Allocator Support for Operation States

Document #:	P3433R0
Date:	2024-10-11
Project:	Programming Language C++
Audience:	Library Evolution Working Group (LEWG) Library Working Group (LWG)
Reply-to:	Dietmar Kühl (Bloomberg) <dkuhl@bloomberg.net>

This document proposes allocator support for objects embedded into operation states created for asynchronous operations. The outer-most operation state defines the location where this state is stored and it can enable using `get_allocator()` on an environment to define the allocator for entities allocated by nested objects. However, the current specification doesn't forward this allocator when embedding user-provided sender arguments into operation states.

1 Motivation

Senders like `std::execution::just`, `std::execution::then`, and `std::execution::let_value` take some arguments which can be chosen fairly freely. Some of these arguments may be allocator-aware to offer control over their respective allocation needs. Carefully using allocators can allow, e.g., using standard container types without using global `operator new()`/`operator delete()`.

For example, consider this code (see here for the complete example):

```
int values[] = { 1, 2, 3 };
auto s{
    ex::just(std::span(values))
    | ex::let_value(allocator_aware_fun([](auto alloc, std::span<int> v){
            return ex::just(std::pmr::vector<int>(v.begin(), v.end(), alloc));
        }))
    | ex::then([](auto&& v) noexcept {
        for (auto x: v){ std::cout << x << ", "; }
        std::cout << "\n";
    })
};
```

The helper class template `allocator_aware_fun` wraps a function object and deals with appropriately managing an allocator which it also forwards as first argument to the wrapped function object. The `just()` with a `span` of values represents some asynchronous operation producing some data, e.g., based on network input exposing the content of some buffer. In actual code the `then()` would be an asynchronous operation not immediately completing but accessing the sequence of values at some later point. It is, thus, necessary to get the data into some controlled location. Copying the data to a `vector<int>` does the trick. However by default the needed memory would be allocated from the heap. The proper location is with the corresponding operation state which would be nested into other operation states. The outermost operation state on which the overall work gets `start()`ed would provide the appropriate allocator through the environment associated with its receiver.

The different sender algorithms currently just move their respective data into the operation state (see [exec.snd.expos] p36 or [exec.let] p7) which causes the default allocator or possibly (if objects are properly moved) an allocator belonging to the location of the sender constrution to be used. Assuming allocators are used neither of these allocators would match the allocator used for the operation state.

2 Proposal

The proposal is to use `std::make_obj_using_allocator` if the receiver's environment has an allocator set up instead of moving the object. If there is no matching allocator set up the objects are moved as they currently are. The recommendation for users would be to use a `std::pmr::polymorphic_allocator<>` for allocator-aware construction and to make a corresponding allocator available from the receiver's environment using something akin to *write-env*. How the environment is made available isn't part of this proposal: the specification already provides the means for users to control the environment.

One small complication of using `std::make_obj_using_allocator` is that the current specification of sender algorithms uses *product-type* in a few places which is a `tuple`-like class template supporting direct-initialization of the elements. In particular, the results of `just(obj...)` are stored using a specialization of *product-type*. This class template can't directly support `std::make_obj_using_allocator` because it can't have an allocator-aware constructor. However, the elements within can still be created using elementwise construction using `std::make_obj_using_allocator`.

3 Proposed Wording

In [exec.snd.expos] add a new paragraph at the end:

```
template <lt;typename T, typename Context>
decltype(auto) allocator-aware-move(T&& obj, Context&& context) // exposition-only
```

44 *allocator-aware-move* is an exposition-only function used to either create a new object of type `T` from `obj` or forward `obj` depending on whether an allocator is available. If the environment associated with `context` provides an allocator, i.e., the expression `get_allocator(get_env(context))` is valid and isn't `void`, let *alloc* be the result of this expression.

45 *Returns*:

(45.1) — If *alloc* is not defined returns `std::forward<T>(obj)`;

(45.2) — otherwise if `decay_t<T>` is not a specialization of *product-type* returns `make_obj_using_allocator<T>(std::forward<T>(obj), alloc)`;

(45.3) — otherwise returns an object of type `decay_t<T>` whose elements are initialized appropriately using `make_obj_using_allocator` with *alloc* and the corresponding element of `obj`.

In [exec.snd.expos] p36 change the `return` statement to use *allocator-aware-move*:

36 The member *default-impls::get-state* is initialized with a callable object equivalent to the following lambda:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept -> decltype(auto) {
  auto& [_, data, ...child] = sndr;
  return allocator-aware-move(std::forward_like<Sndr>(data), rcvr);
}
```

In [exec.let] p7 change the initialization of `fn` to use *allocator-aware-move*:

7 *impls-for< decayed-typeof>::get-state* is initialized with a callable object equivalent to the following:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) requires see below {
  auto& [_, fn, child] = sndr;
  using fn_t = decay_t<decltype(fn)>;
  using env_t = decltype(let-env(child));
  using args_variant_t = see below;
  using ops2_variant_t = see below;
```

```
  struct state-type {
    fn_t fn;                    // exposition only
    env_t env;                  // exposition only
    args_variant_t args;        // exposition only
    ops2_variant_t ops2;        // exposition only
  };
  return state-type{allocator-aware-move(std::forward_like<Sndr>(fn), rcvr),
                    let-env(child), {}, {}};
}
```

4 Implementation

The change is implemented by beman-project/execution26.