Remove return type deduction in `std::apply`

Aaryaman Sagar (aary@instagram.com)

October 7, 2018
Document #: p1317
Library Evolution Working Group

1. Introduction

```
#include <tuple>
#include <iostream>

template <typename VoidT, typename Func, typename... Args>
struct well_formed : std::false_type {};

template <typename Func, typename... Args>
struct well_formed<
    std::void_t<decltype(
        std::apply(std::declval<Func>(), std::declval<Args>()...))>,
    Func,
    Args...> : std::true_type {};

void foo(bool) {}

int main() {
  auto func = []() {};
  auto args = std::make_tuple(1);

  std::ignore = well_formed<void, decltype(func)&, decltype(args)&>{};
}
```

The code above should be well formed. However, since `std::apply` uses return type deduction to deduce the return type, we get a hard error as the substitution is outside the immediate context of the template instantiation.

This paper proposes a new public trait instead of `decltype(auto)` in the return type of of `std::apply`

2. Impact on the standard

This proposal is a pure library extension.

3. `std::apply_result`

`std::apply_result` (and the corresponding alias `std::apply_result_t`) is the proposed trait that should be used in the return type of `std::apply`. With the new declaration being

```
template <class F, class Tuple>
constexpr std::apply_result_t<F, Tuple> apply(F&& f, Tuple&& t);
```

This fixes hard errors originating from code that tries to employ commonly-used SFINAE patterns with `std::apply` that could have otherwise been well formed. And is backwards compatible with well-formed usecases of `std::apply`

4. Implementation

`std::apply_result` can be defined using the existing `std::invoke_result` trait to avoid duplication in implementations

```
// apply_impl is for exposition only
template <class F, class T, std::size_t... I>
constexpr auto apply_impl(F&& f, T&& t, std::index_sequence<I...>) noexcept(
    is_nothrow_invocable<F&&, decltype(std::get<I>(std::declval<T>()))...>{})
    -> invoke_result_t<F&&, decltype(std::get<I>(std::declval<T>()))...> {
  return invoke(std::forward<F>(f), std::get<I>(std::forward<T>(t))...);
}

template <typename F, typename Tuple>
using apply_result_t = decltype(apply_impl(
    std::declval<F>(),
    std::declval<Tuple>(),
    std::make_index_sequence<std::tuple_size_v<std::decay_t<Tuple>>>{}));

template <typename F, typename Tuple, typename = std::void_t<>>
class apply_result {};
template <typename F, typename Tuple>
class apply_result<F, Tuple, std::void_t<std::apply_result_t<F, Tuple>>> {
  using type = std::apply_result_t<F, Tuple>;
};
```

5. Changes to the standard

5.1. Section 23.5.2 ([[tuple.syn]])

```
// 23.5.3.5, calling a function with a tuple of arguments
template <class F, class Tuple>
constexpr std::apply_result_t<F, Tuple> apply(F&& f, Tuple&& t);

template <class F, class Tuple> class apply_result;
template <class F, class Tuple> using apply_result_t = (see below);
```

5.2. Section 23.5.3.5 ([tuple.apply] paragraph 1)

```
template <class F, class Tuple>
constexpr std::apply_result_t<F, Tuple> apply(F&& f, Tuple&& t) noexcept(see below);
```

Effects: Given the exposition-only function:

```
template <class F, class T, std::size_t... I>
constexpr auto apply_impl(F&& f, T&& t, std::index_sequence<I...>) noexcept(
    is_nothrow_invocable<F&&, decltype(std::get<I>(std::declval<T>()))...>{})
    -> invoke_result_t<F&&, decltype(std::get<I>(std::declval<T>()))...> {
  return invoke(std::forward<F>(f), std::get<I>(std::forward<T>(t))...);
}
```

Equivalent to:

```
return apply_impl(
    std::forward<F>(f),
    std::forward<Tuple>(t),
    std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>{}>{});
```

Remarks: The expression inside noexcept is equivalent to:

```
apply_impl(std::forward<F>(f), std::forward<Tuple>(t),
           make_index_sequence<tuple_size_v<decay_t<Tuple>>>);
```

5.3. Section 23.5.3.6 ([tuple.helper])

```
template <class F, class Tuple> using apply_result_t = (see below)
```

Effects: Given the exposition-only function:

```
template <class F, class T, std::size_t... I>
constexpr auto apply_impl(F&& f, T&& t, std::index_sequence<I...>) noexcept(
    is_nothrow_invocable<F&&, decltype(std::get<I>(std::declval<T>()))...>{})
    -> invoke_result_t<F&&, decltype(std::get<I>(std::declval<T>()))...> {
  return invoke(std::forward<F>(f), std::get<I>(std::forward<T>(t))...);
}
```

Equivalent to:

```
decltype(apply_impl(std::forward<F>(f), std::forward<Tuple>(t),
         make_index_sequence<tuple_size_v<decay_t<Tuple>>>{});

template <typename F, typename Tuple, typename = void_t<>>
class apply_result {};
template <typename F, typename Tuple>
class apply_result<F, Tuple, void_t<apply_result_t<F, Tuple>>> {
        using type = apply_result_t<F, Tuple>;
};
```

If the expression `apply_impl(std::declval<F>(), std::declval<Tuple>())` is well-formed when treated as an unevaluated operand (Clause 8), the member typedef `type` names the type `decltype(apply_impl(std::declval<F>(), std::declval<Tuple>()))`. Otherwise there shall be no member type. Access checking is performed as if in a context unrelated to `F` and `Tuple`. Only the validaity of the immediate context is considered. [Note: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions and so on. Such side-effects are not in the ïmmediate context.ªnd can result in the program being ill-formed. - end note]