Doc. no.: P1863R1
Date: 2020-01-09
Reply to: Titus Winters
Audience: DG, WG21

ABI - Now or Never

Note: For a complete introduction to ABI, estimates of value, lists of things to fix, consequences, and a suggested mechanism for breaking, see P2028. LEWG and EWG committee chairs expect to have a joint session on these two papers in Prague.

For the past few years, I've been advocating for WG21 to prioritize progress over backward compatibility. I'm losing faith in that position, especially when it comes to ABI. The past 3 releases (C++14, C++17, C++20) have been as ABI-stable as we can manage. Even if WG21 chooses to make C++23 an ABI break, we'll have provided binary compatibility on many platforms for more than a decade. In my experience making broad changes to software systems, Hyrum's Law[1] dominates. An untold number of users have now baked-in assumptions (wisely or not, explicitly or implicitly) about the ABI stability guarantees of the standard library: perhaps as many as half of all C++ devs globally.

I have been keeping a list of things that WG21 should fix if we decide that we're taking an ABI break. I cannot argue in good faith that the combined value of that list alone compares to the ecosystem cost of an ABI break. We'll get many small improvements in API consistency, standard library code quality, etc, but there's certainly no headlining feature that makes the cost worth it for the average user. We may even get some conformance gains, giving library implementations that are currently out-of-spec the chance to resolve those issues. But there is no single feature in my list that is clearly worth it.

More critically, there is a non-trivial amount of performance that we cannot recoup because of ABI concerns. We cannot remove runtime overhead involved in passing `unique_ptr` by value[2], nor can we change `std::hash` or class layout for `unordered_map`, without forcing a recompile everywhere. Hash performance has been extensively researched for years now, and between table lookup optimizations and hash improvements, we believe we could provide an API-compatible `unordered_map`/`std::hash` implementation that improves existing performance by 200-300% on average[3]. This is disallowed by ABI constraints. Additional research on optimization and SSO-tuning for `std::string` is also assumed to be worth a non-trivial performance boost (1% macrobenchmark/fleet performance) - this has no API impact, but is disallowed by ABI constraints.

All known performance concerns that are blocked solely by ABI easily add to a performance penalty of a few percentage points - perhaps 5-10% aggregate. That overhead is not make-or-break for the ecosystem at large, but it may be untenable for some users (Google among them). It is certainly more overhead than C++ strives for - remember that this is the language that claims to leave no room for a more efficient language. Most users of C++ do not appear to be deeply concerned by this performance loss: there are other hash maps available for those users that need absolute performance. Few workloads

[1] http://hyrumslaw.com
[2] https://www.youtube.com/watch?v=rHIkrotSwcc
[3] https://github.com/google/hashtable-benchmarks

are dominated by the ambient inefficiencies involved in passing `unique_ptr` by value or other language-level ABI issues. Organizations that need absolute performance can (and do) go their own way with non-standard libraries and non-standard toolchain configuration. If that's the expected outcome, we should be clear about that.

By comparison to those few organizations, far more users will be impacted by an ABI break. I suspect that many of those users may not even know how deep their dependence on ABI goes. Google's server ecosystem builds nearly everything from source, has few external dependencies, and has better-than-average ability to undertake large refactoring tasks. Even for us, a recent ABI-breaking standard library change cost us 5-10 engineer-years. The aggregate cost of an ABI break across the entire C++ ecosystem should conservatively be estimated in **engineer-millenia** - coordinating the rebuild-from-source efforts for every provider of a plugin, .so, or DLL in the world will require massive human effort. Coupled with the ecosystem disruption of C++20 modules, an ABI break in the C++23 timeframe may severely fracture the ecosystem.

Bound up in this discussion are a number of impossible-to-answer questions. How long can we expect to proceed before an ABI break is not merely valuable, but critical? If we explicitly adopt a path of ABI stability, how expensive will it be for the ecosystem when such a criticality arises? If a security problem like Spectre or Meltdown had required a change to calling convention for functions, what would it take to for C++ to overcome that? What fraction of users are using C++ because we claim to prioritize performance above all else? Perhaps more concerning: how long can C++ claim to be the highest-performance systems language while leaving these optimizations untouched?

If we cannot or will not break ABI intentionally, we should be very public about that decision. We should be clear that this is a language that prioritizes ABI stability over the last few percent of performance. I'd argue that has been true in practice for a few years at this point. We should let users know what to expect from us, and let them know that it is expected that libraries like Boost, Folly, and Abseil are the right tools for library performance. This does nothing to address language-level ABI concerns, like the overhead in passing `unique_ptr`. In such a model, the standard library is still valuable: the standard library is what you use for compatibility and stability. This might suggest a change in focus and direction for the standard library - we may want to be designing for flexibility in the face of changing requirements, rather than pure performance.

If we instead argue that performance dominates ABI stability, we should decide immediately when we will take that break and do everything possible to make it easy for the ecosystem to adopt. If we go that way, we also need to be very loud about that. We have to recognize the fact that the longer we go between breaks, the more expensive they are - more unsupported dependence on ABI stability creeps in over time. Our implementers have made it very clear that the breaks in C++11 were expensive and painful. It's natural to avoid repeating those costs, but we can choose whether we avoid that repetition by not breaking ABI or by *making it less expensive*.

Fundamentally, I think that there are 3 real options for WG21 on this question.

1. Decide on a release to be an ABI break, be that C++23 or C++26. Give people warning, and produce tools and diagnostics to help identify things that will break at that point. Focus on a more sustainable practice (and tooling) for ABI breaks going forward. It will not be in any single implementer's best interest to force an ABI break for their users if the other implementations are not doing so - this has to be a coordinated effort for the good of future users. Ideally we should break *everything* - make it clear and easy to understand that code compiled in C++23 mode is

not compatible with code compiled in earlier modes. If some users can avoid rebuilding and others suffer link-time or run-time bugs, that will increase confusion and frustrating.
2. Decide that we are committed to ABI, formalizing our current practice. It's been the case for years that implementers effectively have a veto on ABI breaking changes - we have already been prioritizing ABI above design or performance concerns. If we admit that and make it clear to users, that is a better state for the ecosystem. Utility libraries will take on greater importance for users that need every last drop of performance, but don't need the stability of the standard. We risk a long-term challenge from other systems languages that focus on performance.
3. Fail to pick a path, continue with the status quo. For me this is the worst case scenario: we continue to prioritize ABI concerns without being clear about that. We say "performance" but vote "ABI". This dissonance is harmful for the ecosystem, and suggestive of a lack of basic agreement on priorities for the language. I sincerely hope that we can find a good consensus, driven forward by implementers and the DG.

I believe that outcome #1 is best for users that demand absolute performance, but it comes with incredible ecosystem costs and may further fragment the language. Outcome #2 is the boring, responsible, proper choice - coming to the sad admission that we have painted ourselves into a corner and are making the best of it. Option #3 is the sort of leadership failing that I pray we can avoid: any explicit choice is better than the current dissonance and failure to agree on long-term priorities.

I understand that we have arrived here by incrementally-sensible inaction. No individual change in the past decade justifies an ABI break, but a change to the implicit compatibility policy now will be devastating to the ecosystem. However, if we don't do that we are also starting a different sort of end-of-life for C++: we cannot be the performance-oriented systems language if there is so much room for a higher performance language. In theory, each vendor could decide individually to break ABI with any future release, but the general sentiment seems to be going the other direction. I believe we must discuss and find consensus across implementers and WG21: what do we truly prioritize?