Forgetting to use the noexcept operator in the noexcept specifier

The **`noexcept`** specifier is commonly used in conjunction with the **`noexcept`** *operator* to compute the exception specification of a function (or function template) from the exception specification of a particular expression. The tested expression typically involves variables of a type dependent on the template arguments, as otherwise the answer is known a priori given specific types:

```
template <typename T, typename U>
void grow(T& lhs, const U& rhs) noexcept(noexcept(lhs += rhs))
{
    lhs += rhs;
}
```

Using a nested **`noexcept`** — i.e., **`noexcept`**`(`**`noexcept`**`(`*`expression`*`))` — looks odd but is necessary; forgetting the inner **`noexcept`** — i.e., writing just **`noexcept`**`(`*`expression`*`)` — can, in some cases, lead to code that still compiles but not with the expected semantics. Such flawed exception specifications are easy to write yet often hard to spot in code review as they look like the familiar **`noexcept`** specifier. The **`noexcept`** specifier expects a constant expression that is **contextually convertible to `bool`**. Fortunately, when the inner **`noexcept`** is accidentally omitted, the common case is that *expression* is not a compile-time constant expression and will thus trigger a compiler error. There are a few such mistakes that do constitute valid code, however, and those mistakes can easily result in the function declaration having the wrong exception specification.

Consider, for example, a pair of **`inline`** functions `g1` and `g2` that simply return **`false`**, both declared as **`noexcept`**, but with `g2` defined as **`constexpr`** (see Section 2.1.“**`constexpr`** Functions” on page 257) while `g1` is not. We then define two functions, `f1` and `f2`, that simply delegate to `g1` and `g2`, respectively. Each tries to infer its corresponding exception specification from the exception specification of its called function but neglects to nest the **`noexcept`** operator within the **`noexcept`** specification:

```
          bool g1() noexcept { return false; }
constexpr bool g2() noexcept { return false; }

bool f1() noexcept(g1()) { return g1(); }  // Error, g1() not a constant expr.
bool f2() noexcept(g2()) { return g2(); }  // Bug, noexcept(false)

static_assert(noexcept(f2()) == noexcept(g2()), "");  // Error, f2 not noexcept
```

In the example above, the declaration of `f1` is ill formed, producing a compilation error, because the argument, `g1()`, to its **`noexcept`** specifier is not a constant expression. Hence, the compiler prevents us from this pitfall in this common case. In the case of `f2`, however, the expression specifier is valid because `g2()` is a constant expression returning a type convertible