Detecting in C++ whether a type is defined, part 2: Giving it a special name

devblogs.microsoft.com/oldnewthing/20190709-00

July 9, 2019

Raymond Chen

Warning to those who got here via a search engine: Don't use this version. Keep reading to the end of the series.

Last time, we detected whether a type was defined by setting up the unqualified name search order so that the name search would find the type if it were defined, or a fallback type if not. One problem with that technique was that the search had to be done from the specially-constructed namespace.

So let's fix that, and build on the result.

```
// awesome.h
namespace awesome
{
  // might or might not contain
  struct special { ... };
}

// your code
namespace detect::impl
{
  struct not_implemented {};
  using special = not_implemented;
}

namespace awesome::detect
{
  using namespace ::detect::impl;
  using special_maybe = special;
}
```

This time, I introduce a new type `special_maybe`. I did it inside the `awesome::detect` namespace, so the name `special` on the right hand side undergoes unqualified name lookup, like we described last time, and it will pick either the defined type `::awesome::special` or the fallback type `::detect::impl::special`. You can then use some new helpers:

```
namespace detect
{
  template<typename T>
  constexpr bool is_defined_v =
    !std::is_same_v<T, impl::not_implemented>;
}

void foo()
{
 if constexpr (detect::is_defined_v
                <awesome::detect::special_maybe>) {
   // do something now that we know "special" exists.
 }
}
```

This looks like it would work great, but it doesn't. Because inside the "do something now that we know `special` exists", you probably want to use `special`. But you can't, because `special` might not exist!

While it's true that `if constexpr` tells the compiler to discard the not-taken branch, the code in the not-taken branch must still be valid. If `special` is not defined, then the body of the `if constexpr` will contain references to the nonexistent entity `special`, so it will not compile. You could try using `special_maybe`, but that's just a dummy type, and it won't have the methods you want to call.

So we have to play a trick: Use the type without saying the type!

```
template<typename T, typename TLambda>
void call_me_maybe(TLambda&& lambda)
{
  if constexpr (is_defined_v<T>) {
    lambda(static_cast<T*>(nullptr));
  }
}
```

This helper function doesn't look all that useful. I mean, if the type exists, then we call the lambda. That just puts us back where we started, doesn't it? I mean, the lambda will need to use the type, which it can't do if the type doesn't exist.

Not quite. Because it lets us do this:

```
void foo(Source const& source)
{
  call_me_maybe<awesome::detect::special_maybe>(
    [&](auto* p)
    {
      using T = std::decay_t<decltype(*p)>;
      T::static_method();
      auto s = source.try_get<T>();
      if (s) s->something();
    });
}
```

What's going on?

The way C++ lambdas work is that a lambda becomes an anonymous type with an `operator()` method. For your typical lambda, the `operator()` is a const-qualified method whose prototype matches that of the lambda:

```
auto lambda1 = [](int v) -> void { ... };

// becomes

struct anonymous1
{
 auto operator()(int v) -> void const { ... };
};
auto lambda1 = anonymous1();
```

However, if the parameter list uses `auto`, then the `operator()` itself becomes a template function:

```
auto lambda2 = [](auto v) -> void { ... };

// becomes

struct anonymous2
{
 template<typename T>
 auto operator()(T v) -> void const { ... };
};
auto lambda2 = anonymous2();
```

Next, we take advantage of the fact that in a template function, entities that are dependent upon the template parameter are not resolved until the template is instantiated. In this case, the template function is the `operator()` of the lambda.

This means that our lambda body can do things that depend on the type of `p`, and the compiler can't validate that those things are meaningful until the template is instantiated, because it isn't until that time that the templated `operator()` is instantiated and the compiler knows what type it needs to use.

In other words, we use the incoming parameter *merely for its type information*. We extract the type of the pointed-to object and call it `T`. Then whenever we would normally say `special`, we just say `T`.[1]

And then we realize that we don't have to call it `T`. We can call it... `special`!

```
void foo(Source const& source)
{
  call_me_maybe<awesome::detect::special_maybe>(
    [&](auto* p)
    {
      using special = std::decay_t<decltype(*p)>;
      special::static_method();
      auto s = source.try_get<special>();
      if (s) s->something();
    });
}
```

With this little change, the code inside the lambda looks pretty much like the code you would have written all along, with the bonus feature that it's legal code even if `special` doesn't exist!

We're getting closer. Next time, we'll get rid of all this `maybe` nonsense.

[1] C++20 makes this a little easier by letting us get the type directly, rather than having to extract it from the parameter.

```
void foo(Source const& source)
{
  call_me_maybe<awesome::detect::special_maybe>(
    [&]<typename T>(T*)
    {
      T::static_method();
      auto s = source.try_get<T>();
      if (s) s->something();
    });
}
```

Or, using the second trick:

```
void foo(Source const& source)
{
  call_me_maybe<awesome::detect::special_maybe>(
    [&]<typename special>(special*)
    {
      special::static_method();
      auto s = source.try_get<special>();
      if (s) s->something();
    });
}
```

Raymond Chen

Follow