# How to clone a Windows Runtime map in the face of possible concurrent modification, part 2

devblogs.microsoft.com/oldnewthing/20230720-00

July 20, 2023

Raymond Chen

Although we've figured how to clone a Windows Runtime map in the face of possible concurrent modification, we're still fiddling around with the best design for the function that does the cloning.

Last time, we wrote a one-shot function that does everything, but maybe the answer is to not try to jam everything into one function.

We can break things into two functions, one for using a `std::map` and one for using a `std::unordered_map`. (This also avoids problems if the user tries to call the original function with something like `clone_as_map<std::vector<int>>`.)

```
template<typename M>
auto clone_as_kvp_vector(M const& m)
{
    using KVP = decltype(m.First().Current());
    std::vector<KVP> pairs;
    uint32_t expected;
    uint32_t actual;
    do {
        expected = m.Size();
        pairs.resize(expected + 1);
        try {
            actual = m.First().GetMany(pairs);
        }
        catch (winrt::hresult_changed_state const&) {
            continue;
        }
    } while (actual > expected);
    pairs.resize(actual);
    return pairs;
}
```

This worker function does the real work of cloning the Windows Runtime map into a C++ vector of `IKeyValuePair` objects.

The next step is to write some helpers to cut down on the repetitive typing that happens when you do type deduction from the future:

```
template<typename Override, typename Fallback>
using override_or_fallback_t =
    std::conditional_t<
        std::is_same_v<Override, void>,
        Fallback,
        Override>;

template<typename M,
    typename KeyOverride,
    typename ValueOverride>
struct inferred_runtime_map_traits
{
    using KVP = decltype(std::declval<M>().First().Current());

    using Key = override_or_fallback_t<
        KeyOverride,
        decltype(KVP().Key())>;
    using Value = override_or_fallback_t<
        ValueOverride,
        decltype(KVP().Value())>;
};

template<typename M,
    typename KeyOverride,
    typename ValueOverride,
    typename CompareOverride>
struct inferred_map_traits
    {
    using base = inferred_runtime_map_traits
        <M, KeyOverride, ValueOverride>;
    using Key = typename base::Key;
    using Value = typename base::Value;
    using Compare = override_or_fallback_t<
        CompareOverride, std::less<Key>>;
    using type = std::map
        <Key, Value, Compare>;
};

template<typename M,
    typename KeyOverride,
    typename ValueOverride,
    typename HashOverride,
    typename KeyEqualOverride>
struct inferred_unordered_map_traits
{
    using base = inferred_runtime_map_traits
        <M, KeyOverride, ValueOverride>;
    using Key = typename base::Key;
    using Value = typename base::Value;
    using Hash = override_or_fallback_t<
        HashOverride, std::hash<Key>>;
    using KeyEqual = override_or_fallback_t<
```

```
        KeyEqualOverride, std::equal_to<Key>>;
    using type = std::unordered_map
        <Key, Value, Hash, KeyEqual>;
};
```

We start with `override_or_fallback_t`, a type alias that encapsulates the “use the type provided if it is not `void`; otherwise use a fallback type.” We use this to build up an `inferred_runtime_map_traits` which plucks out the `KVP`, `Key`, and `Value` from a Windows Runtime map. We also build an `inferred_map_traits` `inferred_unordered_map_traits` which does the same for the additional template type parameters of a `std::map` and `std::unordered_map`.

We can use these traits types to save ourselves typing in the next few functions.

```
template<typename Key = void,
    typename Value = void,
    typename Compare = void,
    typename M,
    typename Traits = inferred_map_traits
        <M, Key, Value, Compare>>
auto clone_as_map(M const& m,
    typename Traits::Compare const& compare = {})
{
    auto pairs = clone_as_kvp_vector(m);
    typename Traits::type map(compare);
    for (auto&& pair : pairs) {
        map.emplace_hint(
            map.end(), pair.Key(), pair.Value());
    }
    return map;
}

template<typename Key = void,
    typename Value = void,
    typename Hash = void,
    typename KeyEqual = void,
    typename M,
    typename Traits = inferred_unordered_map_traits
        <M, Key, Value, Hash, KeyEqual>>
auto clone_as_unordered_map(M const& m,
    typename Traits::Hash const& hash = {},
    typename Traits::KeyEqual const& equal = {})
{
    auto pairs = clone_as_kvp_vector(m);
    typename Traits::type map(
        static_cast<uint32_t>(pairs.size()),
        hash, equal);
    for (auto&& pair : pairs) {
        map.emplace(pair.Key(), pair.Value());
    }
    return map;
}

template<typename Key = void,
    typename Value = void,
    typename Compare = void,
    typename M,
    typename Traits = inferred_map_traits
        <M, Key, Value, Compare>>
auto CloneMap(M const& m,
    typename Traits::Compare const& compare = {})
{
    return winrt::multi_threaded_map(
        clone_as_map<Key, Value, Compare>
        (m, compare));
}
```

```
template<typename Key = void,
    typename Value = void,
    typename Hash = void,
    typename KeyEqual = void,
    typename M,
    typename Traits = inferred_unordered_map_traits
        <M, Key, Value, Hash, KeyEqual>>
auto CloneUnorderedMap(M const& m,
    typename Traits::Hash const& hash = {},
    typename Traits::KeyEqual const& equal = {})
{
    return winrt::multi_threaded_map(
        clone_as_unordered_map<Key, Value, Hash, KeyEqual>
        (m, hash, equal));
}
```

We front-load the template type parameters that belong to the map or unordered map, so that you can write something that looks almost natural.

```
// Create a std::unordered_map from strings to strings
// but where the string keys are treated as case-insensitive.
auto clone = clone_unordered_map<
    winrt::hstring, winrt::hstring,
    case_insensitive_string_hash,
    case_insensitive_string_equal>(original);
```

Most of this wackiness was just C++ template metaprogramming nonsense.

Notice that we used `std::map::emplace_hint` to optimize for the case that the keys are returned in sorted order.

Next time, we'll port this to C++/CX.