# P0591r2 | Utility functions to implement uses-allocator construction

Pablo Halpern phalpern@halpernwightsoftware.com

2017-06-12 | Target audience: LEWG

## 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 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.

## 3 Changes from R0

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

# 4 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.

If the basics of this proposal are accepted by LEWG, there would need to be a discussion of exactly what should be standardized. The options are:

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.

This proposal chooses option 3, but I am open to the other options.

# 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 March 2017 DIS, N4660.

## 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& a, Args&&... args) -> see below;

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

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

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

Add the following descriptions to uses-allocator-construction.

**Guidance needed**: The wording below expresses `uses_allocator_construction_args` as a bunch of overloads using "does not participate in overload-resolution" wording. It could also be expressed as a single (variadic) function with a bunch of special cases called out, or it could be described with less code and more descriptive English. Which is better for comprehending the standard?

**Guidance needed**: The wording uses `forward_as_tuple`, which prevents copies, and doesn't require copy- or move-constructibility, but can result in dangling references if the resulting `tuple` outlives the full expression in which it was created. Should I repeat the cautionary words already found in the description of `forward_as_tuple`?

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

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller. This template does not participate in overload resolution if `T` is 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 `forward_as_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 `forward_as_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*]

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

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller. This template does not participate in overload resolution unless `T` is a specialization of `std::pair`.

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

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

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

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller. This template does not participate in overload resolution unless `T` is a specialization of `std::pair`.

*Returns*: For `T` specified as `pair<T1, T2>`, equivalent to `uses_allocator_construction_args<pair<T1,T2>>(a, piecewise_construct, tuple<>{}, tuple<>{})`

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

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller. This template does not participate in overload resolution unless `T` is a specialization of `std::pair`.

*Returns*: For `T` specified as `pair<T1, T2>`, equivalent to `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 T, class Alloc, class U, class V>
  auto uses_allocator_construction_args(const Alloc& a, const pair<U,V>& pr) -> see below;
```

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller. This template does not participate in overload resolution unless `T` is a specialization of `std::pair`.

*Returns*: For `T` specified as `pair<T1, T2>`, equivalent to `uses_allocator_construction_args<pair<T1,T2>>(a, 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& a, pair<U,V>&& pr) -> see below;
```

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller. This template does not participate in overload resolution unless `T` is a specialization of `std::pair`.

*Returns*: For `T` specified as `pair<T1, T2>`, equivalent to `uses_allocator_construction_args<pair<T1,T2>>(a, piecewise_construct, forward_as_tuple(std::forward<U>(pr.first)), forward_as_tuple(std::forward<V>(`

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

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller.

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

```
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);
```

*Remark*: `T` is not deduced and must therefore be specified explicitly by the caller.

*Returns*: For T specified as pair<T1, T2>, equivalent to:

```
return apply([p](auto&&... args2){
        return ::new(static_cast<void*>(p))
            T(forward<decltype(args2)>(args2)...);
    }, uses_allocator_construction_args<T>(a, forward<Args>(args)...));
```

**Guidance Needed**: Should we consider adding uninitialized_construct_from_tuple as a separate (non-exposition) function, since it appears to be useful and would simplify (or perhaps eliminate the need for) uninitialized_construct_using_allocator.

## 6.3 Changes to polymorphic_allocator and scoped_allocator_adaptor

Rewrite the construct methods of polymorphic_allocator [mem.poly.allocator.mem] and scoped_allocator_adaptor [allocator.adaptor.members] to simply call uninitialized_construct_from_tuple.

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