Should I pay attention to the warning that I’m std::move‘ing from a trivial type? Part 1

devblogs.microsoft.com/oldnewthing/20220512-00

May 12, 2022

Raymond Chen

Say you have a class that is trivial.

```
struct widget_id
{
    int value;
};
```

and you decide that you want to `std::move` it around.

```
widget_id id = get_widget_id();

widget widget = find_widget_by_id(std::move(id));
```

You are using `std::move` because you want to be prepared for the possibility that the `widget_id` might later be changed to something like

```
struct widget_id
{
    std::string value;
}
```

In that case, you want to use a `std::move` to avoid a copy.

But using `std::move` on the original integer-based `widget_id` generates a warning:

> `std::move` of the variable `id` of the trivially-copyable type `widget_id` has no effect.

What is this warning trying to tell you, and should you care?

The language requires merely that a moved-from object be in a legal (albeit unspecified) state. However, many classes go beyond the bare minimum and define their moved-from state. For example, `std::unique_ptr` specifies that if you move out of a unique pointer, the source is left empty. More generally, all RAII types fall into this category, because moving out of an RAII type transfers the responsibility for the resource to the moved-to object. And most of these RAII types provide a way to inspect whether the RAII wrapper has been absolved of any responsibility.

And that's where the warning comes in.

Consider this helper function:

```
bool is_empty(widget_id const& id)
{
    return id.value == 0;
    // -or-
    return id.value.size() == 0;
}
```

This tells you that the `widget_id` doesn't actually contain an id after all. Somebody who expects the `widget_id` to be an RAII-style type might do this:

```
// Remember to add power to this widget, if possible
widget_id id = get_widget_id();

if (wants_power_early()) {
    add_widget_power(std::move(id));
}

...

if (is_empty(id)) {
    // Nobody added power yet, let's do it now.
    add_widget_power(std::move(id));
}
```

This type of mistake is much more likely if the emptiness check is a member of the `widget_id` itself, either as a named member function or as a boolean conversion operator.

```
struct widget_id
{
    int value;
    bool is_empty() const { return value == 0; }
    operator bool() const { return value != 0; }
};
```

Then that last check would be

```
if (!id.is_empty()) {
```

or the even more natural-looking

```
if (id) {
```

Okay, so maybe you know that you're not operating on an RAII type, and that you know that the `std::move` may not actually move anything. Is there some way to avoid having to disable the warning at every single place you do the `std::move`?

One way is to make your type no longer trivial. Probably the simplest way is to give it a user-defined destructor that is equivalent to the trivial destructor.

```
struct widget_id
{
    int value;

    ~widget_id() { } // no longer a trivial type
};
```

On the other hand, making the type no longer trivial is likely to have unintended cascade effects seeing as triviality affects many other things: If you make a type non-trivial, then you lose the ability to do things like use `memcpy` to copy instances of the type, or use it as a buffer for I/O operations.

Another option is to route the call through a helper, and then annotate the helper.

```
template<typename T>
constexpr decltype(auto) move_allow_trivial(T&& t) noexcept
{
    return std::move(t); // NOLINT
}
```

If you don't mind that you're moving a trivial type, you can call this helper instead of calling `std::move` directly.

There's another case for moving from a trivial type. We'll look at it next time.

Raymond Chen

Follow