P0209r1 | `make_from_tuple`: `apply` for construction

Pablo Halpern phalpern@halpernwightsoftware.com

2016-05-29 | Intended audience: LWG

1 Abstract

This paper proposes a function template that applies a `tuple` of arguments to an object constructor similar to the way `apply` works with non-constructor functions.

The template described in this paper is should be tied to the `apply` function, which is currently targeted for C++17. Therefore, this feature should also be targeted for C++17.

2 Changes from R0

- Removed `uninitialized_construct_from_tuple` as per LEWG review.
- Added `constexpr`
- Added an example.
- Re-based to the March 2016 C++17 working draft

3 Proposal

3.1 Motivation

N3915 introduced the `apply` function template into the Library Fundamentals TS. This template takes an invocable argument and a `tuple` argument and unpacks the `tuple` elements into an argument list for the specified invocable. While extremely useful for invoking a function, `apply` is not well suited for constructing objects from a list of arguments stored in a tuple. Doing so would require wrapping the object construction in a lambda or other function and passing that function to `apply`, a process that, done generically, is more complicated than the implementation of `apply` itself.

3.2 Summary

This proposal introduces a function template, `make_from_tuple`, to fill the void left by `apply`. The signature for `make_from_tuple` is:

```
template <class T, class Tuple>
  constexpr T make_from_tuple(Tuple&& t);`
```

It simply explodes it's `tuple` argument into separate arguments, which it passes to the constructor for type `T`, returning the newly-constructed object. Because of mandatory copy-elision in C++17, the return value is effectively constructed in place for the client.

3.3 Example

`make_from_tuple` can be used to implement the piecewise constructor for `std::pair` as follows:

```
template <class T1, class T2>
template <class... Args1, class... Args2>
pair<T1,T2>::pair(piecewise_construct_t,
                  tuple<Args1...> first_args, tuple<Args2...> second_args)
    : first(make_from_tuple<T1>(first_args))
    , second(make_from_tuple<T2>(second_args))
{
}
```

4 Scope

Pure-library extension

5 Alternatives considered

There has been discussion of making `tuple` functionality more tightly integrated into the core language in such a way that these functions would not be needed. More recently, a proposed `direct_initialize` facility would allow `apply` to work with constructors. Until such a time as such a proposal is accepted, however, these functions are simple enough, useful enough, and self-contained enough to consider for C++17 and would continue to be meaningful and convenient even if `direct_initialize` is accepted.

The names are, of course, up for discussion. A name that contains "apply" might be preferred, but I could think of no reasonable name that met that criterion. LEWG considered several names and stuck with `make_from_tuple`.

6 Implementation experience

The facility in this proposal have been fully implemented and tested. An open-source implementation under the Boost license is available at: https://github.com/phalpern/uses-allocator

7 Formal wording

The following changes are relative to the March 2016 C++17 working draft. N4582.

In section 20.4.1 ([tuple.general]), add the following declarations to the `<tuple>` header (within the `std` namespace), immediately after the declaration of `apply`:

```
    template <class T, class Tuple>
      constexpr T make_from_tuple(Tuple&& t);
```

In section 20.4.2.5 ([tuple.apply]), immediately after the description of `apply`, add the description for `make_from_tuple`:

```
template <class T, class Tuple>
  constexpr T make_from_tuple(Tuple&& t);`
```

Returns: Given the exposition-only function

```
template <class T, class Tuple, size_t... I>
constexpr T make_from_tuple_impl(Tuple&& t, index_sequence<I...>) { // exposition only
    return T(get<I>(forward<Tuple>(t))...);
}
```

Equivalent to

```
make_from_tuple_impl<T>(forward<Tuple>(t),
                        make_index_sequence<tuple_size_v<decay_t<Tuple>>>())
```

Note: The type of `T` must be supplied as an explicit template parameter, as it cannot be deduced from the argument list.