Mundane std::tuple tricks: Selecting via an index sequence, part 2

devblogs.microsoft.com/oldnewthing/20200624-00

June 24, 2020

Raymond Chen

Last time, we developed the `select_tuple` function which takes a tuple and an index sequence and produces a new tuple that selects the elements based on the index sequence. Here's what we had:

```
// Don't use this; see discussion.
template<typename Tuple, std::size_t... Ints>
auto select_tuple(Tuple&& tuple, std::index_sequence<Ints...>)
{
 return std::make_tuple(
    std::get<Ints>(std::forward<Tuple>(tuple))...);
}
```

The idea is that you can do something like

```
std::tuple<int, char, float> t{ 1, 'x', 2.0 };
auto t2 = select_tuple(t, std::index_sequence<0, 2>{});
```

and the result is that `t2` is a `std::tuple<int, float>{ 1, 2.0 }`.

But there's a problem with this function.

Here's a riddle: When does `std::make_tuple<T>()` return something that isn't a `std::tuple<T>`?

`std::make_tuple<T>`	Produces `std::tuple<T>`
`int`	`int`
`const int`	
`int&`	
`int&&`	
`std::reference_wrapper< int >`	`int&`

`std::reference_wrapper<const int >`
`std::reference_wrapper< int& >`
`std::reference_wrapper< int&&>`

Answer: When `T` is subject to decay or decays to a `reference_wrapper`.

Decay is a term in the C++ standard that refers to the changes of type that typically occur when something is passed by value to a function:

- References decay to the underlying type.
- cv-qualifiers (`const` and `volatile`) are removed.
- Arrays decay to pointers.
- Function decay to function pointers.

But `make_tuple` adds an additional wrinkle: If the decayed type is a `reference_wrapper`, then the result is the underlying reference.

We don't want any of these transformations to occur. If you select a type from a tuple that is a reference, then you want the resulting tuple to have the same reference type.

So we can't use `make_tuple`. We'll specify the desired tuple type explicitly.

```
template<typename Tuple, std::size_t... Ints>
auto select_tuple(Tuple&& tuple, std::index_sequence<Ints...>)
{
 return std::tuple<std::tuple_element_t<Ints, Tuple>...>(
    std::get<Ints>(std::forward<Tuple>(tuple))...);
}
```

or alternatively

```
template<typename Tuple, std::size_t... Ints>
std::tuple<std::tuple_element_t<Ints, Tuple>...>
select_tuple(Tuple&& tuple, std::index_sequence<Ints...>)
{
 return { std::get<Ints>(std::forward<Tuple>(tuple))... };
}
```

Okay, now that we have this helper function, we can do a bunch of fancy tuple manipulation.

Which we'll do next time.

Raymond Chen

Follow