```
            if (!rhs.d_movedFrom)
            {
                // Move/copy all state from rhs.
            }
            d_movedFrom = rhs.d_movedFrom;
            rhs.d_movedFrom = true;
        }
        return *this;
    }

    ~MovableMechanism()
    {
        if (!d_movedFrom)
        {
            // ...  (existing implementation of Mechanism)
        }
    }

    void doWork()
    {
        assert(!d_movedFrom);  // needs to be added to ALL public functions
        // ... (existing implementation of doWork)
    }
};
```

As the sketch of the `MovableMechanism` class implementation above suggests, adding movability to a nonmovable noncopyable type, `Mechanism`, requires modifications to almost every aspect of ~~of~~ the type — at least all publicly accessible aspects of it. Should any of the original special member functions of `Mechanism` have been defaulted (see Section 1.1. "Defaulted Functions" on page 33), they might now need to become user provided to properly handle the new moved-from state. Moreover, changes to preconditions or essential behavior will necessarily invalidate any corresponding documentation. What's more, robust software implementing defensive checks (e.g., using standard C `assert` macros) will naturally want to implement new checks for all the newly established preconditions. Finally, all behavioral changes will require thorough updating of existing tests along with addressing all functionality that previously did not exist, including the **negative testing** of all newly added defensive checks.

Inconsistent expectations on moved-from objects

When creating a type that supports move operations, a key decision to be made is in what states moved-from objects of that type may be left and what operations will be valid on such objects. When writing code that uses a movable type, especially generic code, it is also important to understand and document the requirements on the template parameters. When a generic type has higher expectations for what can be done with moved-from objects than