If you're going to crash on an unhandled exception, you may as well do it sooner rather than later

devblogs.microsoft.com/oldnewthing/20231120-00

November 20, 2023

Raymond Chen

Some time ago, we looked at the case of the invalid argument exception from a method that takes no arguments, and the investigation took us through a stowed exception, and then back out the other side. That was a lot of work to dig out the root cause. Can we avoid this problem in the future?

Recall that the problem originated in this lambda:

```
m_closingRevoker = flyout.Closing(winrt::auto_revoke,
    [](auto&& sender, auto&&) {
    if (auto flyout = sender.try_as<xaml::CommandBarFlyout>()) {
        auto popups =
xaml::VisualTreeHelper::GetOpenPopupsForXamlRoot(flyout.XamlRoot());
        〚…〛
    }
});
```

The call to `GetOpenPopupsForXamlRoot` threw an invalid argument exception, which then escaped the lambda and was propagated all the way back to an unrelated function call.

```
void MyPanel::CancelCurrentFlyout()
{
    〚…〛

    m_menu.Hide();

    〚…〛
}
```

This exception then propagated up the call stack to the caller of `MyPanel::CancelCurrentFlyout`, which is `MyPanel::IsReadyForNewContextMenu`, and then propagated further to `MyPanel::DoContextMenu`, and that's where the process finally terminated, because `MyPanel::DoContextMenu` was marked as `noexcept`.

```
void ContextMenuPresenter::DoContextMenu() noexcept
{
    ⟦…⟧
}
```

The `noexcept` specifier means that any attempt to propagate an exception out of the function will result in `std::terminate`.

The problem here is that the source of the exception was so far away from the place the exception was detected, and we had to do a lot of detective work to trace the exception back to its origin.

We could have avoided all this hassle if we had marked the event handler as `noexcept`:

```
m_closingRevoker = flyout.Closing(winrt::auto_revoke,
    [](auto&& sender, auto&&) noexcept {
    if (auto flyout = sender.try_as<xaml::CommandBarFlyout>()) {
        auto popups =
xaml::VisualTreeHelper::GetOpenPopupsForXamlRoot(flyout.XamlRoot());
        ⟦…⟧
    }
});
```

With the `noexcept` specifier on the lambda, an exception from `GetOpenPopupsForXamlRoot` would have terminated the process once it tried to propagate out of the lambda, before XAML could catch it. That would have given us a crash stack trace that directly pinpointed the problem.

We learned this lesson some time ago: If there is no difference between two options, choose the one that is easier to debug.

In general, you probably don't want your event handlers to allow exceptions to propagate. If something goes wrong in the event handler, you're either going to deal with it right away or never at all. And if you're never going to deal with it at all, you may as well fail immediately so the problem is easier to diagnose in a post-mortem.

Bonus chatter: Another reason to mark your event handler as `noexcept` is to avoid accidentally making a secret signal.

```
m_closingRevoker = flyout.Closing(winrt::auto_revoke,
    [weak = get_weak()](auto&& sender, auto&&) {
        if (auto strongThis = weak.get()) {
            strongThis->m_widget.Stop();
            if (strongThis->m_buddy) {
                strongThis->m_buddy.NotifyStopped();
            }
        }
    }
});
```

If `m_buddy` is a reference to an out-of-process object that has crashed, the `NotifyStopped` will fail with something like `RPC_E_SERVER_DIED`, which will propagate out of the lambda and be interpreted as the secret signal for "The object no longer exists. Stop calling the event handler." Your event handler is "helpfully" unregistered, and a few weeks later, you end up studying bugs where the widgets are still running even though the flyout is closed. If you're lucky, you trace it back to this "server died" exception that propagated out of the lambda by mistake.

Putting a `noexcept` on the lambda forces the process to terminate if an exception occurs, which means that if the buddy crashes, your process will crash the next time it tries to access the buddy. And if that's not what you want, you can catch the exception and apply domain-specific recovery.

```
m_closingRevoker = flyout.Closing(winrt::auto_revoke,
    [weak = get_weak()](auto&& sender, auto&&) noexcept {
        if (auto strongThis = weak.get()) {
            strongThis->m_widget.Stop();
            if (strongThis->m_buddy) {
                try {
                    strongThis->m_buddy.NotifyStopped();
                } catch (...) {
                    // Throw away the broken buddy.
                    strongThis->m_buddy = nullptr;
                }
            }
        }
    }
});
```