What would happen if we did the following under the freer rules of early C++?

```c
long a = 3, b = 5;
swapr(a, b);
```

Here there is a type mismatch, so the compiler would create two temporary `int` variables, initialize them to 3 and 5, and then swap the contents of the temporary variables, leaving `a` and `b` unaltered.

In short, if the intent of a function with reference arguments is to modify variables passed as arguments, situations that create temporary variables thwart that purpose. The solution is to prohibit creating temporary variables in these situations, and that is what the C++ standard now does. (However, some compilers still, by default, issue warnings instead of error messages, so if you do see a warning about temporary variables, don't ignore it.)

Now think about the `refcube()` function. Its intent is merely to use passed values, not to modify them, so temporary variables cause no harm and make the function more general in the sorts of arguments that it can handle. Therefore, if the declaration states that a reference is `const`, C++ generates temporary variables when necessary. In essence, a C++ function with a `const` reference formal argument and a nonmatching actual argument mimics the traditional passing by value behavior, guaranteeing that the original data is unaltered and using a temporary variable to hold the value.

**Remember**

If a function call argument isn't an Lvalue or does not match the type of the corresponding `const` reference parameter, C++ creates an anonymous variable of the correct type, assigns the value of the function call argument to the anonymous variable, and has the parameter refer to that variable.

**Use const When You Can**

There are three strong reasons to declare reference arguments as references to constant data:

- Using `const` protects you against programming