Should I pay attention to the warning that I'm std::move'ing from a trivial type? Part 2

devblogs.microsoft.com/oldnewthing/20220513-00

May 13, 2022

Raymond Chen

Last time, we looked at motivations for `std::move`'ing from a trivial type. Our investigation looked at the problem through the eyes of the object moved *from*, but there's another way to look at the problem, and that's from the point of view of the object being moved *to*.

```
struct widget_id
{
    int value;
};

void destroy_id(widget_id&& id)
{
    auto value = std::exchange(id.value, 0);
}
```

The intention of the `destroy_id` function is to destroy the `id`. Now, the widget ID is just an integer, so you technically don't "destroy" it, but the function accepts the `widget_id` by rvalue reference so it can set the `value` of the source to zero, thereby reducing the likelihood that the caller will try to use the `id` later.

```
widget_id id = get_id();
destroy_id(std::move(id));

// accidentally use the id
activate_id(id); // blatantly invalid id of 0
```

Accepting an rvalue reference helps with two problems: First of all, if the ID is held in a variable, the caller must perform an explicit `std::move`, which tickles the C++ brain cells into recognizing that "This `id` is consumed by the `destroy_id` function and is therefore not useful any more."

Second of all, it means that if the caller messes up and tries to use the `id` after it was moved from, the `value` it contains is guaranteed invalid, instead of possibly using an already-destroyed ID which might by coincidence have been reassigned to another new widget in the

meantime. That converts a bug that occurs only if you hit just the right race condition (harder to debug) into a bug that occurs every time guaranteed (easier to debug).

You can even use the “accept an rvalue when destroying or taking ownership” trick for primitive types like pointers.

```
template<typename T>
struct uniquer_ptr
{
    T* m_ptr = nullptr;

    uniquer_ptr(T*&& ptr = nullptr)
        : m_ptr(ptr)
    {
        ptr = nullptr;
    }

    ...
    void reset(T*&& ptr = nullptr)
    {
        T* old_ptr = m_ptr;
        m_ptr = ptr;
        ptr = nullptr;

        if (old_ptr != nullptr) get_deleter()(old_ptr);
    }
};

// or, if you like one-liners...

template<typename T>
struct uniquer_ptr
{
    T* m_ptr = nullptr;

    uniquer_ptr(T*&& ptr = nullptr)
        : m_ptr(std::exchange(ptr, nullprt))
    {
    }

    ...
    void reset(T*&& ptr = nullptr)
    {
        T* old_ptr = std::exchange(m_ptr, std::exchange(ptr, nullptr));
        if (old_ptr != nullptr) get_deleter()(old_ptr);
    }
};
```

This variant of `std::unique_ptr` accepts the raw pointer by rvalue reference so that it can null it out, thereby emphasizing that it has taken ownership.

```
// two-timing widget: This compiles
// and runs, but you crash when ptr
// destructs.
Widget* widget = new Widget();
auto ptr = std::unique_ptr(widget);
auto ptr2 = std::unique_ptr(widget);
```

Accepting the pointer by rvalue reference reduces the likelihood of two-timing:

```
// doesn't compile
Widget* widget = new Widget();
auto ptr = std::unique_ptr(widget);
auto ptr2 = std::unique_ptr(widget);

// compiles but looks suspicious
Widget* widget = new Widget();
auto ptr = uniquer_ptr(std::move(widget));
auto ptr2 = uniquer_ptr(std::move(widget));
```

The revised version looks suspicious because you are using an object (the `widget`) after it has been `std::move`'d. At run time, the `widget` is set to `nullptr` by the first `uniquer_ptr` constructor, so `ptr2` constructs from `nullptr` and is consequently empty. There is no crash at runtime. (Though you might scratch your head if you expected `ptr2` to be non-empty.)

Is this a good use for rvalue references to primitive types? I'm not sure.

Bonus chatter: Many years ago, the Windows division was given an experimental version[1] of the Visual C++ compiler which treated the argument of the `delete` statement as an lvalue reference if possible, and nulled out the value as part of the deletion.

```
int* p = new int();
delete p;
// p is now nullptr!
```

The theory behind this change was that the pointer `p` is unusable after being deleted, so the compiler may as well turn it into a provably unusable value, so you can't use it by mistake.

Unfortunately, this silent breaking change resulted in some runtime crashes because there were some calling patterns that relied on the old value remaining unchanged, even though it wasn't dereferenceable. I forget the details, but I vaguely recall that it involved some sort of reentrancy, and the reentrant call checked the pointer value to see if it had already been processed.

The compiler team backed out the change.

Of course, if you enforce this rule from the start, then mutating the inbound rvalue reference is no longer a breaking change.

[1] One of the ways that the Visual C++ compiler team tests out some of their ideas is to give a copy of their experimental compiler to the Windows team and see what happens. This lets them exercise their compiler with a monstrous real-world code base.

Raymond Chen

Follow