the same way that the return value restricts the potential values that can be returned normally.[27]

Yet another consideration when marking as **noexcept** a function having a specifically narrow contract involves the relative ease of validating defensive precondition checks. Library authors will often want to respond to simple, easily detected violations of a contract by providing, in certain assertion-enabled build modes modes, a defensive check, e.g., using a C-style `assert`. A more sophisticated defensive-checking framework[28] might be configured to throw an exception (e.g., `contract_violation`) when an out-of-contract call is detected; such a configuration could be used within a unit-testing framework to test the defensive checks themselves without causing the test program to terminate. If, however, a function having a narrow contract is marked as **noexcept**, then such a framework would not work, as the attempt to throw `contract_violation` would always terminate the program.

Just prior to the release of C++11, the Standard Library adopted the cautious principle of deliberately *not* marking functions in the Standard Library having narrow contracts as **noexcept**, thereby granting implementers freedom to add such specifications where they believe their own implementation might benefit. This principle is known as the *Lakos Rule* after John Lakos, one of this book's authors, who originally put forward the guideline.[29]

Common best practice these days — especially given the increasingly wide adoption of the zero-overhead exception model — is to treat the use of exceptions in contracts as truly exceptional. Hence, we would not expect an exception to be thrown with any frequency in a modern, well-designed, defect-free program operating within the limits of its platform's available resources.

When acquiring relatively low-level system resources, such as opening a socket or a known-to-exist file, errors, though relatively rare, do occasionally occur. Despite their rarity, the inordinately large latency and inherent nonparallelizability associated with propagating an exception when it does occur can often prove prohibitive. Reproducible benchmarks using the zero-cost model show that a thrown exception is typically "orders of magnitude" slower than returning an error status.[30] Hence, it is not uncommon for such system-level features to opt for other means of communicating such infrequent but urgent information. Ironically, widespread use of **noexcept** is inconsistent with *any* use of exceptions to propagate exceptional failures up through enough stack frames to reach one where sufficient context exists to properly address the error; see *Accidental `terminate`* on page 1124.

Conflating `noexcept` with nofail

A function that is declared **noexcept** might not be nofail. Moreover, a function that happens to be nofail (today) might not be declared **noexcept**, and perhaps for good reason; see *Overly*

[27]See **lakos14b**, time 17:57.
[28]**bde14**, /bsl/bsls/bsls_assert
[29]**meredith11**
[30]See **nayar20**.