Chapter 3

Unsafe Features

Caveat emptor. A few modern C++ language features provide potential value in a few fairly specific, niche use cases, yet expose even experienced engineers to ample opportunities for misuse, often with nonobvious, far-reaching, and sometimes dire consequences. This chapter introduces C++11 and C++14 features that in specific cases can be used profitably but at disproportionately high risk. Moreover, the effort required to understand these features, the subtleties surrounding their effective use, and the risks of misusing them go beyond what many organizations would consider cost effective. Though no modern feature of C++ is inherently "unsafe," these few have an especially unattractive risk–reward ratio. An organization's leadership must be circumspect when supporting any use of this chapter's features. Even if they are used properly, code employing such *unsafe* features might be unmaintainable by engineers lacking the requisite mastery of the original authors. Furthermore, the presence of these features in a codebase might lead less experienced developers to employ them in new situations where such use would be strongly contraindicated.

"Unsafe" features are characterized by being of very high risk and little value. Recall from "An *Unsafe* Feature" in Chapter 0 that **`final`** (p. 1007) was our exemplar for an *unsafe* feature. When appropriate, this feature is exactly what's needed; ironically, unlike its *safe* cousin **`override`** (p. 104), both from the same Standards proposal, **`final`** is easily misused and seldom appropriate. Another *unsafe* feature is the **`noexcept`** specifier (p. 1085); unlike its related *conditionally safe* feature the **`noexcept`** operator (p. 615), misuse of this feature can render a codebase brittle and exception unfriendly. Yet another example of an *unsafe* yet useful feature is **`friend`** (p. 1031). Idiomatic use of this feature (e.g., in CRTP) can be of substantial value (e.g., in avoiding copy-paste errors), but most other uses (e.g., involving *long-distance friendship*) can lead to code that does not scale and is inordinately difficult to understand, test, and maintain. The benefit of fully teaching any of these features as part of even an advanced general training course is dubious. Although most of the features presented in this chapter have the potential to add value, all come with a profoundly high risk of being misused or a disproportionately high training cost — not only for implementors, but for maintainers too — and thus are considered *unsafe*.

In short, widespread adoption of *unsafe* features offers no sensible risk–reward ratio and thus is contraindicated. An organization considering incorporating *unsafe* features into a predominantly C++03 codebase would be well advised to adopt strict standards as to whether and under what circumstances such features shall be used. Even if you're an expert in modern C++, you'd be well advised to fully understand and appreciate the pitfalls in each *unsafe* feature you might want to employ.