Similarly, ~~ref qualifiers~~ other than `&&`, i.e., `&` or `&&` along with any cv-qualifiers, do not alter the deduction process, and they too are applied after deduction:

```
template <typename T> void rf(T& x);
template <typename T> void crf(const T& x);

void example2(int i)
{
    rf(i);   // OK, T is deduced as int; x is an int&.
    crf(i);  // OK, T is deduced as int; x is a const int&.

    rf(0);   // Error, expects an lvalue for 1st argument
    crf(0);  // OK, T is deduced as int; x is a const int&.
}
```

Type deduction works differently for *forwarding* references where the only qualifier on the template parameter is `&&`. For the sake of exposition, consider a function template declaration, `f`, accepting a forwarding reference, `forRef`:

```
template <typename T> void f(T&& forRef);
```

We saw in the example on page 378 that, when `f` is invoked with an *lvalue* of type `S`, then `T` is deduced as `S&` and `forRef` becomes an *lvalue* reference. When `f` is instead invoked with an *xvalue* of type `S` (see Section 2.1."*Rvalue* References" on page 710), then `T` is deduced as `S` and `forRef` becomes an *rvalue* reference. The underlying process that results in this duality relies on **reference collapsing** (see the next section) and special **type deduction** rules introduced for this particular case. When the type `T` of a *forwarding* reference is being deduced from an expression `E`, `T` itself will be deduced as an *lvalue* reference if `E` is an *lvalue*; otherwise, normal type-deduction rules will apply, and `T` will be deduced as a nonreference type:

```
void g()
{
    int i;
    f(i);  // i is an lvalue expression.
           // T is therefore deduced as int& --- special rule!
           // T&& becomes int& &&, which collapses to int&.

    f(0);  // 0 is an rvalue expression.
           // T is therefore deduced as int.
           // T&& becomes int&&, which is an rvalue reference.
}
```

For more on general type deduction, see Section 2.1."**auto** Variables" on page 195.

Reference collapsing

As we saw in the previous section, when a function having a *forwarding* reference parameter, `forRef`, is invoked with a corresponding *lvalue* argument (e.g., a named variable), an