# Detecting in C++ whether a type is defined, part 5: Augmenting the basic pattern

devblogs.microsoft.com/oldnewthing/20190712-00

July 12, 2019

Raymond Chen

As I noted at the start of this series, React Native for Windows has to deal with two axes of agility.

- It needs to compile successfully across different versions of the Windows SDK.
- It needs to run successfully across different versions of Windows, while taking advantage of new features if available.

The second is a common scenario, and the typical way of solving it is to probe for the desired feature and use it if is available.

The first is less common. Usually, you control the version of the Windows SDK that your project consumes. The act of ingesting a new version is often considered a big deal.

Libraries like React Native for Windows have to deal with this problem, however, because the project that consumes them gets to pick the Windows SDK version, and the library has to cope with whatever it's given. In such cases, a feature is used if it is available both in the Windows SDK that the project was compiled with, as well as in the version of Windows that the project is running on.

Not controlling the version of the Windows SDK means that you need to infer at compile time what features are available. The `call_if_defined` helper fits the bill, but we can go even further to make it even more convenient.

For example, consider the `Windows.UI.Xaml.UIElement` class. Support for the `StartBringIntoView` method was added in interface `IUIElement5`, which arrived in the Creators Update.[1] You could write this:

```
void BringIntoViewIfPossible(UIElement const& e)
{
  auto el5 = e.try_as<IUIElement5>();
  if (el5) {
    el5.StartBringIntoView();
  }
}
```

This works great provided the host project is compiling the library with a version of the Windows SDK that contains a definition for `IUIElement5` in the first place.

Boom, textbook case for `call_if_defined`.

```
namespace winrt::Windows::UI::Xaml
{
  struct IUIElement5;
}

using namespace winrt::Windows::UI::Xaml;

void BringIntoViewIfPossible(UIElement const& e)
{
  call_if_defined<IUIElement5>([&](auto* p) {
    using IUIElement5 = std::decay_t<decltype(*p)>;

    auto el5 = e.try_as<IUIElement5>();
    if (el5) {
      el5.StartBringIntoView();
    }
  });
}
```

This type of "probe for interface support" is a common scenario when writing version-agile code, so we could make a specialized version of `call_if_defined` to simplify the scenario.

```
template<typename T, typename TLambda>
void call_if_supported(IInspectable const& source,
                       TLambda&& lambda)
{
  if constexpr (is_complete_type_v<T>) {
    auto t = source.try_as<T>();
    if (t) lambda(std::move(t));
  }
}
```

This version calls the lambda if the specified type is (1) supported in the SDK being consumed, and (2) supported at runtime by the version of Windows that the code is running on. You would use it like this:

```
void BringIntoViewIfPossible(UIElement const& e)
{
  call_if_supported<IUIElement5>(e, [&](auto&& el5) {
    el5.StartBringIntoView();
  });
}
```

The idea here is that `call_if_supported` checks for both compile-time and runtime support, and if both tests pass, it calls the lambda, passing an *actual object* rather than a dummy parameter.

Passing an actual object means that the lambda doesn't need to re-infer the type. It can just use the passed-in object directly.

This lets you write code that is conditional both on compile-time and runtime feature detection.

A case that you might find useful even if you don't use C++/WinRT is declaring a variable or member of a particular type, provided it exists.

```
struct empty {};
template<typename T, typename = void>
struct type_if_defined
{
    using type = empty;
};

template<typename T>
struct type_if_defined<T, std::void_t<decltype(sizeof(T))>>
{
    using type = T;
};

template<typename T>
using type_if_defined = typename type_or_empty<T>::type;
```

You can declare a variable or member of type `type_ if_ defined`, and it will either contain the thing (if it is defined), or it will be an empty struct. You could combine this with `call_ if_ defined` so you have a place to put the thing-that-might-not-be-defined across two calls.

```
// If "special" is available, preserve it.

type_if_defined<special> s;

call_if_defined<special>([&](auto *p)
{
  using special = std::decay_t<decltype(*p)>;

  s = special::get_current();
});

do_something();

call_if_defined<special>([&](auto *p)
{
  using special = std::decay_t<decltype(*p)>;

  special::set_current(s);
});
```

[1] Who names these things?

Raymond Chen

**Follow**