

`apply()` call a function with arguments from a tuple

Peter Sommerlad

2013-10-08

Document Number:	N3802
Date:	2013-10-08
Project:	Programming Language C++

1 History

1.1 `integer_sequence`

N3658 and its predecessor N3493 introduced `integer_sequence` facility and provide application of this features, for example `apply()` that is proposed in this paper.

1.2 Observations

There is a lot of history I am unaware of and several implementations posted on Stack-Overflow. Also the C++14 CD contains an implementation of `apply()` as an example of `std::integer_sequence` in [intseq.general].

1.3 Using `tuplevar...` or `operator...`

Mike Spertus made me aware of the proposed language extension to form a parameter pack from a tuple, i.e., by overloading an `operator...` which might make the provision of `apply()` moot. However, up to now, no such feature has been proposed to the standard committee and it is unclear if it would make it into C++17. Even if it would, it would just make the implementation of `apply` trivial.

2 Introduction

Tuples are great for generic programming with variadic templates. However, the standard does not define a general purpose facility that allows to call a function/functor/lambda with the tuple elements as arguments. Such a feature should be provided, because it is useful (at least for me). It even is given as an example of `std::integer_sequence` in [intseq.general] coming from N3658.

2.1 Rationale

It is easy to create tuples from variadic templates either from types directly as `std::tuple<PACK...>` or with `std::make_tuple()` or `std::forward_as_tuple()` the opposite mechanism of passing a tuple's elements as function arguments is not available.

Some suggested to not use the name `apply()` and reserve that to a mechanism like *INVOKE* in the standard library and use `apply_from_tuple()` instead.

2.2 Open Issues to be Discussed

It is open, if `apply()` should get a `noexcept(noexcept(xxx))` specification delegating to the underlying effect. Advice from library working group was requested, but it seems that it shouldn't get one, because library only provides `noexcept` specifications for "special" functions, like ctors or swap.

Should `apply()` be `constexpr` to make it applicable in meta programming or concepts? LEWG advice is required.

2.3 Acknowledgements

Acknowledgements go to Jonathan Wakely for providing `integer_sequence` and providing `apply()` as the example in the working draft.

Acknowledgements go to Mike Spertus for making me aware of the ... pack formation approaches.

Acknowledgements to the following persons on the `c++std-lib` reflector for their feedback and encouragement: Jonathan Wakely, Andy Sawyer, Stephan T. Lavavej, Jared Hoberock, and Tony van Eerd.

3 Possible Implementation

The following implementation suggestion was derived from N3658, N3690, and Stack-Overflow (<http://stackoverflow.com/a/12650100>) and some simplification. It actually seems to work with current `clang -std=c++1y`.

```
template <typename F, typename Tuple, size_t... I>
decltype(auto) apply_impl(F&& f, Tuple&& t, index_sequence<I...>) {
    return forward<F>(f)(get<I>(forward<Tuple>(t))...);
}
template <typename F, typename Tuple>
decltype(auto) apply(F&& f, Tuple&& t) {
    using Indices = make_index_sequence<tuple_size<decay_t<Tuple>>::value>;
    return apply_impl(forward<F>(f), forward<Tuple>(t), Indices{});
}
```

4 Proposed Library Additions

Add the following declaration in [tuple.general] in the synopsis under the group *element access*:

```
template <typename F, typename Tuple, size_t... I>
decltype(auto)
apply_impl(F&& f, Tuple&& t, index_sequence<I...>) // exposition only
{
    return forward<F>(f)(get<I>(forward<Tuple>(t))...);
}
```

```
template <typename F, typename Tuple>
decltype(auto) apply(F&& f, Tuple&& t);
```

Append the following to section [tuple.elem] after paragraph 11.

```
template <typename F, typename Tuple>
decltype(auto) apply(F&& f, Tuple&& t);
```

- ¹ *Effects:* Equivalent to `return apply_impl(forward<F>(f), forward<Tuple>(t), make_index_sequence<tuple_size<decay_t<Tuple>>::value>{});`

Delete the example code in [intseq.general] p 2.