Doc. no.: P0816R0
Date: 2017-10-13
Reply to: Titus Winters (titus@google.com)
Audience: LEWG

No More Nested Namespaces in Library Design

Abstract

There are few good technical reasons for nested namespaces. We should stop using them in general, with few / small exceptions in very limited circumstances. (The bulk of this paper was sent to the lib-ext reflector during the Toronto meeting - it is provided now as a proper paper so it can be discussed and cited going forward.)

Summary

We generally choose to optimize for readers of C++ code (maintainers, code reviewers, future editors, casual readers) or writers of C++ code (the initial author). It's possible to optimize code practices for other features (performance above all else) but it is unusual and irrelevant to this discussion. Nested namespaces are harmful for both writers and readers - we should not set bad precedent for authors nor harm readers.

The reader side of the equation:

As the length of a fully qualified name goes up, the odds of it being using-declared increase. Similarly, as a commonly-used nested namespace gains more usage, the odds of the namespace being brought in with a using-directive go up, regardless of the common style guidance to never do so.

Types can conceivably be aliased into a readable and terse form (although in practice they often are not):

```
// can rename to capture info from namespace
using fspath = std::filesystem::path;
```

but free functions cannot

```
using std::filesystem::copy;  // cannot rename
```

So here is the crux: if **adding** contextual information (nested namespaces) increases the odds that a name will often be used **without** it, we are not serving readers well by relying on that additional context.

We would be better off choosing unique names that are clear without the additional “clues” provided by the nested namespace.

As it turns out, in general we already have done so: `time_point` is clear, `duration` is clear, `path` is (pretty) clear. Most of the free functions in `filesystem` follow the naming of POSIX free functions: even if those are potentially ambiguous, their meaning will never be surprising or hard to discover. Additionally, in the context of operating on paths and file permissions, they turn out to be clear from context at the call site.

The writer side of the equation:

C++ name lookup is not limited by adding additional levels of hierarchy: your lookup isn’t limited to the innermost namespace, it expands outward into all parent namespaces. As such, unlike languages like Java where fine-grained package groupings give you protection against accidental lookup, nested C++ namespaces give false confidence in the isolation.

Worse, every nested namespace we choose causes conflict with any name lookup on symbols that aren’t fully qualified. By choosing to add `std::filesystem`, we have added potential conflicts for any codebase that has a `::filesystem`, or `acme::filesystem`, etc.

Even if our standard library implementers know enough to qualify consistently and mitigate these risks, **our users do not**. Following the precedent of the standard, users will be enticed into adding nested namespaces and expanding the set of possibilities for name collision.

For example, following the standard’s precedent it seems perfectly reasonable for Acme to introduce nested namespaces to separate different problem domains. The testing subgroup at Acme could easily introduce a namespace `acme::testing` to use for their test utilities. As soon as they pick up a dependency on a test framework using `::testing`, any use of any (not-fully-qualified) name from within `acme::` becomes ambiguous - build breaks will happen because of innocuous addition of `#includes`. This is not theoretical, Google has been fighting related problems internally for 5+ years - collisions among names in the namespace tree cause problems, and one simple way to resolve that is to ensure uniqueness by ensuring the tree is flat. (Notably, Bloomberg has another mechanism, but existing design of `std` makes that comparison irrelevant.)

The guidance for users should be single-level broad namespaces - namespaces are best used to disambiguate between the local project and imported projects. Because of the promiscuity of name lookup rules, nested namespaces should not be used as an attempt to introduce partitions in the current project - it doesn’t work that way.

The practice of relying on nested namespaces in the standard isn’t helping readers and sets bad precedent for writers - we should stop following this precedent. We can leave the option available to us for use in some unusual circumstances, but it should be discussed carefully and used sparingly. In particular, the assumption of nested namespace availability to lend meaning / semantic disambiguation to APIs should be avoided.