circumstances (e.g., embedded systems), we might want to use **noexcept** to reduce object code size; see *Use Cases — Reducing object-code size* on page 1101.

The zero-cost exception model *itself* (unlike previous models) introduces absolutely no overhead into the (exception-free) hot path, leaving only minimal opportunities for **noexcept** to inform the compiler of local "peephole" runtime performance optimization opportunities — e.g., by eliding unused code explicitly supplied on the hot path by the user. Effective use of **noexcept** solely for the purpose of local (*non*algorithmic) runtime object-code optimization fairly demands benchmarking to justify the risks associated with premature specification (see *Overly strong contracts guarantees* on page 1112) and unexpected program termination (see *Accidental* `terminate` on page 1124).

One possible workaround to avoid such pitfalls would be to make the body of the called function, `g`, visible to the calling function, `f`. Doing so, however, strongly compile-time couples the entire body of `g` to `f`, thereby reducing the independent malleability of `g`. Currently, there is no general solution that gives us local code optimization without making a permanent contractual agreement; see *Annoyances — Algorithmic optimization is conflated with reducing object-code size* below.

## Annoyances

### Algorithmic optimization is conflated with reducing object-code size

If a move or swap operation throws an exception in generic code, any modification of the original object is considered irreversible because neither the moved-from nor the moved-to object is known to have a useful value. The **noexcept** operator (see Section 2.1."**noexcept** Operator" on page 615) and the accompanying **noexcept** specifier were invented to allow algorithms — especially those using move or swap operations, such as `std::sort` and `std::rotate` — to choose the fastest way to perform a task without risk of falling into such nonrecoverable situations. An algorithm can use the **noexcept** operator to determine (at compile time) if the move or swap operations it needs to use may throw and, if so, opt to use copy instead of move so that the original objects remain unchanged in the event of a thrown exception.

Decorating a function with **noexcept** might lead to some optimizations; see *Use Cases — Reducing object-code size* on page 1101. This side effect of **noexcept** could perhaps provide an incentive for developers to use the **noexcept** specifier even when there is no algorithmic benefit to be had, possibly committing to nonthrowing interfaces too early and thereby limiting the evolution of the interface design; see *Potential Pitfalls — Overly strong contracts guarantees* on page 1112 and *Unrealizable runtime performance benefits* on page 1134.

Ideally, the compiler could be given enough information to perform optimizations leveraging a nonthrowing function implementation without contractually obligating the function *never* to throw in the future. For any function `g` whose body is visible in the current translation unit (e.g., function templates and **inline** functions from an included header file) or when using link-time optimization, the compiler can already determine that a function's implementation will never throw — even if it is not declared **noexcept**. Without inspecting the function body,