- If the body of a non-trivial destructor (e.g., `S::~S()`) is visible to the compiler and is found to be a no-op — whether or not it is declared **`inline`** — then the optimizer can treat it as trivial, eliminating condition (1), above.
- If the body of a potentially throwing operation is visible to the compiler and is determined not to throw, then the optimizer can treat that operation as not potentially throwing, possibly eliminating conditions (2) and (3), above.

For an exception specification to reduce the size of generated code, that specification must relieve the compiler of some obligation; specifically, the added specification must relieve the function of its obligation to generate some or all of the unwinding logic for a particular function. If function `ff`, above, were decorated with **`noexcept`**, then the compiler would have the option not to lay down code (1) to unwind the stack or (2) even to destroy `s1` and `s2` prior to calling `std::terminate` in the event that an invocation of `gg` throws. Hence — in theory — a compiler can presumably generate less unwinding code for a **`noexcept`** function simply by not destroying any local variables when an uncaught exception tries to escape. Adding a **`noexcept`** specification does, however, place one additional obligation on the function itself: The function must now take responsibility for ensuring that there is no possibility of an `exception` reaching the caller. If there are any potentially throwing expressions in the function, it is required to handle any attempt to unwind by ending with a call to `std::terminate()`, which will unconditionally end the program. Note that `std::terminate` invokes a `terminate_handler`, which can be set globally using `std::set_terminate`. It is **undefined behavior** if this function returns normally or does not terminate execution of the program. This additional call to `std::terminate` might result in a (typically small) increase in the size of the `.o` file, but not necessarily that of a final optimized program.[16]

Decorating a function with **`noexcept`** can also reduce the size of its *caller*, sometimes substantially, even if the called function sees little or no benefit, but see *Potential Pitfalls — Overly strong contracts guarantees* on page 1112 and *Accidental `terminate`* on page 1124. For example, consider what would happen if we were to decorate function `gg`, above, with **`noexcept`**: The two calls to function `gg` in the body of function `ff` would no longer be potentially throwing; thus, `ff` would no longer need any stack-unwinding code. This code-size reduction was observed in every compiler we tested with.[17] Note that no such size reduction would be expected if `S` were trivially destructible (condition #1) or if all invocations of `gg` took place prior to the construction of any non**`static`** local variables of non-trivially destructible type (condition #3), as no unwinding logic would be required in the first place.

A simple yet fairly general framework to investigate the effects of the **`noexcept`** specifier on generated-code size might consist of a non-trivially destructible class type, `S`; the *declaration*

[16]In our observations, GCC 8 (c. 2018) and higher and MSVC 14.x (c. 2019) typically generate less code, sometimes significantly less (but never more), on the cold path when a function meeting all three of the above conditions is declared **`noexcept`**. Conversely, Clang 12.0 (c. 2021), though generating smaller stack-unwinding code in general (compared to GCC), does generate slightly *more* object code with **`noexcept`** than without it to include a **definition** of the runtime-support function `__clang_call_terminate`.

[17]Experiments were performed with GCC 7.1.0 (c. 2017) GCC 11.1 (c. 2021), Clang 12.0 (c. 2021), and MSVC 19.29 (c. 2021).