```
void f()
{
    Dictionary d;

    std::string s = "car";
    d.addWord(s);   // instantiates addWord<std::string&>

    const std::string cs = "toy";
    d.addWord(cs);  // instantiates addWord<const std::string&>

    d.addWord("house");              // instantiates addWord<char const(&)[6]>
    d.addWord("garage");             // instantiates addWord<char const(&)[7]>
    d.addWord(std::string{"ball"});  // instantiates addWord<std::string&&>
}
```

Depending on the variety of argument types supplied to `addWord`, having many call sites could result in an undesirably large number of distinct template instantiations, perhaps significantly increasing object code size, compilation time, or both.

`std::forward<T>` can enable move operations

Invoking `std::forward<T>(x)` is equivalent to conditionally invoking `std::move` if `T` is an *lvalue* reference. Hence, any subsequent use of `x` is subject to the same caveats that would apply to an *lvalue* cast to an unnamed *rvalue* reference; see Section 2.1."*Rvalue* References" on page 710:

```
template <typename T>
void f(T&& x)
{
    g(std::forward<T>(x));  // OK
    g(x);                   // Oops! x could have already been moved from.
}
```

Once an object has been passed as an argument using `std::forward`, it should typically not be accessed again because it could now be in a moved-from state.

A perfect-forwarding constructor can hijack the copy constructor

A single-parameter constructor of a class `S` accepting a forwarding reference can unexpectedly be a better match during overload resolution compared to `S`'s copy constructor:

```
struct S
{
    S();                               // default constructor
    template <typename T> S(T&&);      // forwarding constructor
    S(const S&);                       // copy constructor
};
```