P0209r0 | `make_from_tuple`: `apply` for construction

Pablo Halpern phalpern@halpernwightsoftware.com

2015-02-12 | Intended audience: LEWG

1 Background

1.1 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. This proposal introduces a pair of function templates, `make_from_tuple` and `uninitialized_construct_from_tuple` to fill this void.

1.2 Target

The templates described in this paper are intended for inclusion in the next Library Fundamentals TS. However, these functions should move when `apply` moves, so if `apply` is added to C++17, these functions should also be added to C++17.

1.3 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. Until such a time as a proposal is accepted, however, these functions are simple enough and self-contained enough to be useful.

The names are, of course, up for discussion. A pair of names that contain "apply" might be preferred, but I could think of no reasonable name that met that criterion.

1.4 Scope

Pure-library extension

1.5 Implementation experience

The facilities 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

2 Formal wording

The following changes are relative to the Fundamentals TS version 2 PDTS, N4564.

In section 3.2.1 ([header.tuple.synop]), add the following declarations to the `<experimental/tuple>` header (within the `std::experimental::fundamentals_v3` namespace):

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

    template <class T, class Tuple>
      T* uninitialized_construct_from_tuple(T* p, Tuple&& t);
```

Add a new section after 3.2.2 ([tuple.apply]):

Constructing an object with a `tuple` of arguments [tuple.make_from]

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

Returns: Given the exposition-only function

```
template <class T, class Tuple, size_t... I>
T make_from_tuple_impl(Tuple&& t, index_sequence<I...>)
{
    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.

```
template <class T, class Tuple>
  T* uninitialized_construct_from_tuple(T* p, Tuple&& t);
```

Effects: Given the exposition-only function

```
template <class T, class Tuple, size_t... I>
T* uninitialized_construct_from_tuple_impl(T* p, Tuple&& t,
                                           index_sequence<I...>)
{
    return ::new((void*) p) T(get<I>(std::forward<Tuple>(t))...);
}
```

Equivalent to

```
make_from_tuple_impl<T>(forward<Tuple>(args_tuple),
                        make_index_sequence<tuple_sizev<decay_t<Tuple>>>{})
```

Returns: `p`