Notice that we are using the **typename** keyword in the previous example as a generalized way of indicating, ~~during~~ ~~**template** instantiation~~, that a dependent name is a type as opposed to a value.[1]

Identifying forwarding references

The syntax for a *forwarding* reference (`&&`) is the same as that for *rvalue* references; the only way to discern one from the other is by observing the surrounding context. When used in a manner where type deduction can take place, the `T&&` syntax does not designate an *rvalue* reference; instead, it represents a *forwarding* reference. For type deduction to be in effect, a function template must have a type parameter (e.g., `T`) and a function parameter of a type that exactly matches that parameter followed by `&&` (e.g., `T&&`):

```
struct S0
{
    template <typename T>
    void f(T&& forRef);
        // Fully eligible for template-argument type deduction: forRef
        // is a forwarding reference.
};
```

Note that if the function parameter is qualified, the syntax reverts to the usual meaning of *rvalue* reference:

```
struct S1
{
    template <typename T>
    void f(const T&& crRef);
        // Eligible for type deduction but is not a forwarding reference: due
        // to the const qualifier, crRef is an rvalue reference.
};
```

If a member function of a class template is not itself also a template, then its template type parameter will not be deduced:

```
template <typename T>
struct S2
{
    void f(T&& rRef);
       // Not eligible for type deduction because T is fixed and known as part
       // of the instantiation of S2: rRef is an rvalue reference.
};
```

[1]In C++20, the **typename** disambiguator is no longer required in some of the contexts where a dependent qualified name must be a type. For example, a dependent name used as a function return type — `template <typename T> T::R f();` — requires no **typename**.