Detecting in C++ whether a type is defined, part 4: Predeclaring things you want to probe

devblogs.microsoft.com/oldnewthing/20190711-00

July 11, 2019

Raymond Chen

Note to those who got here via a search engine: This is the last part of the core portion of the series, but there are still parts to come. For the impatient, here's the stuff to copy-paste:

```
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;

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

Last time, we used SFINAE to detect whether a type had a definition, and we used that in combination with `if constexpr` and generic lambdas so that code could use the type if it is defined, while still being accepted by the compiler (and being discarded) if the type is not defined.

However, our usage had a few issues, some minor annoyance, some more frustrating.

- You had to say `struct` all the time.
- If the type didn't exist, the act of naming it caused the type to be injected into the *current* namespace, not the namespace you expected the type to be in.
- You must use the `struct` technique with an unqualified name. You can't use it to probe a type that you didn't import into the current namespace.

We can fix all three of the problems with a single solution: Predeclare the type in the desired namespace.

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

// your code

namespace awesome
{
  // ensure declarations for types we
  // conditionalize on.
  struct special;
}
```

Once you've done this, you don't need to say `struct` because the struct definitely has been declared. Your use of it as a template type parameter in `call_ if_ defined` will not create a new declaration, because it has already been declared. And since it has been declared, you can access it via its unqualified name, its full namespace name, or anything in between. Also a type alias or dependent type. (Sorry, those aren't in between.)

```
namespace app
{
  void foo()
  {
    call_if_defined<awesome::special>([&](auto* p)
    {
       // this code is compiled only if "awesome::special"
       // is defined. Create a local name for "special"
       // by inferring it from the dummy parameter.
       using special = std::decay_t<decltype(*p)>;

       // You can now use the local name "special" to access
       // the features of "awesome::special".
       special::do_something();
    });
  }
}
```

For those who have been following the series from the beginning, you may have noticed that the `call_ if_ defined` method is not quite the same as the version we wrote earlier. The new version supports multiple type parameters and calls the lambda only if all of the types are defined.

Let's take a closer look:

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

The double-parentheses in the `if constexpr ((...))` look weird, but they're required. The outer parentheses are required by the `if constexpr` statement, and the inner parentheses are required by the fold expression. The fold expression expands to

```
  if constexpr (
    (is_complete_type_v<T1> &&
     is_complete_type_v<T2> &&
     ...
     is_complete_type_v<Tn>))
```

The invoke of the lambda uses a parameter pack expansion:

```
    lambda(static_cast<T*>(nullptr)...);
```

This expands to

```
    lambda(static_cast<T1*>(nullptr),
           static_cast<T2*>(nullptr),
           ...,
           static_cast<Tn*>(nullptr));
```

where the `static_ cast<T*>(nullptr)` is repeated once for each type.

As I noted earlier, we can use this function to call a lambda if *all* the types are defined:

```
void foo(Source const& source)
{
  call_if_defined<special, magic>(
    [&](auto* p1, auto* p2)
    {
      using special = std::decay_t<decltype(*p1)>;
      using magic = std::decay_t<decltype(*p2)>;

      auto s = source.try_get<special>();
      if (s) magic::add_magic(s);
    });
}
```

C++20 allows you to write this as

```
void foo(Source const& source)
{
  call_if_defined<special, magic>(
    [&]<typename special, typename magic>
    (special*, magic*)
    {
      auto s = source.try_get<special>();
      if (s) magic::add_magic(s);
    });
}
```

which lets you name the template type, thereby saving you the trouble of having to re-derive it by playing `std::decay_t` games.

Next time, we'll use this as a springboard and extend the pattern.

Raymond Chen

Follow