“safely” as the moniker for the signature aspect of our book and the method by which we rank the risk-to-reward ratio for using a given feature in a large-scale development environment. By contextualizing the meaning of the term “safe,” we apply it to a real-world economy in which everything has a cost in multiple dimensions: risk of misuse, added maintenance burden borne by using a new feature in an older codebase, and training needs for developers who might not be familiar with that feature.

Several factors impact the value added by the adoption and widespread use of any new language feature, thereby reducing its intrinsic safety. By categorizing features in terms of safety, we strive to capture an appropriately weighted combination of the following factors:

- Number and severity of known deficiencies
- Difficulty in teaching consistent proper use
- Experience level required for consistent proper use
- Risks associated with widespread misuse

In this book, the degree of safety of a given feature is the relative likelihood that widespread use of that feature will have positive impact and ~~no~~ adverse effect on a large software company’s codebase.

A *Safe* Feature

Some of the new features of modern C++ add considerable value, are easy to use, and are decidedly hard to misuse unintentionally; hence, ubiquitous adoption of such features is productive, relatively unlikely to become a problem in the context of a large-scale development organization, and generally encouraged — even without training. We identify such staunchly helpful, unflappable C++ features as *safe*.

For example, we categorize the **`override`** contextual keyword as a safe feature because it prevents bugs, serves as documentation, cannot be easily misused, and has no serious deficiencies. If someone has heard of this feature and tried to use it and the software compiles, the codebase is likely better for it. Using **`override`** wherever applicable is always a sound engineering decision.

A *Conditionally Safe* Feature

The vast majority of new features available in modern C++ have important, frequently occurring, and valuable uses, yet how these features are used appropriately, let alone optimally, might not be obvious. What’s more, some of these features are fraught with inherent