```
struct F : B  // All special members of B are noexcept(false) apart
              // from the destructor.
{
    F()                    noexcept(false) = default;  // default constructor
    F(const F&)            noexcept        = default;  // copy constructor
    F& operator=(const F&)                 = default;  // copy assignment
    F(F&&)                 noexcept(true)  = default;  // move constructor
    F& operator=(F&&)                      = default;  // move assignment
    ~F()                   noexcept(true)  = default;  // destructor
};
```

Notice that, in class `F` in the code snippet above, both the *copy* and *move* constructors are mislabeled as being **`noexcept(true)`** when the defaulted declaration would have made them **`noexcept(false)`**. Such inconsistency is not in and of itself an error until an attempt is made to use that function; see Section 1.1."Deleted Functions" on page 53:

```
void test()
{
    F f0, f1;             // OK,    default constructor
    F f2(f0);             // Error, copy constructor is deleted.
    f0 = f1;              // OK,    copy assignment
    F f3(std::move(f0));  // Error, move constructor is deleted.
    f0 = std::move(f1);   // OK,    move assignment
}                         // OK,    destructor
```

Note that the need for an *exact* match between explicitly declared and defaulted **`noexcept`** specifications is unforgiving in *either* direction. That is, had we, say, attempted to restrict the contract of the class `F` above by decorating its destructor with **`noexcept(false)`** when the defaulted implementation would have happened to have been **`noexcept(true)`**, that destructor would have nonetheless been implicitly deleted, severely crippling use of the class.

In C++20, mismatched explicit **`noexcept`** specification for a defaulted special member function no longer results in that member function being deleted. Instead, the explicit **`noexcept`** specification is accepted by the compiler.[5] This behavior change was accepted as a defect report, applying to C++11 and later, and is implemented, e.g., starting from Clang 9 (c. 2019), GCC 10.1 (c. 2020), and MSVC 16.8 (c. 2020).

Finally, the C++11 specification does not address directly the implicit exception specification for inheriting constructors (see Section 2.1."Inheriting ~~Ctors~~" on page 535), yet most popular compilers handle them correctly in that they take into account the exceptions thrown by the inherited constructor and all the member initialization involved in invocation of the inherited constructor.

In C++14, all implicitly declared special member functions, including inheriting constructors, are **`noexcept(false)`** if any function they invoke directly has an exception specification that allows all exceptions; otherwise, if any of these directly invoked functions has a dynamic exception specification, then the implicit member will have a dynamic exception specifica-

[5] See **smith19**.