Detecting in C++ whether a type is defined, part 3: SFINAE and incomplete types

devblogs.microsoft.com/oldnewthing/20190710-00

July 10, 2019

Raymond Chen

Warning to those who got here via a search engine: This is part of a series. Keep reading to the end.

For the past few articles, I've been playing with the unqualified name lookup search order in order to detect whether a type exists in a particular namespace. I did this by defining the type in another namespace that has lower priority than the namespace that I'm probing, and then seeing which type comes out when I access the type with an unqualified name.

There are many problems with this technique. One is that it requires you to set up a `detect` namespace that contains a shadow version of every type you want to check. Another is that you need to inject a `detect` sub-namespace into every namespace you want to do detection in.

But it turns out there's another way, as long as you're willing to change one of the requirements. Instead of checking whether the type exists, check whether the type is *defined*, which in C++ language standard jargon means that you want the type to be complete.

```
template<typename, typename = void>
constexpr bool is_type_complete_v = false;

template<typename T>
constexpr bool is_type_complete_v
    <T, std::void_t<decltype(sizeof(T))>> = true;
```

A type must be complete in order to have the `sizeof` operator applied to it, so we use SFINAE to define `is_type_complete_v` as `true` provided the `sizeof` operator can be applied.[1]

I'm not sure if this is technically legal, but all the compilers I tried seemed to be okay with it. It does lead to weird effects like this:

```
struct s; // incomplete type
bool val1 = is_type_complete_v<s>; // false
struct s {}; // now it's complete
bool val2 = is_type_complete_v<s>; // true
```

The second phase of the trick takes advantage of the fact that you are permitted to refer to a class with the `struct` or `class` prefix. This prefix is usually redundant, but not always. It's also how you declare a forward reference.

The result is that you can say

```
is_type_complete_v<struct special>
```

to determine whether `struct special` has been defined.

1. If it has been defined, then the type exists and is complete.
2. If it has been declared but not defined, then the type exists and is incomplete.
3. If it has been neither declared nor defined, the act of writing `struct special` serves as a declaration! This puts us back into case 2 above, and the type exists and is incomplete.

So now our helper can be simplified to

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

and you would use it like this:

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

We are using the same tricks that we introduced last time: Using a generic lambda to defer resolving the type until the lambda is invoked, using `if constexpr` to avoid invoking the lambda if the type is not defined, and reintroducing the name of the type by deriving it from the dummy parameter.

There is a catch here: If you are probing for a type that is defined in a namespace that you imported via a `using` directive, and the type does not actually exist, then the `struct special` will declare an incomplete `struct special` in the *current* namespace.

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

// your code
namespace app
{
  using namespace awesome;

  void foo()
  {
    call_if_defined<struct special>([&](auto* p)
    {
       ...
    });
  }
}
```

If `special` is not defined, then the `struct special` in the `call_` `if_` `defined` will introduce an incomplete type called `app::` `special`.

Even more frustrating is that you cannot do this:

```
namespace app
{
  void foo()
  {
    call_if_defined<struct awesome::special>([&](auto* p)
    {
       ...
    });
  }
}
```

You cannot forward-declare a type into a non-current namespace. (For one thing, the result is ambiguous: Is this trying to forward-declare `::awesome` `::special`, or is it trying to forward-declare `::app` `::awesome` `::special`?)

And of course there's the annoyance of having to type the word `struct`.

We can trade three annoyances for one. We'll continue the investigation next time.

[1] The `std:: void_t` template type is `void` regardless of its template parameters. The template type exists specifically for SFINAE, so that the overload is removed from consideration if the template parameter ends up being invalid. In this case, what it means is that if `sizeof(T)` is invalid (which is the case if `T` is an incomplete type), then the `std:: void_t` fails substitution, and the rule disappears.

You can think of it as `std:: void_ if_ valid_t`.

Raymond Chen

Follow