```
void f()
{
    S a;
    const S b;

    S x(a);  // invokes forwarding constructor
    S y(b);  // invokes copy constructor
}
```

Despite the programmer's intention to copy from `a` into `x`, the forwarding constructor of `S` was invoked instead, because `a` is a non**`const`** *lvalue* expression, and instantiating the forwarding constructor with `T = S&` results in a better match than even the copy constructor.

This potential pitfall can arise in practice, for example, when writing a value-semantic wrapper template, e.g., `Wrapper`, that can be initialized by *perfectly forwarding* the object to be wrapped into it:

```
#include <string>   // std::string
#include <utility>  // std::forward
template <typename T>
class Wrapper  // wrapper for an object of arbitrary type 'T'
{
private:
    T d_datum;

public:
    template <typename U>
    Wrapper(U&& datum) : d_datum(std::forward<U>(datum)) { }
        // perfect-forwarding constructor to optimize runtime performance

    // ...
};

void f()
{
    std::string s("hello world");
    Wrapper<std::string> w0(s);  // OK, s is copied into d_datum.

    Wrapper<std::string> w1(std::string("hello world"));
        // OK, the temporary string is moved into d_datum.
}
```

Similarly to the example involving class `S` in the example above, attempting to copy-construct a non**`const`** instance of `Wrapper`, e.g., ~~`wr` in the example above~~, results in an error:

```
void g(Wrapper<int>& wr)  // The same would happen if wr were passed by value.
{
```