Doc. no.: P0684R1
Date: 2017-10-11
Reply to: Titus Winters
Audience: WG21 (Full C++ Standards Committee)

C++ Stability, Velocity, and Deployment Plans [R1]

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 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 the prior revision of this paper and the discussion in Toronto on that topic we sketched out several areas where we believe there is room for improvement in committee behavior. This paper assumes familiarity with that previous revision (rather than trying to incorporate concrete details and proposals into the body of and already long and complex document).

Specifically, this paper proposes the creation of two additional standing documents, and one substantive change in committee behavior. As per usual, the SDs require a plenary vote. As an unusual step, we also suggest a plenary vote on the proposed change in committee behavior. We also recognize that some relevant change in that behavior was already witnessed during the Toronto meeting - we would like to establish a clear consensus from the committee as a whole (especially the implementers and maintainers of legacy codebases).

Proposed Standing Document: Committee Goals

As per the section “Proposal - Our Promise To Users” in P0684R0 - we propose a new standing document that clearly identifies the overall goals of the committee and what we intend to provide for users. This document should be useful in setting user expectations, and also internally for committee discussions when we are weighing conflicting values and trade-offs.

As described in R0, the contents of this doc would focus initially on what we aim to provide, and may evolve over time to detail what goals the committee agrees upon for the language as a whole. This is not expected to be perfectly binding, but should provide some common basis for grounding our discussions and setting user expectations.

Proposed Standing Document: Compatibility Guidelines

As per the section “Proposal - Clear User Requirements” in P0684R0 - we propose a new standing document that clearly and concisely identifies the major types of behavior in user code that we do not guarantee to support across language versions. Currently the standard calls out some of these behaviors, usually by resorting to UB (as in [namespace.std]). However, it is not clear which rules are in place for the purpose of reserving space for future standards and which are merely in place to guarantee that the standard library functions properly (malicious injection of names into std could in some cases lead to erratic behavior).

As described in R0, the contents of this doc would focus on the minimum set of behaviors a user should avoid if they want upgrades between language versions to be easy, given the arguments that are currently common in committee. See that revision for more examples.

Once this new SD is developed (and approved by an initial Plenary vote) it should be popularized within the C++ user community. Further refinements to these guidelines should be governed by EWG/Core/LEWG/LWG in the normal and appropriate fashion.

Proposed Approach for Evaluating Change Safety

As suggested in the previous revision of this paper (and as was starting to happen naturally during the Toronto meeting after the presentation of such) - The Committee should be willing to consider the design / quality of proposals even if they may cause a change in behavior or failure to compile for existing code. Rather than consider the effect of every new language version impacting code all at once, we should be aware that the act of deploying a new language

version is likely already time-intensive - even an ordinary update to compiler versions often requires significant effort.

We should assume that users are will upgrade by:

- upgrading to a version of their compiler that supports C++n
- engaging diagnostics in C++(n-1) mode to warn of impending behavior changes
- evaluating those diagnostics / modifying their code as necessary
- turning on C++n mode

This requires additional diagnostics from implementers, and additional care from users, but potentially unblocks significant avenues for improvement in the language at a the cost of acceptable upgrade cost for users.

Examples

It is important to note that the examples provided here are provided on the basis of feasibility and illustration, and are not concrete proposals. Please consider in terms of behavioral change, detection, and the possibility to opt-out in a compatible fashion. The general purpose of this proposal is to allow us to focus on “Do we like this change” and separately “How safe is it to make this change / how difficult will it be for users to adopt to this?”. Otherwise we are forced to hold every possible change against the “no behavioral difference ever” and we debate more on “is this safe” rather than “is this good, and safe enough.”

Safest Change: No behavioral difference

Of course, it goes without saying that the changes we currently value most will continue to be considered best - changes to the language that have no potential for impact on existing code. (That of course presupposes that user code is well-behaved as per the newly proposed SD on compatibility guidelines.)

Easiest-to-Adopt Change: Statically detectable difference with previous-version avoidance options

The best newly-acceptable option: a behavior change that can be statically detected and that has a options available in C++(n-1) that can make every existing instance conforming. That is: it is easy for an engineer performing the language-version upgrade to verify that nothing is affected in the end, as all sites have been modified.

For example: we decide that we want new keywords to support coroutines. In order to support that, we choose to make “await” a keyword.

- Detectable: In C++(n-1) we can issue a warning for uses of the new keyword in existing code (as variable names/type names/function names/etc).

- Opt-in: N/A
- Opt-out: Basic find+replace functionality will generally suffice. For comparison purposes, reclaiming a keyword like await in Google's codebase would require edits to 17 files. In theory there may be cases where this becomes difficult or impossible for any given user - if they have promised ABI compatibility and the affected keyword appears in their ABI, this may be hard to resolve. (We should, of course, weigh the likelihood of these issues against the gain for the whole C++ community / Standard quality.)

For example: In C++20 we decide to make the assignment/initialization + conditional ill-formed, codifying the existing common warnings and instead relying on if+initializer syntax from C++17. That is:
if (int i = Foo()) {
would become an error in favor of the new syntax:
if (int i; i = Foo())
or without declaration:
if (i = Foo())
would require the existing solution:
if ((i = Foo())

- Detectable: We can clearly issue a warning for this (we have done so for years in most/all compilers)
- Opt-in: Every existing instance can be converted to one of the two alternate syntaxes with no behavior change, in C++17 mode.
- Opt-out: N/A

Feasible-to-Adopt Change: Statically detectable difference w/ previous-version opt-out

The most common newly-acceptable option: a behavior change that can be statically detected and that has a syntax available in C++(n-1) that can opt-out each potential instance of that change. This has somewhat more cost than the previous, as each site of the impact needs to be tracked and evaluated, but some (many) will remain in any given codebase when switching to C++n. This leads to increased but tractable tracking problems for those performing the language upgrade, scaling with the number of affected instances.

For example: we decide that synthesizing operator <=> for some set of classes is a preferable direction for the language. A user-defined operator< is assumed to suppress that generation.

- Detectable: In C++(n-1) we can issue a warning for classes that will be impacted by this change. (Obviously the compiler can statically determine if the class declaration makes it eligible for the synthesis.)
- Opt-out: a type owner can opt-out in a backward-compatible manner by adding something like:

bool operator< (const MyClass&) const = delete;
This would not change the behavior of the type in the previous language version, and relies only upon previous-version syntax.

For example: we decide to stop synthesizing copy/assign special member functions in the presence of a user-defined d'tor. A user-provided declaration (with =default) keeps the generation intact.

- Detectable: We can issue a warning in C++(n-1) for classes where this synthesis would be changed.
- Opt-out: Add the relevant =default explicitly. (Again, no change to behavior in the previous language version.)

Potentially-Expensive-to-Adopt Change: Statically detectable difference without opt-out

There may exist changes that we decide we wish to make where there is no reasonable change that can be made to opt-out. These should generally be avoided, but may be decided to be worth it if the expected outcome is fewer surprises/bugs in the long-term (or vanishingly few impacted locations). Every instance of the resulting diagnostic will need to be (manually) evaluated by those performing the language version upgrade.

For example: we decide to change overload resolution rules to consider template specializations in the overload set (see the motivations in P0551R1).[1]

- Detectable: Although it might be expensive, we can issue a warning in C++(n-1) when performing overload resolution between a function and a function template that has specializations.
- Opt-out: There is no obvious mechanism (at least none that I can think of) to generally say at any given call site which of those overloads to pick, especially not in a generic context.

Potentially Dangerous Change: Runtime Behavior Change without diagnostic

We should obviously not increase the amount that we do this to well-behaved code, regardless of whether we accept the rest of this proposal.

We should consider doing this to code that violates our description of well-behaved, as per the proposed SD on Compatibility Guidelines. For example, if we suggest that move-constructors

[1] Remember: I'm not suggesting we necessarily do this, I'm providing this as a thought-experiment example to categorize types of changes.

are always assumed to be no-worse than copy-constructors, changing the behavior of std::accumulate to prefer move accumulation is warranted. (This would of course be a behavioral change for any user-provided type that has worse performance on move or different semantics for a type that is moved-into instead of copied-into.)