Getting a value from a std::variant that matches the type fetched from another variant

devblogs.microsoft.com/oldnewthing/20190620-00

June 20, 2019

Raymond Chen

Suppose you have two `std::variant` objects of the same type and you want to perform some operation on corresponding pairs of types.

```
using my_variant = std::variant<int, double, std::string>;

bool are_equivalent(my_variant const& left,
                    my_variant const& right)
{
  if (left.index() != right.index()) return false;

  switch (left.index())
  {
  case 0:
    return are_equivalent(std::get<0>(left),
                          std::get<0>(right));
    break;

  case 1:
    return are_equivalent(std::get<1>(left),
                          std::get<1>(right));
    break;

  default:
    return are_equivalent(std::get<2>(left),
                          std::get<2>(right));
    break;
  }
}
```

Okay, what’s going on here?

We have a `std::variant` that can hold one of three possible types. First, we see if the two variants are even holding the same types. If not, then they are definitely not equivalent.

Otherwise, we check what is in the `left` object by switching on the index, and then check if the corresponding contents are equivalent.

In the case I needed to do this, the variants were part of a recursive data structure, so the recursive call to `are_equivalent` really did recurse deeper into the data structure.

There's a little trick hiding in the `default` case: That case gets hit either when the index is 2, indicating that we have a `std:string`, or when the index is `variant_` `npos`, indicating that the variant is in a horrible state. If it does indeed hold a string, then the calls to `std::get<2>` succeed, and if it's in a horrible state, we get a `bad_variant_access` exception.

This is tedious code to write. Surely there must be a better way.

What I came up with was to use the visitor pattern with a templated handler.

```
bool are_equivalent(my_variant const& left,
                    my_variant const& right)
{
  if (left.index() != right.index()) return false;

  return std::visit([&](auto const& l)
    {
      using T = std::decay_t<decltype(l)>;
      return are_equivalent(l, std::get<T>(right));
    }, left);
}
```

After verifying that the indices match, we visit the variant with a generic lambda and then reverse-engineer the appropriate getter to use for the right hand side by studying the type of the thing we were given. The `std::get<T>` will not throw because we already validated that the types match. (On the other hand, the entire `std::visit` could throw if both `left` and `right` are in horrible states.)

Note that this trick fails if the variant repeats types, because the type passed to `std::get` is now ambiguous.

Anyway, I had to use this pattern in a few places, so I wrote a helper function:

```
template<typename Template, typename... Args>
decltype(auto)
get_matching_alternative(
    const std::variant<Args...>& v,
    Template&&)
{
    using T = typename std::decay_t<Template>;
    return std::get<T>(v);
}
```

You pass this helper the variant you have and something that represents the thing you want, and the function returns the corresponding thing from the variant. With this helper, the `are_ equivalent` function looks like this:

```
bool are_equivalent(my_variant const& left,
                    my_variant const& right)
{
  if (left.index() != right.index()) return false;

  return std::visit([&](auto const& l)
    {
      return are_equivalent(l,
                   get_matching_alternative(right, l));
    });
}
```

I'm still not entirely happy with this, though. Maybe you can come up with something better?

Raymond Chen

Follow