because (a) move construction of `std::string` objects is cheap and (b) most compilers will *elide* the extra move anyway, yielding equivalent code to the `RedString` case.[1]

Similarly, the expression `BlueString("goodbye").value()` yields a temporary `std::string`, but in this case the temporary ~~variable~~ is bound to the reference, `s2`, which extends its lifetime until `s` goes out of scope. Thus, `s2[0]` safely indexes a string that is still live.

Note one more, rather subtle, difference between the behavior of `value` for `RedString` versus `BlueString`:

```
void f4()
{
    RedString  rs("hello");
    BlueString bs("hello");

    std::move(rs).value();  // rs.d_value is unchanged.
    std::move(bs).value();  // bs.d_value is moved from.
}
```

Calling `value` on an *rvalue* of type `RedString` doesn't actually change the value of `d_value`; it is not until the returned rvalue reference is actually used (e.g., in a move constructor) that `d_value` is changed. Thus, if the return value is ignored, nothing happens. Conversely, for `BlueString`, the return of `value` is always a move-constructed temporary `std::string` object, causing `d_value` to end up in a **moved-from state**, even if the return value is ultimately ignored. This difference in behavior is seldom important in practice, as reasonable code will assume nothing about the value of a variable after it was used as the argument to `std::move`.

Forbidding modifying operations on *rvalues*

Modifying an *rvalue* means modifying a temporary object that is about to be destroyed. A common example of a defect resulting from this behavior is accidental assignment to a temporary object. Consider a simple `Employee` class with a `name` accessor and a function that attempts to set the name:

```
#include <string>   // std::string

class Employee
{
public:
    // ...
    std::string name() const;
    // ...
};
```

[1] Beginning with C++17, the description of the way return values are initialized changed so as to no longer **materialize** a ~~temporary variable~~ in this situation. This change is sometimes referred to as **guaranteed copy elision** because, in addition to defining a more consistent and portable semantic, it effectively legislates the optimization that was previously optional.