C++/WinRT performance trap: Switching to Windows Runtime too soon

devblogs.microsoft.com/oldnewthing/20240301-00

March 1, 2024

Raymond Chen

The Windows Runtime contains equivalents to C++ vectors and maps, namely `IVector` and `IMap`. If you have a choice, stick with the C++ versions, and if you have to produce a Windows Runtime version, delay the conversion as long as possible.

The reason is that the Windows Runtime types are all virtual interfaces, which means vtable calls for all the methods which cannot be inlined or optimized. Whereas C++ library types have a richer set of available methods to allow you to write simpler code, and since they are written in C++ itself, the compiler can perform optimizations that aren't available to virtual methods.

For example, suppose you have a method that has to return an `IVector<Widget>`. I see a lot of people write it like this:

```
winrt::IVector<Widget> GetWidgets()
{
    winrt::IVector<Widget> widgets =
        winrt::multi_threaded_vector<Widget>();

    widgets.Append(GetFirstWidget());
    widgets.Append(GetSecondWidget());
    widgets.Append(GetThirdWidget());

    return widgets;
}
```

This creates an empty Windows Runtime vector and adds three widgets to it. Each of those `Append` calls is a virtual method call that the compiler may not be able to devirtualize, and since you have a multi-threaded vector, it's going to do some locking to protect against concurrent access, even though no concurrency is possible here because the vector hasn't been shared with anyone else yet.

More efficient would be

```
winrt::IVector<Widget> GetWidgets()
{
    std::vector<Widget> widgets;
    widgets.emplace_back(GetFirstWidget());
    widgets.emplace_back(GetSecondWidget());
    widgets.emplace_back(GetThirdWidget());

    return winrt::multi_threaded_vector<Widget>(
        std::move(widgets));
}
```

Creating the vector is done in the C++ world, where the compiler can do all sorts of clever optimizations. Only at the end is this vector converted to a Windows Runtime vector to be returned to the caller.

And now that you've gotten the vector-building in C++, you can take advantage of additional C++ features, like reservation to avoid reallocation:

```
winrt::IVector<Widget> GetWidgets()
{
    std::vector<Widget> widgets;
    widgets.reserve(3);

    widgets.emplace_back(GetFirstWidget());
    widgets.emplace_back(GetSecondWidget());
    widgets.emplace_back(GetThirdWidget());

    return winrt::multi_threaded_vector<Widget>(
        std::move(widgets));
}
```

Or construction from an initializer-list.

```
winrt::IVector<Widget> GetWidgets()
{
    std::vector<Widget> widgets{
        GetFirstWidget(),
        GetSecondWidget(),
        GetThirdWidget(),
    };

    return winrt::multi_threaded_vector(
        std::move(widgets));
}
```

Bonus reading: On the virtues of the trailing comma.

Once we construct the vector from an initializer-list, we can let CTAD save us some more typing:

```
winrt::IVector<Widget> GetWidgets()
{
    std::vector widgets{
        GetFirstWidget(),
        GetSecondWidget(),
        GetThirdWidget(),
    };

    return winrt::multi_threaded_vector(
        std::move(widgets));
}
```

And finally, we can just in-place construct the vector in the parameter list for `multi_threaded_vector`.

```
winrt::IVector<Widget> GetWidgets()
{
    return winrt::multi_threaded_vector(
        std::vector{
            GetFirstWidget(),
            GetSecondWidget(),
            GetThirdWidget(),
        });
}
```

A similar shortcut applies to maps.

```
winrt::IMap<winrt::hstring, int32_t> GetCounts()
{
    return winrt::multi_threaded_map(
        std::unordered_map<winrt::hstring, int32_t>{
            { L"triangles", GetTriangleCount() },
            { L"rectangles", GetCircleCount() },
            { L"circles", GetCircleCount() },
        });
}
```

Even if you can't reduce your function to a one-liner, it's more efficient (and probably a lot easier) to collect the contents into a C++ vector or map (or unordered map) and then wrap it inside a Windows Runtime interface as a final step.

```
winrt::IVector<Widget> GetWidgets()
{
    std::vector<Widget> widgets;

    // Collect all the widgets from enabled doodads
    for (auto&& doodad : m_doodads) {
        if (doodad.m_enabled) {
            widgets.insert(widgets.end(),
                doodad.m_widgets.begin(),
                doodad.m_widgets.end());
        }
    }

    // Sort by population descending
    std::sort(widgets.begin(), widgets.end(),
        [](Widget const& left, Widget const& right) {
            return left.Population() > right.Population();
        });

    return winrt::multi_threaded_vector(
        std::move(widgets));
}
```