or contained object ought to be constructed. Instead the most general way became accepting a function whose returned value, a prvalue, would be materialized directly into storage.

While C++11 not only brought the language features to enable emplace construction, but also library support therefor, C++17 only brought the language features. The standard library was not enriched with support for this new construction deferral modality.

Shortly after C++17 it was discovered that emplace construction could be used to inject a function which is invoked thereby wiring prvalue materialization into existing library functionality. This is accomplished through abuse of operator overloading, particularly the conversion operator [2].

A previous paper [3] proposed that the above-mentioned technique be given first class support in the standard library (as `std::elide`). That paper has been abandoned by its author.

C++26 however, does, in some sense, integrate the above-mentioned technique into its standard library. `std::execution` [1] uses that exact technique via the exposition-only class template *`emplace-from`*.

Discussion

The entity proposed by this paper, `std::emplace_from`, is widely-understood and -used under a variety of names:

- `elide` [3]
- `emplace_from` [1]
- `with_result_of_t` [2]

And is required to implement the standard library as written.

Moreover the fact that `std::execution`'s operation states are:

- Immovable, and
- Created by being returned from a function (`std::execution::connect`) via C++17 guaranteed RVO

Means that users will increasingly need the functionality of *`emplace-from`* (it is for this reason, in fact, that `std::execution` requires said functionality). We should promote *`emplace-from`* to `std::emplace_from` to save users having to implement it themselves.

Wording

[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;

  // [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.emplace_from]

Note: This is a new section.

`emplace_from` is used to emplace non-movable types into `tuple`, `optional`, `variant`, containers, et cetera.

```
template<class Fun>
concept emplace-from-concept =                          // exposition only
  !same_as<call-result-t<Fun>, void>;

template<class Fun>
struct emplace_from {
  Fun fun;                                              // exposition only

  template<class Self>
  using fun-t =                                         // exposition only
    decltype(std::forward_like<Self>(declval<Fun>()));

  template<class Self>
  using result-t = call-result-t<fun-t<Self>>;          // exposition only

  template<class Self>
  static constexpr bool enabled =                       // exposition only
    emplace-from-concept<fun-t<Self>>;

  template<class Self>
  static constexpr bool nothrow =                       // exposition only
    nothrow-callable<fun-t<Self>>;

  template<class Self>
    requires enabled<Self>
  constexpr result-t<Self> operator()(this Self&& self) noexcept(
    nothrow<Self>)
  {
    return std::forward_like<Self>(self).fun();
  }
```

```
  template<class Self>
    requires enabled<Self>
  constexpr operator result-t<Self>(this Self&& self) noexcept(
    nothrow<Self>)
  {
    return std::forward_like<Self>(self)();
  }

  emplace_from() = default;
  explicit constexpr emplace_from(Fun f) noexcept(
    is_nothrow_move_constructible_v<Fun>)
    : fun(std::move(f)) {}
};
```

[exec.snd.expos]

```
template<class Tag, class Env, class Default>
  constexpr decltype(auto) query-with-default(
    Tag, const Env& env, Default&& value) noexcept(see below);
```

Let e be the expression `Tag()(env)` if that expression is well-formed; otherwise, it is `static_cast<Default>(std::forward<Default>(value))`.

Returns: e.

Remarks: The expression in the `noexcept` clause is `noexcept(e)`.

~~template<callable Fun>~~
~~requires is_nothrow_move_constructible_v<Fun>~~
~~struct emplace-from {~~
~~Fun fun; // exposition only~~
~~using type = call-result-t<Fun>;~~

~~constexpr operator type() && noexcept(nothrow-callable<Fun>) {~~
~~return std::move(fun)();~~
~~}~~

~~constexpr type operator()() && noexcept(nothrow-callable<Fun>) {~~
~~return std::move(fun)();~~
~~}~~
~~};~~

~~[*Note 1: `emplace-from` is used to emplace non-movable types into `tuple`, `optional`, `variant`, and similar types. — end note*]~~

```
struct on-stop-request {
  inplace_stop_source& stop-src;        // exposition only
  void operator()() noexcept { stop-src.request_stop(); }
};
```

Instruction to the Editor

Replace all occurrences of *`emplace-from`* with `emplace_from`.

Implementation Experience

beman.emplace_from [4] (previously known as beman.elide) implements this paper.

References

[1] M. Dominiak et al. std::execution P2300R10
[2] A. O'Dwyer. Superconstructing super elider, round 2
[3] T. Healy. std::elide P3288R3
[4] https://github.com/bemanproject/emplace_from