Mundane std::tuple tricks: Creating interesting index sequences

devblogs.microsoft.com/oldnewthing/20200625-00

June 25, 2020

Raymond Chen

We saw last time that manipulating tuples boils down to the `index_sequence`. The problem is that the C++ standard library doesn't provide very much in the way of helpers to manipulate these index sequences.

The only ones that come with the standard library are `make_integer_sequence`, a helper for generating the integer sequence `0, 1, 2, …, N-1`, and its close relative `make_index_sequence`, which does the same thing for `size_t`.

Note that the template parameter to `make_index_sequence` is the size of the resulting index sequence, not the highest value in the index sequence. The highest value is one less than the size since the sequence starts at zero.

Even with only index sequences that start at zero, we can do something interesting.

```
template<typename Tuple>
auto remove_last(Tuple&& tuple)
{
  constexpr auto size = std::tuple_size_v<Tuple>;
  using indices = std::make_index_sequence<size-1>;
  return select_tuple(std::forward<Tuple>(tuple), indices{});
}
```

The `remove_last` function removes the last element from the tuple and returns what's left. We do this by extracting the size of the source tuple, subtracting one, and using that to generate a new index sequence that goes from 0 to `size` − 2, for a total of `size` − 1 elements.

Okay, but what about other index sequences? We'll have to build those ourselves.

```
template<std::size_t N, typename Seq> struct offset_sequence;

template<std::size_t N, std::size_t... Ints>
struct offset_sequence<N, std::index_sequence<Ints...>>
{
 using type = std::index_sequence<Ints + N...>;
};
template<std::size_t N, typename Seq>
using offset_sequence_t = typename offset_sequence<N, Seq>::type;

// example = index_sequence<3, 4, 5, 6>
using example = offset_sequence_t<3, std::make_index_sequence<4>>;
```

To offset an index sequence, we generate a new index sequence whose integers are the old integers plus the provided offset value *N*. The magic happens in the template parameter pack expansion:

```
 using type = std::index_sequence<Ints + N...>;
```

This takes each of the integers in the original index sequence, adds *N* to each one, and uses the results to build a new index sequence.

Now we can remove the *first* element from the tuple.

```
template<typename Tuple>
auto remove_first(Tuple&& tuple)
{
  constexpr auto size = std::tuple_size_v<Tuple>;
  using indices = offset_sequence_t<1,
                    std::make_index_sequence<size-1>>;
  return select_tuple(std::forward<Tuple>(tuple), indices{});
}
```

And in fact, we can remove the *N*th element.

```
template<std::size_t N, typename Tuple>
auto remove_Nth(Tuple&& tuple)
{
  constexpr auto size = std::tuple_size_v<Tuple>;
  using first = std::make_index_sequence<N>;
  using rest = offset_sequence_t<N+1,
                std::make_index_sequence<size-N-1>>;
  return std::tuple_cat(
    select_tuple(std::forward<Tuple>(tuple), first{}),
    select_tuple(std::forward<Tuple>(tuple), rest{}));
}
```

What we want to do is extract the first *N* elements, skip element *N*, and then extract elements $N + 1$ to the end.

Extracting the first *N* is easy: We select from the index sequence that goes from 0 to $N - 1$.

Extracting the rest of the elements is trickier: We want to start at $N + 1$ and continue until `size` $- 1$, for a length of (`size` $- 1) - (N + 1) + 1 =$ `size` $- N - 1$. We generate an index sequence of length `size` $- N - 1$ (starting at zero), then add $N + 1$ to each element, to bring us to the desired sequence.

We call `select_tuple` twice, once to get the front part, once to get the back part, and then use `tuple_cat` to put them together.

Another way to do this is with a single selection. For that, we need a way to concatenate two index sequences.

```
template<typename Seq1, typename Seq> struct cat_sequence;

template<std::size_t... Ints1, std::size_t... Ints2>
struct cat_sequence<std::index_sequence<Ints1...>,
                    std::index_sequence<Ints2...>>
{
 using type = std::index_sequence<Ints1..., Ints2...>;
};
template<typename Seq1, typename Seq2>
using cat_sequence_t = typename cat_sequence<Seq1, Seq2>::type;

// example = index_sequence<3, 1, 4, 1, 5, 9>
using example = cat_sequence_t<std::index_sequence<3, 1, 4>,
                               std::index_sequence<1, 5, 9>>;
```

The magic happens at

```
 using type = std::index_sequence<Ints1..., Ints2...>;
```

which takes the two sequences and places them one after the other into a single index sequence.

We can now use this alternate formulation:

```
template<std::size_t N, typename Tuple>
auto remove_Nth(Tuple&& tuple)
{
  constexpr auto size = std::tuple_size_v<Tuple>;
  using first = std::make_index_sequence<N>;
  using rest = offset_sequence_t<N+1,
               std::make_index_sequence<size-N-1>>;
  using indices = cat_sequence_t<first, rest>;
  return select_tuple(std::forward<Tuple>(tuple), indices{});
}
```

We'll continue manipulating index sequences next time.

Raymond Chen

Follow