```
{
    try
    {
        return fun(std::forward<Ts>(xs)...);   // perfect forwarding to fun
    }
    catch (const std::exception& e)
    {
        log(e.what());                         // log exception information
        throw;                                 // Rethrow the same exception.
    }
    catch (...)
    {
        log("Nonstandard exception thrown.");  // log exception information
        throw;                                 // Rethrow the same exception.
    }
}
```

Here, we enlist the help of `std::forward` and also that of the **`auto -> decltype`** idiom; see Section 1.1."Trailing Return" on page 124 and Section 1.1."**`decltype`**" on page 25. By using **`auto`** instead of the return type of `logExceptions` and following with `->` and the trailing type **`decltype`**`(fun(std::forward<Ts>(xs)...))`, we state that the return type of `logExceptions` is the same as the type of the call `fun(std::forward<Ts>(xs)...)`, which matches perfectly the expression that the function will actually return.

In case the call to `fun` throws an exception, `logExceptions` catches, logs, and rethrows that exception. So `logExceptions` is entirely transparent other than for logging the passing exceptions. Let's see it in action. First, we define a function, `assumeIntegral`, that is likely to throw an exception:

```
#include <stdexcept>  // std::runtime_error

long assumeIntegral(double d)            // throws if d has a fractional part
{
    long result = static_cast<long>(d);  // Compute the returned value.
    if (result != d)                     // Verify.
        throw std::runtime_error("Integral expected");
    return result;
}
```

To call `assumeIntegral` via `logExceptions`, we just pass it along with its argument:

```
void test()
{
    long a = logExceptions(assumeIntegral, 4.0);  // Initialize a to 4.
    long b = logExceptions(assumeIntegral, 4.4);  // throws and logs
}
```