same syntax. For any given type `T`, whether the `T&&` syntax designates an *rvalue* reference or a *forwarding* reference depends entirely on the surrounding context.[5]

```
template <typename T> struct S0 { void f(T&&); };  // rvalue reference
struct S1 { template <typename T> void f(T&&); };  // forwarding reference
```

Furthermore, even if `T` is subject to template argument deduction, the presence of *any* qualifier will suppress the special *forwarding*-reference deduction rules:

```
template <typename T> void f(T&&);           // forwarding reference
template <typename T> void g(const T&&);     // const rvalue reference
template <typename T> void h(volatile T&&);  // volatile rvalue reference
```

It is remarkable that we still do not have some unique syntax — hypothetically, `&&&` — that we could use, at least optionally, to imply unequivocally a *forwarding* reference that is independent of its context.

Metafunctions are required in constraints

As we showed in *Use Cases* on page 386, being able to perfectly forward arguments of the same general type and effectively leave only the value category of the argument up to type deduction is a frequent need.

The challenge of correctly forwarding only the value category, however, is significant. The template must be constrained using SFINAE and the appropriate **type traits** to disallow types that aren't some form of a cv-qualified or ref-qualified version of the type that we want to accept. As an example, let's consider a function intended to *copy* or *move* a `Person` object into a data structure:

[5] In C++20, developers might be subject to additional confusion due to the new terse concept notation syntax, which allows function templates to be defined without any explicit appearance of the `template` keyword. As an example, a constrained function parameter, like `Addable auto&& a` in the example below, is a forwarding reference; looking for the presence of the mandatory `auto` keyword is helpful in identifying whether a type is a forwarding reference or *rvalue* reference:

```
template <typename T>
concept Addable = requires(T a, T b) { a + b; };

void f(Addable auto&& a);  // C++20 terse concept notation

void example()
{
    int i;

    f(i);  // OK, decltype(a) is int& in f.
    f(0);  // OK, decltype(a) is int&& in f.
}
```