A simple workaround for the fact that std::equal takes its predicate by value

devblogs.microsoft.com/oldnewthing/20190617-00

June 17, 2019

Raymond Chen

The versions of the `std::equal` function that takes a binary predicate accepts the predicate by value, which means that if you are using a functor, it will be copied, which may be unnecessary or unwanted.

In my case, the functor had a lot of state, and I didn't want to copy it.

```
class comparer
{
  ...

  template<typename R>
  bool ranges_equiv(R const& left, R const& right)
  {
    using T = typename std::decay_t<decltype(*begin(left))>;
    return std::equal(
      begin(left), end(left),
      begin(right), end(right),
      equiv<T>);
  }

  template<typename T>
  bool equiv(T const& left, T const& right) = delete;

  template<>
  bool equiv(Doodad const& left, Doodad const& right)
  {
    return (!check_names || equiv(left.Name(), right.Name())) &&
           (!check_children || ranges_equiv(left.Children(), right.Children()));
  }

  ... other overloads omitted ...
};
```

The idea behind the `comparer` is that you configure it with information about what you care about and what you don't, and then you call `equiv` and let it walk the object hierarchy comparing the things you asked for according to the rules you specified.

This works great, except that `std::equal` copies its predicate, and our `comparer` is somewhat expensive to copy, since it may have lots of configuration `std::string`s and stuff. What we're looking for is a version that takes the predicate by reference, so that we can use the same `comparer` all the way down.

The workaround is to replace the predicate with something that is cheap to copy.

```
template<typename R>
bool ranges_equiv(R const& left, R const& right)
{
  return std::equal(
    begin(left), end(left),
    begin(right), end(right),
    [this](auto&& l, auto&& r) { return equiv(l, r); });
}
```

Instead of passing a full `comparer` object, we pass a lambda that captures the `comparer`'s `this` pointer. This lambda is cheap to copy, and it allows us to reuse the same `comparer` all the way down the object hierarchy.

This solution looks obvious in retrospect, but I got all hung up trying to create a cheap copyable object, like a nested type called `compare_forwarder` that kept a `std::reference_wrapper` to the `comparer`, before realizing that I was just writing a verbose version of a lambda.

Raymond Chen

Follow