Of what use is a type-dependent expression that is always false?

devblogs.microsoft.com/oldnewthing/20200312-00

March 12, 2020

Raymond Chen

Last time, we saw how to create a type-dependent expression that is always false, and used it in a potentially-discarded statement so that the assertion failed only if the statement ended up being used.

Another case where you want to defer a static assertion failure to instantiation is if you want to reject a particular specialization.

Say you have a method that you want to overload, but a particular version of the overload is disallowed. You could use `std::enable_if` to remove that overload from consideration, leading to a compiler error of the form “No suitable overload found.”

For example, suppose we have a `buffer_ view` that represents the raw bytes stored in a vector.

```
struct buffer_view
{
  template<typename C>
  buffer_view(std::vector<C> const& v) :
    data(v.data()), size(v.size() * sizeof(C)) { }

  // Imagine other constructors for std::array, etc.

  void const* data;
  std::size_t size;
};
```

The idea here is that this is a buffer for passing raw bytes to another function. Therefore, in practice, you probably would add a

```
typename = std::enable_if_t<std::is_trivial<C>::value>
```

to the template parameters, so that people won’t try to pass things like `std::string` as a buffer. In practice, you probably also would want a second template parameter `std::vector<T, Alloc>` in order to support non-default allocators. But I’ve left off these adjustments to simplify the exposition.

This class works great until somebody tries this:

```
std::vector<bool> flags;
auto view = buffer_view(flags);
```

The C++ language defines a specialization `std::vector<bool>` which represents a packed bit array, rather than defining a separate type like `std::bitvector`. This has made a lot of people very angry and has been widely regarded as a bad move.

One of the quirks of `std::vector<bool>` is that it lacks a `data()` method.

If you pass in a `std::vector<bool>`, you get the weird error from the Microsoft compiler:

```
error C2039: 'data': is not a member of 'std::vector<bool, std::allocator<_Ty>>'
        with
        [
            _Ty=bool
        ]
note: see declaration of 'std::vector<bool, std::allocator<_Ty>>'
        with
        [
            _Ty=bool
        ]
note: see reference to function template instantiation
'buffer_view::buffer_view<bool>(const std::vector<bool, std::allocator<_Ty>> &)'
being compiled
        with
        [
            _Ty=bool
        ]
```

gcc and clang produce a completely bizarre error:

```
error: 'this' argument to member function 'data' has type 'const std::vector<bool>',
but function is not marked const
    data(v.data()), size(v.size() * sizeof(C)) { }
         ^
```

I mean, technically, all the error messages are “correct” in the sense that all the standard requires is the generation of a diagnostic, but does not require that the diagnostic be useful.

We might try to improve the error message by specializing the constructor for `std::vector<bool>` and deleting it.

```
template<>
buffer_view::buffer_view(std::vector<bool> const& v) = delete;
```

Now the error messages are a little better:

```
error C2280: 'buffer_view::buffer_view<bool>(const std::vector<bool,
std::allocator<_Ty>> &)':
attempting to reference a deleted function
        with
        [
            _Ty=bool
        ]
note: see declaration of 'buffer_view::buffer_view'
note: 'buffer_view::buffer_view<bool>(const std::vector<bool, std::allocator<_Ty>>
&)': function was explicitly deleted
        with
        [
            _Ty=bool
        ]

call to deleted constructor of 'buffer_view'

use of deleted function 'buffer_view::buffer_view(const std::vector<C>&) [with C =
bool]'
```

But it would be great if we could generate a custom error message. You might think you could do it by putting a `static_assert` in the body:

```
  template<typename C>
  buffer_view(std::vector<C> const& v) :
    data(v.data()), size(v.size() * sizeof(C))
  {
    static_assert(!is_same_v<C, bool>,
      "Can't use std::vector<bool>. Try std::array instead.");
  }
```

Unfortunately, this `static_ assert` happens after the attempt to use `v.data()`, so the first error the developer sees is the incomprehensible one. We want our message to be the first error message, so we can quickly steer the developer in the right direction.

So we try again with a specialization that doesn't try to use the `v.data()` method, thereby avoiding the incomprehensible error message. We can then put our custom error message in the specialization.

```
template<>
buffer_view::buffer_view(std::vector<bool> const& v)
{
  static_assert(false, "blah blah blah");
}
```

However, this generates an error even if nobody tries to use the `std::vector<bool>` overload because the controlling expression of the `static_ assert` is not dependent upon the template type.

So let's make it dependent upon the template type.

```
template<typename C,
         std::enable_if_t<!std::is_same_v<C, bool>, int> = 0>
buffer_view(std::vector<C> const& v) :
  data(v.data()), size(v.size() * sizeof(C)) { }

template<typename C,
         std::enable_if_t<std::is_same_v<C, bool>, int> = 0>
buffer_view(std::vector<C> const& v)
{
  static_assert(!sizeof(C), "blah blah blah");
}
```

We create two templated constructors and let `enable_if` decide which one is active. For anything that isn't `bool`, we activate the first one, and we activate the second one only for `bool`.

The trick is that we now have a template type name "`C`" that we can use to generate a type-dependent always-false expression to put into the `static_ assert`. In this case, we can save a character and elide the `*` because we know that the type is exactly `bool`. We don't need to worry about the case where `C` is an incomplete type or `void`.

After I wrote this up, I discovered that Kenny Kerr came up with a simpler solution, which we'll look at next time.

Raymond Chen

Follow