In the above final version, `eval7`, the subexpression `std::string(std::forward<S>(str))` will be **`noexcept(true)`** if and only if it would invoke a **`noexcept`** constructor were it actually evaluated, thus expressing the correct predicate for the pass-through function.

Unrealizable runtime performance benefits

As we know (see *Use Cases — Reducing object-code size* on page 1101), using **`noexcept`** under certain well-understood circumstances can measurably reduce object-code size.

Conventional wisdom suggests that "less code runs faster" and "more code runs slower." When there is less object code, it typically means that fewer machine instructions are executed. When there is more code, there might be more *proximate* instructions that — even if seldom executed — can nonetheless put pressure on a limited hardware resource, namely, the instruction cache.

Historically, support for exceptions resulted in executing additional instructions along the **exception-free path** — even when no exceptions were thrown. With the increasingly ubiquitous use of the **zero-cost exception model**, no additional proximate machine instructions are introduced to support exceptions when an exception is not thrown — a.k.a the `hot path`. In adopting this zero-cost model, we aggressively trade off both latency and throughput when an exception is actually thrown — a.k.a., the `cold path`.

Despite contributing no overhead on the `exception-free path`, a call to a function that could potentially throw might, however, preclude otherwise beneficial optimizations. Short of disabling all support for exceptions ubiquitously (e.g, on the compiler's command line), the best we can do currently to reinstate such runtime optimization opportunities is either to (1) make the body of a nonthrowing function visible when compiling a function that calls it, or (2) declare the nonthrowing function as **`noexcept`**, but see *Annoyances — Algorithmic optimization is conflated with reducing object-code size* on page 1143.

It will turn out that — unlike algorithmic improvements — opportunistic use of **`noexcept`** *solely* as a hint to enable the compiler to optimize *runtime* performance will seldom provide *any* gains, let alone consequential ones; hence, its widespread use throughout a codebase for such a purpose would be highly dubious[34]; see *Overly strong contracts guarantees* on page 1112.

Chandler Carruth said it best[35]:

> If you didn't write a benchmark for your code, you do not care about performance. That's not an opinion, Okay? That's fact.

The hot path and the cold path We call the code that runs in the common, or typical, case the `hot path` to reflect that it is the path that is almost always taken. We call the exceptional case the `cold path` because it is by far the path less traveled, is unlikely to be in

[34] On MSVC 19.29 (c. 2021), adding **`noexcept(true)`** to an **`inline`** function — especially a member function of a DLL-exported class — might yield a significant drop in runtime performance; see **dekker19a**.

[35] **carruth17**, time 3:56–4:38