7. User-provided move constructor. Instrumenting a move constructor during development, if just to ensure that it is being called when expected, isn't a bad idea. Again, we will provide the final result and an appropriate analysis of how we got here:

```
struct Person7b  // adding a user-provided move constructor
{
    String firstName;
    String lastName;

    Person7b(Person7b&& expiring)                    // move constructor
    : firstName(std::move(expiring.firstName))
    , lastName (std::move(expiring.lastName)) { /* user-provided */ }

    // already added to Person7a (not shown)
    Person7b() = default;                            // default constructor
    Person7b(const Person7b&) = default;             // copy constructor
    Person7b& operator=(const Person7b&) = default;  // copy assignment

    // new here in Person7b
    Person7b& operator=(Person7b&&) = default;       // move assignment
};
```

First note the use of `std::move` in the user-provided implementation of the move constructor in the example above. Recall that a parameter of type rvalue reference (`&&`) is itself an *lvalue*, so `std::move` is *required* to enable a move from such a parameter. Not employing `std::move` would mean that these data members would be individually copied rather than moved. Absent a thorough unit test, such inadvertent, pessimizing omissions might well find their way into widespread use.

In the original enhanced version (`Person7a`, which is not shown in the example), the user-provided move constructor immediately renders the class to be of nonaggregate type. Moreover, both the copy constructor and the copy-assignment operator are deleted, and the default constructor and the move-assignment operator are not implicitly generated. Since there is no way to create an object and then learn that the move-assignment operator is missing (short of knowing, as we do here), the first step is to get the unit test driver to compile, which is accomplished by defaulting the default constructor, copy constructor, and copy-assignment operator in `Person7a`.

After that, our thorough unit tests can observe that what should be move assignment is falling back on copy assignment, which needlessly allocates new resources rather than transferring them when the source is expiring. By now also defaulting the move-assignment operator, we arrive at a class that again has all the regular functionality of the original `Person` class, namely, `Person7b` (shown in the code snippet above), but absent the ability to be aggregate initialized.

8. User-provided move-assignment operator. Instrumenting a move-assignment operator during development, just like a move constructor, can be useful: