```
    ~S();  // user-provided, hence non-trivial destructor
};
```

For a function (e.g., `ff`) that constructs one or more objects of type `S` (e.g., `s1` and `s2`) and then invokes a potentially throwing operation (e.g., a call to function `gg`), the compiler must generate unwinding code to destroy those instances of `S` in the event that `gg` throws an exception:

```
void gg();  // declaration of potentially throwing function

void ff()   // definition of function having no exception specification
{
    S s1;
    gg();   // gg call #1
    S s2;
    gg();   // gg call #2
}
```

If, in the body of `ff` above, the first call to `gg` throws, then the unwinding logic must invoke the destructor for `s1`, but if the second call to `gg` throws, then the unwinding logic must invoke the destructors for both `s2` and `s1`, in that order. Nested blocks and conditional statements can complicate the unwinding logic, thereby producing a larger number of unwinding scenarios. Note that this unwinding logic appears on the **cold path** — i.e., it is invoked only if an exception is thrown at run time. Given the seemingly ubiquitous adoption of the **zero-overhead exception model**, ~~this~~ unwinding ~~code~~ bloat~~s~~ object size but do~~es~~ not compromise the performance of the **hot path**; see *Potential Pitfalls — Unrealizable runtime performance benefits* on page 1134.

Such unwinding logic is not always needed and is typically generated only when all three of the following conditions hold.

1. The function body contains one or more automatic variables of **non-trivially destructible** type (see Section 2.1."*Rvalue* References" on page 710); otherwise, there are no destructors to invoke when unwinding.

2. There is at least one expression in the body of the function that might throw — e.g., the invocation of a function that is not `noexcept(true)`; otherwise, there is no need for unwinding logic.

3. The potentially throwing expression is invoked during the lifetime of one of the aforementioned variables, i.e., after it is constructed and before said variable leaves scope; otherwise, there is again no needed unwinding logic.

Note that there are other ways — unrelated to the exception specification — by which the compiler is able to reduce or eliminate stack-unwinding code. The other opportunities typically require the compiler to have visibility into the implementations of functions invoked from the function being compiled.