whether any part of the expression — the function as well any expressions used in its arguments — might throw. Failing to grasp this subtlety can result in a function being marked **noexcept(false)** even when the expression required by the implementation does not throw. Consider a function template `eval1` that takes an **invocable** object `f` of the **template parameter** type `F` and calls it with a string argument, `stringArg`, of type **const** `std::string&`. We want `eval1` to have the same exception specification as `f::`**operator**`()`:

```
#include <string>  // std::string

template <typename F>
void eval1(F f, const std::string& stringArg) noexcept(noexcept(f("")))  // Bug
{
    f(stringArg);
}
```

Here, we are making a concerted effort to pass the exception specification from `f` through to the exception specification of `eval1`. For conciseness, we pass an empty string as a **placeholder** for the string argument in the expression given to the **noexcept** operator, reasoning that the expression is *unevaluated* and hence can be safely abbreviated. Alas, if the argument to `f` has type **const** `std::string&`, then passing `""` requires a call to the potentially throwing **converting constructor**, `std::string(`**const char***`)`. Consequently, **noexcept**`(f(""))` would be **false** because **noexcept**`(std::string(""))` is **false** regardless of whether the call to `f` is guaranteed not to throw when called with an already constructed `std::string`, such as the object referred to by `stringArg`.

The obvious fix in this case is to use exactly the same expression in the **noexcept** specifier that is used in the return statement, i.e., `f(stringArg)`. Before we explore this approach, let's consider a couple of alternatives that might be more appealing in the case of more complex expressions. The argument to `f` must be a nonthrowing expression that can bind to a **const** `std::string&` without invoking any possibly throwing conversions. A simple fix in this case, therefore, would be to simply switch our placeholder string to an invocation of the default constructor for `std::string`, which is declared **noexcept**:

```
#include <string>  // std::string

template <typename F>
void eval2(F f, const std::string& stringArg)
     noexcept(noexcept(f(std::string())))     // OK
{
    f(stringArg);
}
```

This fix, however, does not generalize to the case where `F` is passed an argument that is dependent on a template parameter type, perhaps using **perfect forwarding** (see Section 2.1."Forwarding References" on page 377). Because the type of the argument to `f` is not known until the template is instantiated, it is not known whether it has a default constructor nor whether any such default constructor is **noexcept**. Rather than place an