P0591r0 | Utility functions to implement uses-allocator construction

Pablo Halpern phalpern@halpernwightsoftware.com

2017-02-05 | Target audience: LEWG

1 Abstract

The phrase "*Uses-allocator construction* with allocator `Alloc`" is defined in section **[allocator.uses.construction]** of the standard (20.7.7.2 of the 2014 standard or 20.10.7.2 of the 2016 CD). 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, a `vector<pair<T,U>, A>` fails to pass the allocator to the pair elements if `A` is a scoped or polymorphic allocator.

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 that, 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 a technical specification.

2 Choosing a direction

Originally, I considered proposing a pair of function templates, `make_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.

The decision before the LEWG (assuming the basics of this proposal are accepted) would be whether to:

1. Standardize the function template that generates a `tuple` of arguments.
2. Standardize the function templates that actually construct a `T` from an allocator and list of arguments.
3. Both.

3 Proposed wording

The following wording is still rough. More detailed wording to come after LEWG review and revision. Wording is relative to the November 2016 Committee Draft, N5131.

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

```
template <class T, class Alloc, class... Args>
  auto uses_allocator_construction_args(const Alloc& a, Args&&... args);
```

Returns: A `tuple` value determined as follows:

- if `uses_allocator_v<T, Alloc>` is false and `is_constructible_v<T, Args...>` is true, return `make_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 `make_tuple(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 `make_tuple(std::forward<Args>(args)..., alloc)`.
- otherwise, the program is ill-formed. [*Note*: An error will result if `uses_allocator_v<T, Alloc>` is true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass the allocator to a constructor. — *end note*]

Editorial note: The following are specializations for `T` being `pair<T1,T2>`. They are not in the correct form for a specialization/overload because of the absence of partial specialization for functions. This detail will be corrected in the next version of this paper.

```
template <class T1, class T2, class... Args1, class... Args2>
  auto uses_allocator_construction_args(const Alloc& a,
                                        piecewise_construct_t,
                                        tuple<Args1...> x,
                                        tuple<Args2...> y);
```

Returns: Equivalent to

```
make_tuple(piecewise_construct,
           apply(x, [](Args1... args1){
              uses_allocator_construction_args<T1>(a,
                  std::forward<Args1>(args1)...);
           }),
           apply(y, [](Args2... args2){
              uses_allocator_construction_args<T2>(a,
                  std::forward<Args2>(args2)...);
           }));
```

```
template <class T1, class T2>
  auto uses_allocator_construction_args(const Alloc& a);
```

Returns: `uses_allocator_construction_args<pair<T1,T2>>(a, piecewise_construct, tuple<>(), tuple<>)`

```
template <class T1, class T2, class U, class V>
  auto uses_allocator_construction_args(const Alloc& a, U&& u, V&& v);
```

Returns: `uses_allocator_construction_args<pair<T1,T2>>(a, piecewise_construct, forward_as_tuple(std::forward<U>(u)), forward_as_tuple(std::forward<V>(v)))`.

```
template <class T1, class T2, class U, class V>
  auto uses_allocator_construction_args(const Alloc& a, const pair<U,V>& pr);
```

Returns: `uses_allocator_construction_args<pair<T1,T2>>(a, piecewise_construct, forward_as_tuple(pr.first), forward_as_tuple(pr.second))`.

```
template <class T1, class T2, class U, class V>
  auto uses_allocator_construction_args(const Alloc& a, pair<U,V>&& pr);
```

Returns: `uses_allocator_construction_args<pair<T1,T2>>(a, piecewise_construct, forward_as_tuple(std::forward<U>(pr.first)), forward_as_tuple(std::forward<V>(pr.second)))`.

```
template <class T, class Alloc, class... Args>
  T make_using_allocator(const Alloc& a, Args&&... args)
```

Returns: equivalent to

```
return make_from_tuple<T>(
    uses_allocator_construction_args<T>(a, forward<Args>(args)...));
```

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

Returns: equivalent to

```
return uninitialized_construct_from_tuple(
    p,
    uses_allocator_construction_args<T>(a, forward<Args>(args)...));
```

Additionally, rewrite the `construct` methods of `polymorphic_allocator` and `scoped_allocator_adaptor` in terms of the above.

Consider replacing all uses of *uses allocator construction* with references to these functions and removing *uses allocator construction* from the standard.