Doc. no.: P0684R0
Date: 2017-06-19
Reply to: Titus Winters, Bjarne Stroustrup, Daveed Vandevoorde, Beman Dawes, Michael Wong, Howard Hinnant
Audience: C++ Standards Committee

C++ Stability, Velocity, and Deployment Plans [R0]

Problems with stability, velocity, and deployment of the C++ programming language as it evolves are identified. Policies are proposed to mitigate those problems.

Introduction

Over the past few years, the committee has increasingly demonstrated a lack of agreement on priorities. As a result, many of the following questions have arisen in committee discussions:

- Is C++ a language of exciting new features?
- Is C++ a language known for great stability over a long period?
- Do we believe that upgrading to a new language version should be effortless?
- If so, how do we reconcile those effortless upgrades with a practical need to evolve the language?
- If we instead prioritize stability over all else, are we bound to move slowly - only making a change when we are certain it is correct and will never need future fixes?

It seems that members of the committee (and indeed the authors of this paper) have held differing (perhaps even inconsistent) positions on these questions. In this position paper we aim to discuss the themes we've seen and heard on this topic in committee meetings, how those match our actual experience, and propose a concrete policy that the committee as a whole could adopt providing both rights and responsibilities for users of the language.

In so doing, we hope to change from the normal state to an improved state where the committee has a more coherent direction, rather than Brownian Motion in action.

Present Policy and Practice

While there is no concrete policy written down as part of the standard itself, we believe we could benefit from such a policy and more specifically, many committee members would agree that the current policy is roughly:

> New versions of the standard will not change the behavior of existing well-behaved user code. In rare instances a standard change may cause build breaks, and silent changes should be even rarer. All changes should in principle be statically detectable. ABI changes should be avoided as long as possible, and be as infrequent as possible.

Note that much hangs on the definition of "well-behaved user code." Currently the standard carves out a few things as undefined behavior (Please see SG13, Undefined Behavior) with the intent of allowing for future changes without breakage. This includes (but is not limited to) "taking the address of a member function of a type from namespace std is UB", as well as more commonly-known things like broad prohibitions on adding names to namespace std. Do these prohibitions suffice to avoid breakage? Of course not. In many cases they are poorly publicized, but more fundamentally we make changes that could even break those users that do obey the few published prohibitions. For instance, users may still rely upon "using namespace std" and write in the global namespace - any overlap in new names in std and their global namespace is an immediate break. Using the global namespace is not forbidden. Nor are using directives. What gives us the "right" to break such code?

More subtly, with SFINAE and existing template metaprogramming facilities, there is sufficient introspection in the language already to (in theory) have runtime/semantic dependence on nearly any library or language feature. Nothing in the standard declares it to be UB or otherwise gives the committee the right to break code that relies on things like "vector::emplace_back returns void" but you can certainly imagine constructing code that relies on that through metaprogramming trickery. We pretty clearly do not consider that behavior "in bounds" - metaprogramming users do not have an expectation of free upgrades between language versions. What gives us the "right" to break such metaprogramming code?

In a completely different vein: When was the last time that your organization took a compiler update without having to fix your codebase? Even a minor compiler release tends to require some effort, including extensive testing. Changing standard libraries is often near impossible - at best it is a significant effort. What makes us believe that enabling a new version of the language is going to be any less effort? Why are we worried about breakages that are likely lower-cost than updating compiler versions?

Proposal - Velocity, Deployment Plans

We suggest that the level of concern that the committee currently holds for breakage may be ill-founded - the cost paid during a compiler update stemming from user error is already significant. We should recognize that all compiler/standard library updates come with some cost - so long as any intentional breakage on our part is expected to be dominated by those existing costs, we are still serving our users well.

Procedurally, a change to the toolchain (be it compiler update or language version change) is a task that our users are already generally tackling as an explicit task — someone or some team is assigned to do this. For language updates, that upgrade period is the exact time that we want to provide diagnostics and refactoring tools designed to make the upgrade process low-risk and low-effort. Consider the cost reduction for users if we expect them to use the following process when updating:

1. Update to a current version of the compiler without changing language versions. For example, update to a current version of gcc while staying at C++14, rather than updating to current gcc and updating to C++17 mode at the same time.
2. Turn on diagnostics in that current compiler version to identify places where behavior will change in the next language version.
3. Evaluate those changes, resolving as necessary. For risky behavior changes to the language or the library, this requires providing C++(n-1)-compatible mechanisms to opt-out of the change. For example, in the case of synthesizing operator <=>, explicitly deleting a comparison operator would suppress the new behavior in the new language mode.
4. Turn on the new language mode.

This isn't a magic wand — we will still need to find ways to have pre/post compatibility and consider cases where the necessary updates cannot be done in sync (libraries with varying release schedules), but it is potentially a significant power that the committee is lacking right now.
In such a world, we argue that it is far easier to make substantive changes to the language. For example, if we decide that synthesizing operator <=> is the superior long-term design, we can ask that compilers in C++(n-1) mode provide a diagnostic for types that will have their behavior changed by the new synthesis with an easy mechanism to opt out (=delete). This hopefully allows for a smooth rollout of more-invasive changes than we currently feel safe accepting. In such a world, we can perhaps stop arguing about the safety of such a change and focus more on the long-term value of it and the safety of the deployment strategy. In trade, we have to recognize what is already true: upgrades are not free. If we focus on understanding and minimizing the upgrade cost, we can find a balance between stability and velocity that keeps most users happy.
This would effectively update the compatibility promise of the standard relative to the earlier phrasing:

> New versions of the standard will not change the behavior of existing well-behaved user code. In rare instances a standard change may cause build breaks, and silent changes should be even rarer. **All behavior changes should be statically detectable in the previous language version and have previous-version-compatible opt-out mechanics to preserve existing behavior.** ABI changes should be avoided as long as possible, and be as infrequent as possible.

In more detail, we can consider the following examples:

- Adding a keyword – easily detectable, old code breaks if it uses the new keyword as an identifier. This generally cannot change meaning of old code, it can only make it ill

formed. This is easy to detect in a C++(n-1) mode - an engineer updating a codebase to C++(n) only has to find-replace places where the new keyword is used and use a different identifier.
- Adding a public data member to a standard-library class – easily detectable by use, but could hide public member from a base class. Can change meaning of old code.
- Adding a private data member to a standard-library class – detectable (SFINAE or sizeof), but the set of private members is unspecified, so relying on a specific layout is UB (e.g., debug and optimized versions may differ). No diagnostic required.
- Changing the signature of a standard-library function call (e.g., changing the return type from void to non-void or changing an argument type from int to double) – detectable but it could change the resolution of a call. Can change meaning of old code – mostly through implicit conversion to standard / fundamental types. Can be identified in C++(n-1) mode, and opt-out is clear – provide explicit conversion to the C++(n-1) format, presumably ensuring compatibility.
- Changing the meaning of a standard-library function call – not detectable. Can change meaning of old code. Don't do that without a rollout plan.
- Changing the constraints of the implementation of a standard-library function call – not detectable. To do that we'd need a trait, feature macro, or other trickery. (Generally we all know not to add constraints to existing APIs.)
- Adding an overload to the standard library – detectable. Can change meaning of old code. We'd want a warning for each call for which the overload resolution changed.
- Changing rules (e.g., lookup rules or adding generated functions) to allow more candidates for overload – detectable. Can change meaning of old code. We'd want a warning for each call for which the overload resolution changed.

Building a collection of examples would make it easier to use the proposed rule for clear user requirements.

Proposal - Clear User Requirements

However, in some cases the above does not suffice: we have still not addressed our lack of guarantees for what types of breaks are considered relevant. Let's consider the simple case of std::accumulate and Peter Sommerlad's p0616. As specified, std::accumulate is O(n^2) when invoked on anything with an O(n) operator+, like std::string. This is because the accumulated object is copied at each step, rather than moved - by specification. The only code anywhere that can be broken by making changes to that specification is code with user-defined types where move is not an optimization of copy - and yet we are fearful to proceed. A "safe" deployment plan for this issue would likely require two language versions: one to introduce a std::accumulate variant that explicitly chooses the existing copy behavior, and a second to change the default - and with such a plan we are still left with both versions indefinitely. Or, we can collectively act boldly and say "Anyone broken by this change deserves it", which is likely true. Still, it would be more satisfying and healthy for the community in the long term to provide

clear guidance - what exactly is it that we expect from users? Why do they "deserve it" to be broken by such a change?
We believe the community would be well served if we defined what is expected of "well-behaved" user code. With respect to the std::accumulate example, this might include "The standard library assumes that if a user-defined type has both move constructor and copy constructor, the move constructor is semantically equivalent and no less efficient." More generally this would include discussions of namespace std and metaprogramming/SFINAE guarantees (specifically, our lack of guarantees). Concepts could address many such examples.

This would effectively update the compatibility promise of the standard:

> New versions of the standard will not change the behavior of existing well-behaved user code **as defined by (forthcoming rules)**. In rare instances a standard change may cause build breaks, and silent changes should be even rarer. All changes should in principle be statically detectable. ABI changes should be avoided as long as possible, and be as infrequent as possible.

The particulars of the rules for well-behaved user code can (and should) be discussed at length, but could include some or all of the following. Note in particular that users of the standard are not special in this: abuse of these rules is generally bad usage for any library.

- Do not define names in namespace std (or namespaces of libraries you do not own), except as specifically directed for library extension points.
    - Defining names in foreign namespaces is a break when a real API of that name is added. Preventing addition of new entries in libraries you depend on is counterproductive.
- Do not add things to the global namespace especially if you have a namespace-scope “using namespace std” directive in your code.
    - This would also prevent introduction of new names in namespace std.
- Do not forward declare symbols from namespace std.
    - Forward declarations require that the signature of that API does not change - adding new default parameters to functions or new default template parameters to template definitions will be a build break.
- Do not take the address of functions or member functions in namespace std. More generally, do not depend on the signature of standard APIs - assume only that it is callable as specified.
    - Adding default parameters or new overloads will break code that does this.
- Do not depend on the presence or absence of APIs via template metaprogramming methods, nor metaprogram properties of standard types (layout, size of, alignment of).
- If a user-defined type defines both copy and move operations, move should be semantically equivalent to copy on the target AND no less efficient than the copy.
    - If the standard optimizes library operations or additional language rules to make better use of move, this should be good for everyone. If you define exotic types where move is less efficient or semantically different than copy, you will be broken by such changes.

Proposal - Our Promise To Users

AKA The C++ Programmers’ Bill of Rights

We, the ISO C++ Standards Committee, promise to the best of our ability to deliver the following, assuming user code adheres to the above

1. Compile-time stability: Every change in behavior in a new version of the standard is detectable by a compiler for the previous version.
2. Link-time stability: ABI breakage is avoided except in very rare cases, which will be well documented and supported by a written rationale.
3. Compiler performance stability: Changes will not imply significant added compile-time costs for existing code.
4. Performance stability: Changes will not imply added run-time costs to existing code.
5. Progress: Every revision of the standard will offer improved support for some significant programming activity or community.
6. Simplicity: Every revision of the standard will offer some simplification of some significant programming activity.
7. Timeliness: The next revision of the standard will be shipped on time according to a published schedule.

Note “to the best of our abilities”. These are guiding principles, rather than executable statements. For example, if I add a function to a header file, the compilation of code that includes that header will slow down imperceptibly. That’s understood and could be argued not to be “existing code” because of the change. Adding enormous amounts of code to a header so that compilation slowed down noticeably would be another matter.