What should the state of a moved-from object be?

devblogs.microsoft.com/oldnewthing/20201218-00

December 18, 2020

Raymond Chen

In the C++ language, there is the concept of *moving*, which is a way of transferring resources from one object to another. The language specifies that a moved-from object is in a legal but indeterminate state. Basically, the object is in a state that can be safely destructed, or operated upon in a way that is not dependent on the previous state (say, assigning a new value).

From a language-lawyer point of view, any legal state is a valid state for a moved-from object. In fact, it's even legal for "moving" an object to consist of simply copying it, with no resources being transferred at all.

What's left is then a philosophical question of what should, in an ideal world, be the state of a moved-from object.

There are two leading schools of thought on this subject.

One is that the implementation should take full advantage of the rules and use the source object as a garbage can into which it can throw anything it doesn't want. In practice, this philosophy means that move assignments are largely swap operations, where the resources of the source object are assumed by the destination, and the previous resources of the destination are given to the source. Now, if there are simple scalar values, then there are no real "resources" to transfer out, so it's not really a full swap. But anything that requires destruction has been given to the source object.

Another school of thought is that the source object should be left in a state where it controls no resources, rather than being left controlling displaced resources from the destination. In the specific case of an object whose entire purpose is to control a single resource (such as a vector or smart pointer), the object is left in an empty state.

I personally belong to the first camp, where the source object is just a vessel into which the destination object can empty its detritus. An advantage of this is that the resource destruction occurs only once, namely when the source object destructs. (Leaving the object empty would require the destination's former resources to be destructed explicitly as part of the move operation, and then the empty source object will be destructed when the source object is destructed.)

The problem is that people tend to expect the source object to be empty, especially when the object manages a single resource.

For example, if you are making sure to destruct objects outside an internal lock, you are counting on the move-assignment leaving the source empty so that you can erase it from the collection without triggering a call to external code.

As consolation, you should provide a `swap` method and a free `swap` method that is discoverable via argument-dependent lookup. That way, people who really want a swap have a way to do it.

Bonus chatter: Regardless of which school of thought you subscribe to, you need to be careful to leave the source object in an internally-consistent state. For example, consider the class

```
struct totaled_ints
{
  std::vector<int> ints;
  int total = 0;

  void append(int value) {
    ints.push_back(value);
    total += value;
  }
};

totaled_ints v;
v.append(1);
v.append(2);
totaled_ints v2 = std::move(v); // oops
```

In this class, the `total` is a running total of the integers in the `ints` vector. When we use it as the source of a move-constructor, the default move constructor will move the `ints` vector and copy the `total`. That's fine for the newly-constructed `t2` object: Its integers are `1, 2` and its total is 3.

But that's bad news for the source object.

The source object `v` will almost certainly be left with an empty `ints`, but the `total` will remain 3.

The source object was not left in a valid state. This can cause problems if the object is destructed or reused, because the destructor or `reset()` method will try to clean up an object that is in an inconsistent state, which could result in strange bugs. For example, if our `totaled_ints` sends a warning when the total gets too large and recalls the warning when the total drops below the danger level, then this inconsistent state can lead to warnings that are never recalled, or failure to raise warnings when we should.

Raymond Chen

Follow