Given a nofail function, adding **noexcept** will not affect its in-contract behavior. For functions that use exceptions to report failure, adding **noexcept** can easily turn a *reliable* function having an infallible implementation into an optimistic one stripped of its ability to communicate any difficulties in satisfying the contract, should any arise. As an example, suppose we applied **noexcept** to the two functions of `std::vector` discussed above:

```
template <typename T>
class vector {
// ...
    T& operator[](std::size_t index) noexcept;
        // Return a reference to the modifiable element at the specified index.
        // The behavior is undefined unless index < size().

    T& at(std::size_t index) noexcept;
        // Return a reference to the modifiable element at the specified index
        // unless !(index < size()) in which case call std::terminate.
// ...
};
```

For `std::vector::operator[]`, adding **noexcept** has no effect on in-contract behavior but limits flexibility (e.g., to provide arbitrary behavior when called out of contract). In the case of `std::vector::at`, however, instead of detecting an out-of-range call and reporting it via an exception, the function will now be forced to call `std::terminate`. To be clear, adding **noexcept** to a function that might throw is not a means of suppressing (i.e., swallowing) exceptions.

In short, when we see a function declared **noexcept**, it does not necessarily — in and of itself — imply an *optimistic* let alone a nofail function. Decorating a function with **noexcept** implies merely that the function cannot, under ~~no~~ circumstances, throw an exception.

It is also important to note that nofail and **fault tolerant** are not the same thing. Achieving a fault-tolerant nofail guarantee in, say, an embedded system typically requires redundant, independently designed processes running on autonomous hardware. For example, to provide a fault-tolerant nofail guarantee in a vehicle-control system, there might be three or more redundant processes running on independent hardware that participate in a *voting system* to determine the best course of action or to decide whether to trust some surprising sensor data. The software is designed to fail hard if it detects that the subsystem in which it resides has becomes unreliable and to restart quickly and smoothly while the redundant processes continue voting and maintain control of the vehicle.

To be clear, nothing we have said above is about trying to defend against defects in our own software. Extensive code review followed by extensive unit, integration, system, and beta testing is how we do that. An effective technique for ensuring correctness that is complementary to unit testing and static analysis involves redundant (optional) runtime defensive checks. If one of these checks is enabled and determines that the software is no longer in a logically coherent state then, rather than attempting to work around the defect, the prevailing wisdom is to fail fast and loudly — but possibly after attempting to safely save any important in-process data.