Doc. no.: P2028R0
Date: 2020-01-07
Reply to: Titus Winters
Audience: LEWG, EWG, WG21

# What is ABI, and What Should WG21 Do About It?

A short refresher on what Application Binary Interface (ABI) compatibility entails, a list of known proposals that have been dropped due to ABI compatibility concerns, and a summary of the critical issue: should we make C++23 an ABI break, or should we commit to ABI stability going forward? (This paper is similar to P1863 but provides much more introductory material and concrete suggestions.)

## ABI and WG21

Formally, ABI is outside of the scope of the standard. WG21 does not control the mangling of symbols, the calling convention, unspecified layout of types, or any of the other things that factor into ABI. This is historically well-reflected in the voting pattern at the heart of the issue: implementers have historically held a veto on any WG21 proposal, in the form of “We won’t implement that.”

Practically speaking, if all of the major implementations are unanimous on ABI concerns, this quickly stops being a WG21 issue. We cannot force implementations to implement features that they (and their users) do not want. On the other hand, WG21 members can also individually choose to not be constrained by this precedent.

My belief is that short-term user and implementer demands should be weighed against long-term needs and ecosystem health, and WG21 participants at large should better understand the tradeoffs in our current informal policy. If we want C++ to be the preferred high-performance language through the 2020s, we need to be able to argue that performance concerns are taken seriously. If we want to keep adoption simple and remove barriers to adoption, we should double down on the status quo and formally promise ABI stability.

In P1863 I’ve tried to briefly lay out the paths forward. This paper is a longer and more detailed discussion for those that may not have as much familiarity with ABI and what is at stake.

# What is ABI?

ABI is the platform-specific, vendor-specified, not-controlled-by-WG21 specification of how entities (functions, types) built in one translation unit interact with entities from another. This covers a wide range of topics, including but not limited to:

- The mangled name for a C++ function (Functions in extern “C” blocks handled differently).
- The mangled name for a type, including instantiations of class templates.
- The number of bytes (`sizeof`) and the alignment, for an object of any type.
- The semantics of the bytes in the binary representation of an object.
- Register-level calling conventions governing parameter passing and function invocation.

## Name Mangling

Name mangling governs how we interact with C and rely upon C-era linkers. Since my background is primarily on Linux systems, I’ll describe things in terms of the mangling and linking used by gcc and clang: the Itanium ABI.

For a declaration of a C function `void c_func(int)`, when a `.o/.a/.so` needs to call into that function, the call is left in symbolic form in the object code. In order to ensure that the right code is called when a program is fully linked and loaded, the compiler leaves those external calls in symbolic form as a call to `_c_func` or similar. During linking and loading, the final address of `c_func` is replaced accordingly.

C++ functions are more complex: we allow for overloading and namespaces, so the name of the function itself doesn’t suffice to resolve the jump location in the executable. As a result, both the parameter list and namespace of the function need to be included in the mangled name. In order to support these requirements and minimize name collisions with functions in C, C++ manglings are all prefixed specially with `_Z`, followed by an encoding of the (possibly nested) namespaces, followed by an encoding of the parameter types. Special encodings may be applied to common parameters from the standard library (`std::string`, for example, has a special shorthand encoding as `Ss`, instead of something like `N3std12basic_stringIcSt11char_traitsIcESaIcE`). This has no semantic implication, but reduces the size of the resulting object code.

Borrowing an example from wikipedia,

```
namespace wikipedia
{
   class article
   {
```

```
   public:
      /* Mangles as: _ZN9wikipedia7article6formatEv */
      std::string format(void);
   };
};
```

Similar rules apply for types, class templates, and function templates. Primarily, “name mangling” can be viewed as a function for taking the name of an entity in C++ and producing a name for that function or type (after template instantiation) that can co-exist with C entities in a `.o/.a/.so`.

Changing the name of an entity, changing the parameter list of a function, or (sometimes) changing the set of template arguments for a class template is an ABI change, in that it results in changing the mangled name for that entity. In some cases such a change may be backward compatible: if a pre-existing binary reliably has the definition of the old symbol, then `.o` files built at different times with different definitions may work together[1].

When we discuss “ABI breaks”, some incompatibilities come from these sorts of changes to the resulting mangled names. For instance, consider the discussion of `std::scoped_lock<T...>` in C++17. We wished to extend `std::lock_guard<T>` to allow for a variadic set of heterogeneous mutexes. However, we could not change `lock_guard<T>` to `lock_guard<T...>` because on some platforms the name mangling algorithm for variadic templates is different than the mangling algorithm for class templates of fixed arity. Code built with `lock_guard<T>` would be unable to pass a pointer or reference to such an entity to code built with `lock_guard<T...>` on those platforms - an “ABI break”.

## Object Representation

Name mangling represents an important *syntactic* level of compatibility for object code: do two translation units compiled at different times agree on the mangled name for an entity? Equally important is the *semantic* compatibility for objects: does an object compiled at one time and stored into memory work properly when interpreted by code compiled at another time?

For example, consider `std::string` and the ABI break associated with C++11 for gcc. Before C++11, the libstdc++ implementation of `std::string` relied upon copy-on-write (COW) semantics and the class layout for `std::string` itself was merely a pointer: the actual data, capacity, size, and reference count were accessible at fixed offset from the pointed-to allocation.

C++11 disallowed the COW behavior. A new/efficient `std::string` implementation without COW has fewer indirections, storing all of the relevant information directly in the object (no

[1] Not always guaranteed, may be broken by ODR-style issues with static variables or stable address requirements.

control block) and possibly relying on small string optimization (SSO) to inline some of the data directly. Even a simple `std::string` is going to be roughly 3 words: a pointer to the data, and one word each for the capacity and current size of the data.

Building one translation unit with the COW string and then passing a COW string to a function that accepts `std::string` built with the SSO implementation may link (the mangled name has not changed, necessarily), but will fail spectacularly at run time. The COW string is only passing one word to the function. The function is reading 3 words and interpreting those as (perhaps) `(data, capacity, size)`. Run time bugs and memory errors are practically guaranteed. In the particular instance of the gcc update, the mangled name was changed using inline namespaces (more on that later), but even having two manglings present in `libstdc++.so` leaves options for user code to fail.

This will play out in the same fashion even if we aren't dealing directly with strings. A `vector<string>` will have a significantly different allocation size in the two modes. Iteration through that vector will have a different stride length because of a mismatch on `sizeof(string)`. User defined types that contain `string` will misbehave in the same way.

Changing object representation is obviously an ABI break. This includes functional no-ops like reordering fields, changing padding or object packing. The C++11 example of invasively changing the representation for `std::string` is still a cause of frustration for the ecosystem and standard library implementers, even today. Any distribution that is focused on backward compatibility may still be building against the old string ABI to provide compatibility for `.so`s from the 2000s that may have reliance on the COW implementation of `string`.

## Object Representation and Semantics

Importantly, it isn't only changes to object layout that may result in this sort of ABI incompatibility. Any change to the semantic meaning of the binary representation of an object in memory is an ABI break. Among other things, this means that we cannot change `std::hash` without an ABI incompatibility. The hash value for an object computed in one translation unit is embedded in how it is stored in an `unordered_map`. If that `unordered_map` is passed to a TU that has a different definition of `std::hash` for the key, then we will be unable to find that object in the map.

Even when the layout of the objects in question doesn't change, semantics are a critically important part of ABI. Both translation units must agree perfectly on the number of bits for every object and the meaning of those bits.

## Calling Convention

Finally there is the language-level ABI, which for these purposes largely boils down to the question of "How do we call a function that was built in a different translation unit?" What values

are passed in registers, how many registers can be used in that fashion, what type properties govern whether a type may be passed in registers vs. in memory?

The largest example we have for issues with the calling convention comes from passing `unique_ptr`. For the Itanium ABI (used on Linux systems, among others), any type with a destructor that is passed by value must be passed in memory: the address held in this `unique_ptr` must be written out to memory (and then potentially re-fetched in the calling function), rather than passed in registers as it would be for a raw `T*`.

Had implementers and the specifiers of the Itanium ABI agreed to specialize the handling of `unique_ptr<T>` passed by value, it could perhaps be no more expensive than a raw `T*`. That wasn't done before `unique_ptr<T>` was introduced, and thus now it would be an ABI break to change the calling convention without updating all callers and callees in sync.[2]

## It is "Just" a Binary Format

In many respects, ABI definitions are similar to the way we define a binary file format or network protocol. If the sender and receiver disagree on how to interpret a file or a network message, there is going to be an error. Those disagreements can be syntactic (framing, message size, structure) or semantic (is this sequence of bits a checksum or a size). Although relatively few of us have much hands-on experience managing ABI concerns (and thus ABI takes on an unnecessary level of mystery), the comparison to binary formats has been illustrative for many discussions in the past year. Every function is a server, every function call is a client. Clients and servers must be updated in lock-step, or not updated, or must be updated with great care to ensure that clients never transmit something to the server that will be misinterpreted.

Following this client/server comparison, it is noteworthy that many binary formats explicitly encode a version number in order to properly identify version skew issues. Our current ABIs may or may not have such identification, which makes ABI changes even more troublesome. Lacking versioning, we can't quickly identify version skew / ABI-break issues at runtime, and thus the cost of these problems goes up because of greatly increased user frustration and mysterious runtime failures.

# Why Would We Consider ABI Changes?

As discussed in P1863, the behavior of WG21 for several years has been to give standard library implementers an effective veto on any proposal that would break ABI. Even in cases where it is unlikely that a user would notice the ABI change, like the proposal for

[2] There is also a language issue surrounding the order of destruction of function parameters and the execution of `unique_ptr`'s destructor. For simplicity that is being ignored in this paper, but a complete solution to "`unique_ptr` is as cheap to pass a `T*`" would have to address that as well.

`lock_guard<T...>`, we have come down on the side of implementers wishing to avoid a reprise of the C++11 `std::string` difficulties.

I believe that, in aggregate, providing a stable ABI for the past decade costs C++ generally and the standard library in particular perhaps 5-10% aggregate performance cost[3]. It prevents many API changes, because of changes to mangling, object size, or memory representation. It precludes adoption of new optimizations for existing types[4].

For the past few years, I have been keeping an informal list of things to clean up in the standard library if and when we take an ABI break. The following is an incomplete list of optimizations and API cleanups that are precluded on ABI grounds. It is believed that most or all of these could be adopted without causing any new source breakage - we are vetoing these changes in favor of compatibility with a potential binary/.so/.dll/.a/.o that was built at some arbitrary point in the past.

- Change `std::hash`, memory layout, and default allocation sizes for `std::unordered_map`. More advanced storage and probing techniques (as seen in `absl::node_hash_map`) could result in microbenchmark improvements of 300-400% over most `unordered_map` implementations. Without the ability to change these interfaces, we can never provide a salted hash container or a hash that provides any default resistance to hash collision attacks. The aggregate result of the performance and security concerns around hashing are sufficient that I will not recommend use of `std::unordered_map` for anything other than toy programs.

  This is QOI: No API changes are gating on this, but currently it doesn't seem that implementers are going to break ABI without a motivating reason. The same efficiency gains could be had by adding new types instead. However, even if we implement a wholly new type to take advantage of newer techniques, research on hashing continues to provide new optimizations and improvements: this strategy suggests periodic addition of new types on an ongoing basis, which is messy and unpleasant.
- Update and optimize regular expression implementations. Current implementations of `std::regex` are often 2x slower than equivalent use of regular expressions in interpreted languages like Python and PHP, and are roughly 10-100x slower than equivalents in modern systems languages like rust or go. (stats)

  This is also primarily QOI, the same
- Make `lock_guard` an alias for `scoped_lock`, and deprecate the old name. In the single-argument case these classes are strictly identical, there is no need for us to have both.

---

[3] This is based on estimates for a variety of Google workloads given the aggregate improvements we saw for moving away from facilities from `std`, in addition to experimental results for upcoming ABI-incompatible optimizations.

[4] There is, of course, the option of introducing new versions of everything (e.g. adding `std::new_hash` and newer/faster `std::better_unordered_map`). This has many complexities and costs for the ecosystem as well.

- Make `vector<T>::push_back` return a reference to the element in its new location, to match `emplace_back`.
- Add `uint128_t` and `int128_t`. Since `intmax_t` exists and there are ABI dependencies on it, we can't really add larger integer types, even though these are becoming increasingly common to see (and optimize for) in the wild. This could potentially be hacked around, but even C is considering removing the requirement that this is actually the largest type for the same reason. We have similar long-term ABI issues lurking with `max_align_t` and the hardware_interference_size constants.
- Make `std::bitset` trivially-destructible. This would be a minor optimization improvement, but is disallowed because it would change the calling convention when passing bitset parameters.
- Allow vendors to remove `std::uncaught_exception,` which was theoretically removed in C++20 - as a free function, any vendor that ever provided this is likely still providing this because of ABI compatibility.
- Many changes in error codes and error handling are currently forbidden because of ABI - we cannot store more information nor change function signatures. (P1196, P1197, P1198)
- Allow any of the deferred / vetoed changes to our few virtual interfaces (iostreams, pmr memory_resource, etc). We are unable to change the number of virtual methods in any interface, because the vtable size/layout would change, which is an ABI break.
- Unify algorithms with operator and function object variants (LWG 1053)

Anecdotally, I've heard from several implementers that they have their own laundry lists of implementation improvements (bug fixes, optimization, cleanup, refactoring) to perform if and when there is an ABI break. Per-implementation cleanups of that form may additionally result in a meaningful performance improvement and reduction in complexity for implementers.

I'll also note that in meetings that have a higher-than-average attendance of newcomers, and in papers that are written by new attendees, we get more proposals that are dropped because of ABI concerns. This suggests that long-term committee members are already filtering proposals that would be an ABI break. The full scope of what may be improved given the opportunity to break backward compatibility with pre-compiled code is hard to estimate.

As I said in P1863, I don't believe that any one of these is worth the aggregate ecosystem cost of an ABI break. On the other hand, getting all of these and everything that we've been deferring unconsciously, may be worth it. Especially if we learn how to avoid this backlog in the future.

# Nesting and ABIs

There is a notion that has been floated that inline namespaces provide some ability for the standard library to avoid ABI breaking concerns. This is unfortunately not the case.

The basic principle for inline namespaces is to provide the ability for source-compatibility while adding extra differentiation in mangled names. For example, a standard library that utilizes inline namespaces can provide an inline “abi42” namespace inside `std`, which results in source referring to `std::string` and `std::sort` but changes the mangling for those names to derive from `std::abi42::string` and `std::abi42::sort`. A binary can then be stitched together with multiple versions of those symbols.

Unfortunately, this only works for direct usage of entities from `std`. User code that relies on these isn’t governed by the inline namespace of the standard library (`abi42`), and so (for instance) a user type that contains a `std::string` does not gain any ABI-resilience from this approach: a function compiled to accept a user-defined type that contains a string is still tied to a single ABI, and doesn’t have a clear way of customizing its mangling to move in sync with the ABI of the standard.

Another way to see this issue: version numbers apply to the entire client/server protocol, not individual fields in the protocol.

# Approaches to Breaking

Historically there have been two forms of ABI change that I’m aware of: the style used by MSVC and the Windows ecosystem (changing version numbers in DLLs, clearly failing to link/load binaries compiled against the wrong version, and forcing users to recompile) and the style used in the gcc/C++11 transition with the `std::string` breakage (providing one library that is compatible with both ABIs so that pre-existing code continues to work without a recompile).

I posit that the aggregate cost of the `std::string` break was exacerbated by a few main issues:

- Users coming from C have an expectation that ABI is stable (if they think about it at all) - we use the same tools and generate the same format of libraries and object files, but without promising a stable ABI we have subtly different rules. Without an MSVC-like all-or-nothing link-time failure, we also have no enforcement of those rules.
- Any code that didn’t rely upon unsupported uses of `std::string` (or any of the smaller ABI changes) worked just fine.
- Code that did rely upon `std::string` passing across an ABI boundary as a member of a user-provided type would compile, link, and load in general. Only when a `string` was passed across an ABI boundary and misinterpreted on the other side did anything go wrong, and the manifestation of that wrongness left very little to suggest the actual cause of the problem.
- Up until the C++11 change, passing these ABI-unstable types across ABI boundaries happened to work, even if it wasn’t promised to work. As always, Hyrum’s Law applies.

From this, we might suggest that it is less-costly and less-confusing to make such a breaking change if:

- It is easy to explain what does and doesn't work together.
- Mistakes are detectable at link / load time.

This is, if I understand correctly, what the Windows ecosystem has done historically. ABI breaks are annoying for users, but fairly easy to diagnose.

To that end, I suggest that the simplest answer for users is this: **We should change the mangling for C++23 symbols in a way that is not compatible with previous C++ manglings.** For instance, on Itanium we might change the introducer for C++ manglings from _Z to _Y or something similar[5].

This has some nice properties:

- It is easy for users to internalize. "I built with -std=c++23, everything I'm linking against has to also be built that way." The mental model is "C++23 requires a rebuild." This is particularly compelling if it is not platform specific.
    - Note that this is orthogonal to *source* compatibility requirements. We're only discussing changing compatibility with pre-existing object code. However, it is an all-or-nothing compatibility: precompiled libraries, plugins, etc will require the production of new versions before the adoption of C++23.
- It is easy to detect at link/load time. Small changes to linkers and loaders can identify the version skew by virtue of mismatch between manglings.
- Clever vendors can ship both forms (for Itanium, both _Z and _Y versions of every symbol, for example) in the same libraries, but can just as easily use the break as a chance to drop support for older forms.

The same approach can optionally be taken at a regular and well-advertised cadence going forward, be that at every 3 year release or once every decade. The MSVC experience suggests that users will complain about regular forced-rebuilds - whether we decide this is tolerable and necessary for ecosystem health and performance is up for WG21 to decide.

Anything more incremental, harder to explain, or less diagnosable than this is going to lead to user confusion and be a reprise of the `std::string` debacle.

# Alternatively, We Promise Stability

Alternatively, we can stop pretending that we are going to make ABI breaking changes. This will have the effect of changing the advertised nature of the standard library: there are things in the standard library that are meaningfully inefficient (`regex`, `unordered_map`) and will be so forever.

[5] We can optionally discuss adding language support for `extern "C++20"` or similar, to allow direct access to older ABI variants.

There are even small overheads in language constructs that cannot be resolved because of ABI. The aggregate costs of making such a breaking change only go up: Hyrum's Law tells us that with more time and more users, there are more dependencies upon the currently-available (if unpromised) ABI stability. On gcc and clang platforms, that stability goes back a decade now: changes at this point are a significant ecosystem expense.

If we promise ABI stability, I believe that industry involvement in (and dependence upon) the standard library will diminish. Concretely, I believe that Google's interest in the standard library will be limited to primitives that are demonstrably efficient, are expensive to copy, and that commonly pass through interface boundaries (roughly `std::vector` and maybe `std::string`, although we have ABI-incompatible optimization efforts underway for `string`). The standard will be useful as a starting point for small projects, for interoperability, and for organizations with only modest investment in compute resources. Any project that is more constrained by cost of compute than cost of human resources may need to evaluate whether their use of the standard library is efficient or if they are better served by the utility libraries published by large tech companies (Folly, Abseil, etc), or by the research and proving grounds projects like Boost.

This is not a doomsday scenario, but I believe it does represent a fundamental shift in values and strategy for the standard, especially for the standard library. For many years C++ has been known as the language to look to for performance. Explicitly leaving 5-10% aggregate performance on the table in preference to code compiled years ago will be a clear signal that we are not as committed to performance as we have suggested.

I call on WG21 to make a conscious and explicit choice here, with the clear awareness that status quo is an endorsement of indefinite ABI stability. If we wish to be the systems language known for performance, we have to act now. If not, we have to be aware that we are giving up on some important user bases.