Make `std::make_from_tuple` SFINAE friendly

Document #:	P3738R0
Date:	2025-06-11
Project:	Programming Language C++
Audience:	Library Evolution Library

Reply-to:	Yihan Wang <yronglin777@gmail.com>

1 Introduction

This paper introduce constraints for `std::make_from_tuple` to make it SFINAE friendly.

2 Motivation

[LWG3528] introduce constraints:

```
template<class T, class Tuple, size_t... I>
  requires is_constructible_v<T, decltype(get<I>(declval<Tuple>()))...>
constexpr T make-from-tuple-impl(Tuple&& t, index_sequence<I...>) {     // exposition only
  return T(get<I>(std::forward<Tuple>(t))...);
}
```

When someone write SFINAE code like the following to check whether `T` can constructed from a tuple, they may hit hard errors like "no matching function for call to `make-from-tuple-impl`" (Compiler Explorer).

```
template <typename T, typename Tuple, typename = void>
inline constexpr bool has_make_from_tuple = false;

template <typename T, typename Tuple>
inline constexpr bool has_make_from_tuple<
    T, Tuple,
    std::void_t<decltype(std::make_from_tuple<T>(std::declval<Tuple>()))>> =
    true;
struct A {
    int a;
};
static_assert(!has_make_from_tuple<int *, std::tuple<A *>>);
```

Even If the effects are *Equivalent to* calling a constrained function, the constraints has not apply to `std::make_from_tuple`.

This is somehow unclear when the constraints are not literally specified with *Constraints* in the standard wording (16.3.2.4 [structure.specifications]). At least *Equivalent to* doesn't propagate every substitution failure in immediate context. In the case of `make-from-tuple-impl`, the constraints were introduced via a requires-clause but not literal *Constraints*. Some implementors believed the requires-clause should be treated same as *Constraints*, but this is not explicitly stated.

3 Impact on the Standard

This proposal is a pure library improvement.

4 Implementation Experience

I've implemented this improvement in `libc++`, `microsoft/STL`, `libstdc++`.

5 Proposed Wording

Modify section 22.4.6 22.4.6 [tuple.apply] as indicated:

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

Mandates: If `tuple_size_v<remove_reference_t<Tuple>>` is 1, then `reference_constructs_from_temporary_v<T, decltype(get<0>(declval<Tuple>()))>` is `false`.

Let `I` be the pack `0, 1, ..., (tuple_size_v<remove_reference_t<Tuple>> - 1)`.

Constraints:

— `is_constructible_v<T, decltype(get<I>(declval<Tuple>()))...>` is `true`.

Effects: Given the exposition-only function template:

```
namespace std {
  template<class T, tuple-like Tuple, size_t... I>
    requires is_constructible_v<T, decltype(get<I>(declval<Tuple>()))...>
  constexpr T make-from-tuple-impl(Tuple&& t, index_sequence<I...>) {   // exposition only
    return T(get<I>(std::forward<Tuple>(t))...);
  }
}
```

Equivalent to:

```
return make-from-tuple-impl<T>(
           std::forward<Tuple>(t),
           make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{});
```

[*Note:* Note 1: The type of T must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. - end note — *end note*]

6 Acknowledgements

Thank you to Jiang An, Mark de Wever, Stephan T. Lavavej, Jonathan Wakely, Barry Revzin, Daniel Krügler, and everyone else who contributed to the discussions, and encouraged me to write this paper.

7 References

[LWG3528] Tim Song. make_from_tuple can perform (the equivalent of) a C-style cast. https://wg21.link/lwg3528