Using C++ type aliasing to avoid the ODR problem with conditional compilation, part 1

devblogs.microsoft.com/oldnewthing/20250501-00/?p=111134

May 1, 2025

Some time ago, I discussed the C++ concept of *ill-formed no diagnostic required* (IFNDR). A common accidental source of this is violating the *One Definition Rule* (ODR) by defining a class or function differently based on compile-time switches.

```
// widget.h

struct Widget
{
    Widget();

    void SetName(std::string const& name);

    ⟦ more stuff ⟧

#ifdef EXTRA_WIDGET_DEBUGGING
    Logger m_logger;

    void Log(std::string const& message) {
        m_logger.log(message);
    }
#else
    void Log(std::string const& message) {
        // no extra logging
    }
#endif

    std::string m_name;
};
```

If one .cpp file is compiled with extra widget debugging enabled, but another is compiled with extra widget debugging disabled, and they are linked together, then you have a One Definition Rule violation because the `Widget` structure and the `Widget::Log` method have conflicting definitions.

But all is not lost.

Type aliases are not subject to the One Definition Rule!

It’s okay to have a type alias with different definitions in different translation units because a type alias is just a a way to introduce an alternate name for an existing type; it does not introduce a new type.

```
// widget.h

template<bool debug>
struct WidgetT
{
    WidgetT();

    ⟦ more stuff ⟧

    [[msvc::no_unique_address]]
    [[no_unique_address]]
    std::conditional_t<debug, Logger, std::monostate> m_logger;

    void Log(std::string const& message) {
        if constexpr (debug) {
            m_logger.log(message);
        }
    }

    std::string m_name;
};

extern template struct WidgetT<false>;
extern template struct WidgetT<true>;

#ifdef EXTRA_WIDGET_DEBUGGING
using Widget = WidgetT<true>;
#else
using Widget = WidgetT<false>;
#endif

// widget.cpp
#include "widget.h"

template<bool debug>
void Widget<debug>::Widget()
{
    ⟦ constructor stuff ⟧
}

template<bool debug>
void Widget<debug>::SetName(std::string const& name)
{
    m_name = name;
    Log("Name changed");
    Log(name);
}

template struct WidgetT<false>;
template struct WidgetT<true>;
```

There are two versions of `WidgetT`. The `WidgetT<true>` is the debugging version, and the `WidgetT<false>` is the non-debugging version. In the debugging version, there is a `Logger` member object, and in the non-debugging version, we have a `std::monostate`, which is a dummy object that does nothing. We also mark the object as `no_unique_address` to tell the compiler that it's okay to collapse the empty object to nothing, so that it disappears entirely when not debugging.[1]

All of the methods are implemented as templates, which is a bit annoying, but it's just a bunch of boilerplate repetition for each method you want to implement.

Since the implementations are not in the header file, we have to instantiate the templates explicitly to trigger the code generation. There are two versions of the template, and we instantiate them both.

Meanwhile, when clients use the `widget.h` header file, they can pick what they want the name `Widget` to refer to. For example, if they have debugging enabled, then it is an alias for `WidgetT<true>`, so when they create a `Widget`, they are creating a `WidgetT<true>`, and when they call a method on it, they are calling a method of `WidgetT<true>`, and when they pass it to another function, they are passing a `WidgetT<true>`.

Meanwhile, another client has debugging disabled, then all of its operations on a `Widget` are really happening with a `WidgetT<true>`. Even though each client uses the name `Widget` to refer to a different thing, there is no conflict here because the compiler cares about the actual type, not any nickname you may have given it.

```
// client1.h

#include <widget.h>

void Client1DoSomething(Widget const& widget);
```

If Client1 is compiled with debugging enabled, its `client1.cpp` will implement `void Client1DoSomething(Widget<true> const& widget)`. But if Client2 is compiled with debugging disabled, its `client2.cpp` will try to call `void Client1DoSomething( Widget<false> const& widget)`. Since there is no definition for that function, you get a linker error.

If the two clients (which disagree on what `Widget` refers to) try to talk to each other through a `Widget`, you will get a linker error because one side is trying to call a function that takes a `WidgetT<true>`, but the other side implemented a function that takes a `WidgetT<false>`.

There is a tricky bit if one or the other client exposes a class that uses a `Widget`.

```
// client1.h

#include <widget.h>

struct Client1
{
    Widget m_widget;
};
```

Then Client1 (with debugging enabled) thinks that the `Client1` structure uses a `Widget<true>`, but Client2 (with debugging disabled) thinks that the the `Client1` structure uses a `Widget<false>`. This is an ODR violation, and depending on how unlucky you are, it may go undetected.

Similarly, there is an ODR violation with the global function if the presence of the `Widget` is in something that doesn't participate in overload resolution, like a return value.

```
// client1.h

#include <widget.h>

Widget Client1MakeWidget();
```

Okay, so I got stuck.

But I think can still save this. We'll do that next time.

Bonus reading: What is `__wchar_t` (with the leading double underscores) and why am I getting errors about it? uses a similar technique to deal with multiple possible definitions of `wchar_t`.

[1] The `no_unique_address` attribute also tells the compiler that if there is any trail padding in the `Logger` object, it is allowed to put other `WidgetT` members inside the trail padding.