One downside of this design is that the reference returned from the *rvalue*-ref-qualified overload could outlive the `RedString` object:

```
void f2()
{
    std::string&& s1 = RedString("goodbye").value();
    char c1 = s1[0];  // Bug, s1 refers to a destroyed string.

    const std::string& s2 = RedString("goodbye").value();
    char c2 = s2[0]; // Bug, s2 refers to a destroyed string too.
}
```

The temporary ~~variable~~ created by the expression `RedString("goodbye")` is destroyed at the end of the **statement**; **lifetime extension** does not come into play because `s` is not bound to the **temporary object** itself, but to a reference returned by the `value` member function. Returning a **dangling reference** can be avoided by returning by *value* rather than by reference:

```
class BlueString
{
   std::string d_value;

public:
    BlueString(const char* s = "") : d_value("Blue: ") { d_value += s; }

          std::string& value() &       { return d_value; }
    const std::string& value() const & { return d_value; }
          std::string  value() &&      { return std::move(d_value); }
    // Note that this third overload returns std::string by value.

    // ...
};

void f3()
{
    std::string s1 = BlueString("hello").value();

    std::string&& s2 = BlueString("goodbye").value();
    char c = s2[0];  // OK, lifetime of s has been extended.
}
```

The expression `BlueString("hello").value()` yields a temporary `std::string` initialized via **move-construction** from the **data member** `d_value`. The **variable** `s1` is, in turn, move-constructed from that temporary. Compared to the `RedString` version of `value` that returned an rvalue reference, this sequence logically has one extra **move operation** (two move-constructor calls instead of one). This extra move does not pose a problem in practice