Doc. no.: P0921R0
Date: 2018-02-01
Reply to: Titus Winters
Audience: LEWG, EWG

For a sufficiently clever user, effectively any change we make to the standard library will be a breaking change. In a few instances, we are clear about what users are not allowed to do ("do not take the address of member functions of standard types", "do not add new names into namespace std") - those restrictions are not generally understood, are incomplete, and leave completely unclear **why** those rules are in place. In keeping with general discussions of compatibility, and as mentioned in P0684, I'd like us to be clear to users about what does and does not constitute acceptable use of the C++ standard library. What behaviors are off-limits to users in order to ensure that we have as much flexibility as possible when making changes to future standard library revisions?

What follows is an initial proposal for such a list of user restrictions, largely drawn from P0684r0. I hope for three things from the committee:

- LEWG and EWG will discuss and agree upon the contents of this list, and populate a new Standing Document with the result of that consensus.
- LEWG and EWG will in the future ignore arguments predicated on, "But users could be depending on X!" for X on this list.
- Committee members should work to disseminate the understanding of such an SD to the public - this only works in practice if the C++ community understands the trade-offs in question.

Note in particular that users of the standard are not special in this: abuse of these rules is generally bad usage for any library.

Primarily, the standard reserves the right to:

- Add new names to namespace std
- Add new member functions to types in namespace std
- Add new overloads to existing functions
- Add new default parameters to functions and templates
- Change return-types of functions in compatible ways (void to anything, numeric types in a widening fashion, etc)
- make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions. Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

As a result of the above reservations, users must obey the following (non-exhaustive) list of restrictions, or be willing to invest (potentially significant) effort when upgrading between versions of the standard library.

- Do not define names in namespace std (or namespaces of libraries you do not own). Do not define anything in namespace std except as specifically directed for library extension points (specializing `std::hash` with your own types).
    - 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.
- Similar to above: do not define supplemental APIs for standard types - don't define iostream operations, swap, etc for standard types that do not already support those.
    - This can manifest as a build break, ODR violation, or behavioral change in the future if the standard defines those APIs for those types.
- Do not add `using namespace std` (or any using directive for sub-namespaces of std) to your code.
    - This would also prevent introduction of new names in namespace std, because any addition to std can be a build break if the chosen name matches any existing name in that namespace. (This is especially troublesome with the global namespace and any C libraries.)
- Do not forward-declare names 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.
    - Similarly: do not use a deduction-guide inferred declaration of `std::function` initialized from a standard API. Eg: `std::function f = std::mt19937();` This may break in the same fashion if overloads are added.
- Do not depend on the absence of APIs via template metaprogramming methods, nor metaprogram properties of standard types (layout, size of, alignment of, triviality), except where specified by the standard.
- Do not rely on ADL when calling any function involving standard types, except as follows:
    - Any operator (consider `operator<<` and `operator>>`, or `operator+` for string)
    - swap (intentional in the design for swap)
    - (Is there anything else we can think of that is expected to be called by user code via ADL?)

  Unqualified lookup that relies upon ADL can be broken by the standard adding new APIs to std.
- Always rely on ADL for lookup on operators.
    - Don't call `operator +("a"s, "b"s);` or `”a”s.operator +(“b”s)` - it's silly and we're not planning for it.

- Similarly: do not make unqualified calls to functions in the global namespace when calling any function involving types from namespace std (or any namespace within std).
- If a user-defined type defines both copy and move operations, move must have the same post-conditions for the logical state of the target as copy does AND be 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.
- Names beginning with leading underscores, as described in [lex.name], are off limits.
- Do not add a nested namespace called “std” anywhere.
    - There are numerous scenarios where future additions to namespace ::std would become ambiguous in the face of nested std.