How can I prevent myself from using a parameter after I’ve extracted all value from it?

devblogs.microsoft.com/oldnewthing/20230915-00

September 15, 2023

Raymond Chen

Suppose you have a function that takes a parameter that you want to transform in some way, and you want to require that all future access to the parameter be done through the transformed version. One example is a wrapper class that does logging.[1]

```
struct WidgetRefWrapper
{
    WidgetRefWrapper(
        Widget& widget,
        Logger& logger) :
    m_widget(widget), m_logger(logger) {}

    void Toggle() try
    {
        m_logger.Log("Toggling the widget");
        m_widget.Toggle();
        m_logger.Log("Toggled the widget");
    } catch (...) {
        m_logger.Log("Exception while toggling the widget");
        throw;
    }

private:
    Widget& m_widget;
    Logger& m_logger;
};

void DoSomething(Widget& widget)
{
    Logger& logger = GetCurrentLogger();
    WidgetWrapper wrapper(widget, logger);

    // Do not use the widget directly!
    // Always use the wrapper!

    if (needs_toggling) {
        wrapper.Toggle();
    }
}
```

You want that “Do not use the widget directly!” comment to have some teeth. Can you “poison” the `widget` parameter so it cannot be used any more?

One idea is to split the method into two. The outer function does the preliminary wrapping, and the worker function does the bulk of the work.

```
void DoSomething(Widget& widget)
{
    Logger& logger = GetCurrentLogger();
    WidgetWrapper wrapper(widget, logger);

    DoSomethingWorker(wrapper, logger):
}

void DoSomethingWorker(
    WidgetRefWrapper& wrapper,
    Logger& logger)
{
    if (needs_toggling) {
        wrapper.Toggle();
    }
}
```

Since the raw `widget` is never passed to the worker function, there is no way to access it. If you type `widget`, you get an undefined identifier.

On the other hand, it also means that you have to pass all of the work you've done so far to the worker function. In this case, we pass the `logger`.

Also, somebody might see that you split the work into two functions and say, “What's the point of a function that is called in only one place? I can just inline it!” and now you're back where you started.

We can reuse the clever hack / shameless abuse of the language known as `hide_name`.

```
void DoSomething(Widget& widget)
{
    Logger& logger = GetCurrentLogger();
    WidgetWrapper wrapper(widget, logger);

    // From now on, all access must be done through the wrapper.
    hide_name widget;

    if (needs_toggling) {
        wrapper.Toggle();
    }
}
```

Previously in “clever hack or shameless abuse of the language”: Bringing thread switching tasks to C#.

[1] Another case would be code that takes the inbound parameter and looks it up in some data structure to find a partner object, and we want all future operations to occur on the partner.