Mundane std::tuple tricks: Finding a type in a tuple

devblogs.microsoft.com/oldnewthing/20200629-00

June 29, 2020

Raymond Chen

Given a tuple, you can extract the type of the *N*th element in the tuple with the help of `std::tuple_element_t<N, Tuple>`:

```
// example = char
using example = std::tuple_element_t<1,
                  std::tuple<int, char, float>>;
```

The index is zero-based, so element 0 is `int`, element 1 is `char`, and element 2 is `float`.

What about going in reverse? Given a tuple, find the index of a specific type.

```
template<typename T, typename Tuple>
struct tuple_element_index_helper;
```

We start by writing a `tuple_ element_ index_ helper` which does the heavy lifting. It sets `value` equal to the index of the first element whose type matches, or equal to the number of elements (*i.e.*, one greater than the maximum legal element index) if the type was not found.

```
template<typename T>
struct tuple_element_index_helper<T, std::tuple<>>
{
  static constexpr std::size_t value = 0;
};
```

This is the base case. The type is not found in the empty tuple, so we set the `value` to zero.

```
template<typename T, typename... Rest>
struct tuple_element_index_helper<T, std::tuple<T, Rest...>>
{
  static constexpr std::size_t value = 0;
  using RestTuple = std::tuple<Rest...>;
  static_assert(
    tuple_element_index_helper<T, RestTuple>::value ==
    std::tuple_size_v<RestTuple>,
    "type appears more than once in tuple");
};
```

This is the success case. The type is the first element in the tuple. Therefore, the `value` is zero (index zero). We also validate that the type is not present in the remaining types of the tuple. If the `value` for `std::tuple<Rest...>` is not equal to the size of the tuple, then that means that the type was found among the remaining types, so we raise a compile-time assertion failure.

```
template<typename T, typename First, typename... Rest>
struct tuple_element_index_helper<T, std::tuple<First, Rest...>>
{
  using RestTuple = std::tuple<Rest...>;
  static constexpr std::size_t value = 1 +
       tuple_element_index_helper<T, RestTuple>::value;
};
```

And then we have the failure case. If the type does not match the first element in the tuple, then we search for the type in the remaining elements and add one to account for the fact that we removed the first type.

```
template<typename T, typename Tuple>
struct tuple_element_index
{
  static constexpr std::size_t value =
    tuple_element_index_helper<T, Tuple>::value;
  static_assert(value < std::tuple_size_v<Tuple>,
                "type does not appear in tuple");
};
```

Now that the helper is written, we can write the real template class. It asks the helper to do the work and validates that the resulting `value` is less than the size of the tuple, meaning that the type was found. If not, then we complain with a compile-time assertion.

```
template<typename T, typename Tuple>
inline constexpr std::size_t tuple_element_index_v
 = tuple_element_index<T, Tuple>::value;
```

Finally, we create an alias type to reduce future typing.

Let’s take it out for a spin.

```
// index = 1
constexpr std::size_t index =
    tuple_element_index_v<int, std::tuple<char, int, float>>;

// error: type does not appear in tuple
constexpr std::size_t index =
    tuple_element_index_v<double, std::tuple<char, int, float>>;

// error: type appears more than once in tuple
constexpr std::size_t index =
    tuple_element_index_v<int, std::tuple<char, int, int>>;
```

All of these mundane tuple tricks will come in handy soon.

Bonus chatter: You might be tempted to try something like this:

```
template<typename T, typename Tuple>
constexpr int tuple_element_index_helper()
{
  for (int i = 0; i < std::tuple_size_v<Tuple>; i++) {
    if constexpr (std::is_same_v<
        T, std::tuple_element_t<i, Tuple>>) {
      return i;
    }
  }
  return std::tuple_size_v<Tuple>;
}
```

Unfortunately, it doesn't work because a variable `i` is not valid as a template non-type parameter, not even in a `constexpr` context.

Raymond Chen

Follow