```
class String
{
public:
    String(const std::string&);  // Copy the contents of string.
};

class S
{
    String d_s;  // Implementation changed.

public:
    S(std::string s) : d_s(std::move(s)) { }  // Implementation did not change.
};

std::string getStr();

int main()
{
    std::string lval;

    S s1(lval);     // 2 copies
    S s2(getStr()); // 1 move and 1 copy
}
```

The problem is that now we are copying the argument twice: once into the `lval` parameter and then again into the `String` data member, `d_s`. Had we written the requisite overloads, we would not be in this situation:

```
class S
{
    String d_s;

public:
    S(const std::string& s) : d_s(s)            { }
    S(std::string&& s)      : d_s(std::move(s)) { }
};
```

So, unless we are absolutely certain that we will never change the implementation of our class, designing a constructor to take a sink argument by value can be suboptimal.

Disabling NRVO

Named return value optimization (NRVO) can occur only if the expression being returned from all paths through the function is the name of the same local variable. If we use `std::move` in a return statement, we are returning the return value of another function, i.e., `std::move`, and not a local variable by name, even though as developers we know that