unnecessary default-constructible constraint on that type, the solution is to use the C++11 library function `std::declval`, ~~defined~~ within the `<utility>` header:

```
template <typename T>
typename std::add_rvalue_reference<T>::type std::declval() noexcept;
```

The `std::declval` function is not defined, so it is an error to call it in a context where it would actually be evaluated. Its purpose is to create a reference to a type that can be used in an **unevaluated context** such as within the **`noexcept`** operator, **`decltype`**, **`alignof`**, or **`sizeof`**. We can use `std::declval` to create a more general definition of `eval2` (e.g., `eval3`) with a correctly deduced exception specification:

```
#include <utility>  // std::declval, std::forward

template <typename F, typename T>
void eval3(F f, T&& arg) noexcept(noexcept(f(std::declval<T>())))
{
    f(std::forward<T>(arg));
}
```

Note that calling this function can still throw if the copy or move constructor or destructor for the functor type `F` can throw. Avoiding an unnecessary copy of `F` is why, in practice, `F` would typically be passed by reference:

```
template <typename F, typename T>
void eval4(F&& f, T&& arg)
    noexcept(noexcept(std::declval<F>()(std::declval<T>())))
{
    std::forward<F>(f)(std::forward<T>(arg));
}
```

The observant reader will have noticed that all this trouble arises from trying to avoid using function parameters in the exception specification. These parameters are in scope, as the exception specification follows the function parameter list, so the simplest and most reliably correct form of the exception specification would simply contain a copy of the expression used in the body of the function:

```
template <typename F, typename T>
void eval5(F&& f, T&& arg)
    noexcept(noexcept(std::forward<F>(f)(std::forward<T>(arg))))
{
    std::forward<F>(f)(std::forward<T>(arg));
}
```

Because the **`noexcept`** specification exactly mirrors the expression of concern, there is no chance of a discrepancy between the two. Unfortunately, this idiom requires significant code duplication; see *Annoyances — Code duplication* on page 1144.

The example of `eval1` incorrectly produced a **`noexcept(false)`** specification, but it is also possible to produce an erroneous **`noexcept(true)`** specification if an important part of the