Note that a function may be overloaded on the reference type alone (see Section 2.1."*Rvalue* References" on page 710); however, overloading on a **`const`** *lvalue* reference and an *rvalue* reference occur most often in practice. In this specific case — where `f` is a function template, `T` is a template type parameter, and the type of the parameter itself is exactly `T&&` — the `forRef` function parameter in the code snippet above denotes a *forwarding reference*. If `f` is invoked with an *lvalue*, `forRef` is an *lvalue* reference; otherwise, `forRef` is an *rvalue* reference.

Given the dual nature of `forRef`, one rather verbose way of determining the original value category of the passed argument would be to use the `std::is_lvalue_reference` **`type trait`** on `forRef` itself:

```
#include <type_traits>  // std::is_lvalue_reference
#include <utility>  // std::move

template <typename T>
void f(T&& forRef)      // forRef is a forwarding reference.
{
    if (std::is_lvalue_reference<T>::value)  // using a C++11 type trait
    {
        g(forRef);             // propagates forRef as an lvalue
    }                          // invokes g(const S&)
    else
    {
        g(std::move(forRef));  // propagates forRef as an rvalue
    }                          // invokes g(S&&)
}
```

The `std::is_lvalue_reference<T>::value` predicate above asks if the object bound to `forRef` originated from an *lvalue* expression and allows the developer to branch on the answer. A better solution that captures this logic at compile time is generally preferred; see *The* `std::forward` *utility* on page 385:

```
#include <utility>  // std::forward

template <typename T>
void f(T&& forRef)
{
    g(std::forward<T>(forRef));
        // same as g(std::move(forRef)) if and only if forRef is an rvalue
        // reference; otherwise, equivalent to g(forRef)
}
```

A `client` function invoking `f` will enjoy the same behavior with either of the two implementation alternatives offered above: