boost. The often-overlooked downside, however, is that this choice, once made, is not easily reversed. After a library is released and a **constexpr** function is evaluated as part of a constant expression, no clean way of turning back is available because clients now depend on this compile-time property.

Overzealous use

Overzealous application of **constexpr** can also have a significant impact on compilation time. Compile-time calculations can easily add seconds — or in extreme cases much more — to the compilation time of any translation unit that needs to evaluate them. When placed in a header file, these calculations need to be performed for all translation units that include that header file, vastly increasing total compilation time and hindering developer productivity.

Similarly, making public APIs that are **constexpr** usable without making it clear that they are suboptimal implementations can lead to both (1) excessive runtime overhead compared to a highly optimized non**constexpr** implementation (e.g., for `isPrime` in *Difficulty implementing **constexpr** functions* on page 296) that might already exist in an organization's libraries and (2) increased compile time wherever algorithmically complex **constexpr** functions are invoked.

Compilation limits on compile-time evaluation are typically per *constant expression* and can easily be compounded unreasonably within just a single translation unit through the evaluation of numerous constant expressions. For example, when using the `generateArray` function in *Use Cases — Precomputing tables of data at compile time* on page 291, compile-time limits apply to each individual array element's computation, allowing total compilation to grow linearly with the number of values requested.

One time is cheaper than compile time or run time

Overall, the ability to use a **constexpr** function to do calculations before run time fills in a spectrum of possibilities for who pays for certain calculations and when they pay for them, both in terms of computing time and maintenance costs.

Consider a possible set of five evolutionary steps for a computationally expensive function that produces output values for a moderate number of unique input values. Examples include returning the timestamp for the start of a calendar year or returning the nth prime number up to some maximum n.

1. An initial version directly computes the output value each time it is needed. While correct and written entirely in maintainable C++, this version has the highest runtime overhead. Heavy use will quickly lead the developer to explore optimizations.

2. Where precomputing values might seem beneficial, a subsequent version initializes an array once at run time to avoid the extra computations. Aggregate runtime performance can be greatly improved but at the cost of slightly more code as well as a