P0591r4 | Utility functions to implement uses-allocator construction

Pablo Halpern phalpern@halpernwightsoftware.com

2018-11-07 | Target audience: LWG

1 Abstract

The phrase "*Uses-allocator construction* with allocator `Alloc`" is defined in section **[allocator.uses.construction]** of the standard (23.10.7.2 of the 2017 DIS). Although the definition is reasonably concise, it fails to handle the case of constructing a `std::pair` where one or both members can use `Alloc`. This omission manifests in significant text describing the `construct` members of `polymorphic_allocator` [memory.polymorphic.allocator.class] and `scoped_allocator_adaptor` [allocator.adaptor]. Additionally neither `polymorphic_allocator` nor `scoped_allocator_adaptor` recursively pass the allocator to a `std::pair` in which one or both members is a `std::pair`.

Though we could add the `pair` special case to the definition of *Uses-allocator construction*, the definition would no longer be concise. Moreover, any library implementing features that rely on *Uses-allocator construction* would necessarily centralize the logic into a function template. This paper, therefore, proposes a set of templates that do exactly this centralization, in the standard. The current uses of *Uses-allocator construction* could then simply defer to these templates, making those features simpler to describe and future-proof against other changes.

Because this proposal modifies wording in the standard, it is targeted at C++20 (aka, C++Next) rather than at a technical specification.

2 History

Revision R4 approved by LWG in San Diego, 2018-11-07

2.1 Changes from R3

- Wording fixes from LWG review.
- Added missing function overloads to the synopsis section.
- Cleaned up description of *uses-allocator construction*.
- Fleshed out WP deltas to `polymorphic_allocator` and `scoped_allocator_adaptor`.
- Rebased to Oct 2018 C++ Working Draft

2.2 Changes from R2

- Renamed `make_using_allocator` to `make_obj_using_allocator` as per LEWG vote in Jacksonville.
- Added example of use.
- Changed audience to LWG after approval by LEWG in Jacksonville, March 2018.

2.3 Changes from R1

- Fix bugs in formal wording. Everything in this paper has been implemented and tested (and a link to the implementation added).
- Explicitly called out recursive handling for a `std::pair` containing a `std::pair`. (No change to actual functionality from R0.)
- Update section references to match C++17 DIS.
- Minor editorial changes.

2.4 Changes from R0

- Fixed function template prototypes, which incorrectly depended on partial specialization of functions.

3 Choosing a direction

Originally, I considered proposing a pair of function templates, `make_obj_using_allocator<T>(allocator, args...)` and `uninitialized_construct_using_allocator(ptrToT, allocator, args...)`. However, implementation experience with the feature being proposed showed that, given a type `T`, an allocator `A`, and an argument list `Args...`, it was convenient to generate a `tuple` of the final argument list for `T`'s constructor, then use `make_from_tuple` or `apply` to implement the above function templates. It occurred to me that exposing this `tuple`-building function may be desirable, as it opens the door to an entire category of functions that use `tuple`s to manipulate argument lists in a composable fashion.

4 Example

If we are creating a simple wrapper class around an object of type `T`, our class might look something like this:

```
template <typename T, class Alloc = std::allocator<T>>
class Wrapper {
  Alloc m_alloc;
  T     m_data;
  int   m_meta_data;
  ...
public:
  using allocator_type = Alloc;

  Wrapper(const T& v, const allocator_type& alloc = {});
  ...
};
```

This wrapper is intended to work with many types for `T`, including types that don't use an allocator at all, types that take an allocator on construction using the `allocator_arg` protocol, and types that take an allocator on construction as a trailing argument. The constructor for `Wrapper` would thus require much metaprogramming in order to handle all of the cases. This metaprogramming is already done by implementers of the standard library, because it is needed by `scoped_allocator_adaptor` and by `pmr::polymorphic_allocator`, but such metaprogramming is beyond the skills of most programmers and is time consuming for even those possessing the skills. The facilities in this proposal make the task almost trivial. The `Wrapper` constructor below takes advantage of the `make_obj_using_allocator` function template to construct the `m_data` member using the supplied allocator. The allocator will be ignored if `T` does not use an allocator or uses an allocator of a different type.

```
template <typename T, class Alloc>
Wrapper<T,Alloc>::Wrapper(const T& v, const Alloc& alloc)
```

```
  : m_alloc(alloc)
  , m_data(make_obj_using_allocator<T>(alloc, v))
  , m_meta_data(0)
  , etc.
  { ... }
```

5 Implementation experience

A working implementation of this proposal can be found on GitHub at https://github.com/phalpern/uses-allocator.git.

6 Proposed wording

Wording is relative to the October 2018 WP, N4778

6.1 Header `<memory>` synopsis [memory.syn]

Add the following new function templates to the to the `<memory>` synopsis:

```
template <class T, class Alloc, class... Args>
  auto uses_allocator_construction_args(const Alloc& alloc,
                                        Args&&... args) -> see below;

template <class T, class Alloc, class Tuple1, class Tuple2>
  auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
                                        Tuple1&& x, Tuple2&& y) -> see below;

template <class T>
  auto uses_allocator_construction_args(const Alloc& alloc) -> see below;

template <class T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) -> see below;

template <class T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& alloc,
                                        const pair<U,V>& pr) -> see below;

template <class T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& alloc, pair<U,V>&& pr) -> see below;

template <class T, class Alloc, class... Args>
  T make_obj_using_allocator(const Alloc& alloc, Args&&... args);

template <class T, class Alloc, class... Args>
  T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);
```

6.2 Uses-allocator construction [allocator.uses.construction]

Replace the entire [allocator.uses.construction] section with the following:

Uses-allocator construction with allocator `alloc` and constructor arguments `args...` refers to the construction of an object of type `T` such that `alloc` is passed to the constructor of `T` if `T` uses an allocator type compatible

with `alloc`. When applied to the construction of an object of type `T`, it is equivalent to initializing it with the value of the expression `make_obj_using_allocator<T>(alloc, args...)`, described below.

The following utility functions support three conventions for passing `alloc` to a constructor:

1. If `T` does not use an allocator compatible with `alloc`, then `alloc` is ignored.
2. Otherwise, if `T` has a constructor invocable as `T(allocator_arg, alloc, args...)` (leading-allocator convention), then uses-allocator construction chooses this constructor form.
3. Otherwise, if `T` has a constructor invocable as `T(args..., allocator)` (trailing-allocator convention), then uses-allocator construction chooses this constructor form.

The `uses_allocator_construction_args` function template takes an allocator and argument list and produces (as a tuple) a new argument list matching one of the above conventions. Additionally, overloads are provided that treat specializations of `std::pair` such that uses-allocator construction is applied individually to the `first` and `second` data members. The `make_obj_using_allocator` and `uninitialized_construct_using_allocator` function templates apply the modified constructor arguments to construct an object of type `T` as a return value or in-place, respectively. [*Note:* For `uses_allocator_construction_args` and `make_obj_using_allocator`, type `T` is not deduced and must therefore be specified explicitly by the caller. - *end note*]

```
template <class T, class Alloc, class... Args>
  auto uses_allocator_construction_args(const Alloc& alloc, Args&&... args) -> see below;
```

Constraints: `T` is *not* a specialization of `std::pair`.

Returns: A `tuple` value determined as follows:

- If `uses_allocator_v<T, Alloc>` is false and `is_constructible_v<T, Args...>` is true, return `forward_as_tuple(std::forward<Args>(args)...)`.
- Otherwise, if `uses_allocator_v<T, Alloc>` is true and `is_constructible_v<T, allocator_arg_t, Alloc, Args...>` is true, return `tuple<allocator_arg_t, const Alloc&, Args&&...>(allocator_arg, alloc, std::forward<Args>(args)...)`.
- Otherwise, if `uses_allocator_v<T, Alloc>` is true and `is_constructible_v<T, Args..., Alloc>` is true, return `forward_as_tuple(std::forward<Args>(args)..., alloc)`.
- Otherwise, the program is ill-formed. [*Note*: This definition prevents a silent failure to pass the allocator to a constructor of a type for which `uses_allocator_v<T, Alloc>` is true. — *end note*]

```
template <class T, class Alloc, class Tuple1, class Tuple2>
  auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
                                        Tuple1&& x, Tuple2&& y) -> see below;
```

Constraints: `T` is a specialization of `std::pair`.

Effects: For `T` specified as `pair<T1, T2>`, equivalent to:

```
return make_tuple(piecewise_construct,
          apply([&alloc](auto&&... args1) {
                  return uses_allocator_construction_args<T1>(alloc,
                         std::forward<decltype(args1)>(args1)...);
              }, std::forward<Tuple1>(x)),
          apply([&alloc](auto&&... args2) {
                  return uses_allocator_construction_args<T2>(alloc,
                         std::forward<decltype(args2)>(args2)...);
              }, std::forward<Tuple2>(y)));
```

```
template <class T>
  auto uses_allocator_construction_args(const Alloc& alloc) -> see below;
```

Constraints: `T` is a specialization of `std::pair`.

Effects: Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           tuple<>{}, tuple<>{});
```

```
template <class T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) -> see below;
```

Constraints: `T` is a specialization of `std::pair`.

Effects: Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           forward_as_tuple(std::forward<U>(u)),
                                           forward_as_tuple(std::forward<V>(v)));
```

```
template <class T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& alloc,
                                        const pair<U,V>& pr) -> see below;
```

Constraints: `T` is a specialization of `std::pair`.

Effects: Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           forward_as_tuple(pr.first),
                                           forward_as_tuple(pr.second));
```

```
template <class T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& alloc, pair<U,V>&& pr) -> see below;
```

Constraints: `T` is a specialization of `std::pair`.

Effects: Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           forward_as_tuple(std::move(pr).first),
                                           forward_as_tuple(std::move(pr).second));
```

```
template <class T, class Alloc, class... Args>
  T make_obj_using_allocator(const Alloc& alloc, Args&&... args);
```

Effects: Equivalent to:

```
return make_from_tuple<T>(
           uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...));
```

```
template <class T, class Alloc, class... Args>
  T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);
```

Effects: Equivalent to:

```
return ::new(static_cast<void*>(p))
           T(make_obj_using_allocator<T>(alloc, std::forward<Args>(args)...));
```

6.3 Changes to `polymorphic_allocator` and `scoped_allocator_adaptor`

In [mem.poly.allocator.class], remove the second through sixth `construct` member functions from `polymorphic_allocator`, (i.e., those whose first argument is a `pair`):

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);

template<class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct_t,
tuple<Args1...> x, tuple<Args2...> y);
template<class T1, class T2>
void construct(pair<T1, T2>* p);
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, U&& x, V&& y);
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& pr);
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& pr);
```

In [mem.poly.allocator.mem], , remove the second through sixth `construct` member functions from `polymorphic_allocator`, (i.e., those whose first argument is a `pair`).

In [mem.poly.allocator.mem], make the following edits to the first `construct` member function description:

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);
```

~~Requires~~ Mandates: Uses-allocator construction of `T` with allocator `*this` (see 19.10.8.2) and constructor arguments `std::forward<Args>(args)...` is well-formed. ~~[*Note:* Uses-allocator construction is always well formed for types that do not use allocators. — *end note*~~]

Effects: Construct a T object in the storage whose address is represented by `p` by uses-allocator construction with allocator `*this` and constructor arguments `std::forward<Args>(args)...`.

Throws: Nothing unless the constructor for `T` throws.

~~Remarks~~: ~~This function shall not participate in overload resolution if `T` is a specialization of `pair`.~~

In [allocator.adaptor.syn], remove the second through sixth `construct` member functions (the same ones removed from the synopsis) and their entire descriptions.

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);

template<class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct_t,
tuple<Args1...> x, tuple<Args2...> y);
template<class T1, class T2>
void construct(pair<T1, T2>* p);
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, U&& x, V&& y);
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& pr);
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& pr);
```

In [allocator.adaptor.members], remove the second through sixth `construct` member functions (the same ones removed from the synopsis) and their entire descriptions.

In [allocator.adaptor.members], completely replace the *Effects* and *Remarks* clauses for the first `construct` member function:

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);
```

~~*Effects*:~~

~~— If uses_allocator_v<T, inner_allocator_type> is false…~~

~~… *Remarks*: This function shall not participate in overload resolution if T is a specialization of pair.~~

Effects: Equivalent to:

```
apply([p,this](auto&&... newargs){
        OUTERMOST_ALLOC_TRAITS(*this)::construct(
          OUTERMOST(*this), p, std::forward<decltype(newargs)>(newargs)...);
      }, uses_allocator_construction_args(inner_allocator(),
                                          std::forward<Args>(args)...));
```

Drafting note: `OUTERMOST_ALLOC_TRAITS` and `OUTERMOST` should be *italicized*.