Making C++ primitive types meaningfully movable when they have sentinel values

devblogs.microsoft.com/oldnewthing/20230116-00

January 16, 2023

Raymond Chen

C++ primitive types do not have special semantics for move constructor or move assignment. Their move operations are just copies. But what if you really want them to move, say, because they have a sentinel value that represents an "empty" state.

```
template<typename T, T empty_value>
struct movable_primitive
{
    T value = empty_value;
    movable_primitive() = default;
    movable_primitive(T const& init) : value(init) {}
    movable_primitive(movable_primitive const&) = default;
    movable_primitive(movable_primitive&& other) :
        value(std::exchange(other.value, empty_value)) {}
    movable_primitive& operator=(movable_primitive const&) = default;
    movable_primitive& operator=(movable_primitive&& other)
    {
        value = std::exchange(other.value, empty_value);
        return *this;
    }

    operator T() { return value; }
};
```

The idea here is that when the primitive is copied, we copy the value, but when the primitive is moved, the old primitive's value reverts to the `empty_value`.

This class isn't really useful on its own, since you can probably manage the state yourself without too much difficulty, but it comes in handy when it is part of a larger class, since it allows you to use the Rule of Zero.

For example, you might have one scalar member that records information about another movable member. If that other member is moved, then the scalar should be moved with it.

```
enum class flavor
{
    none,
    vanilla,
    chocolate,
};

struct vector_with_flavor
{
    std::vector<int> v;
    movable_primitive<flavor, flavor::none> flav;
};
```

If you copy this structure, the vector and its flavor are copied. If you move this structure, the vector is emptied, and the flavor reverts to `none`.

If you use the Windows Implementation Libraries, you can sort of build this out of `wil::unique_struct`:

```
template<typename T>
struct primitive_wrapper
{
    T value;
    primitive_wrapper() = default;
    primitive_wrapper(T const& initial) : value(initial) {}
    static void close(primitive_wrapper*) noexcept {}
    template<T empty_value> static void init(primitive_wrapper* p)
        { p->value = empty_value; }
    operator T() { return value; }
};

template<typename T, T empty_value>
using movable_primitive = wil::unique_struct<
    primitive_wrapper<T>,
    decltype(primitive_wrapper<T>::close),
            &primitive_wrapper<T>::close,
    decltype(primitive_wrapper<T>::template init<empty_value>),
            &primitive_wrapper<T>::template init<empty_value>>;
```

WIL's `unique_struct` comes with other features like `release()` to return the value and reset to the empty state. However, setting a new value is a bit more cumbersome.

```
movable_primitive<int, 0> v;
v = 42; // for original movable_primitive
v.reset(42); // for wil-based movable_primitive
```

Raymond Chen

Follow