Defining a Portable C++ ABI

Herb Sutter

Document #: N4028
Date: 2014-05-23
Reply to: Herb Sutter
(hsutter@microsoft.com)

Contents

Motivation

Problem

A C++ developer cannot compile C++ code and share the object file with other C++ developers on the same platform and know that the result will compile and link correctly. Our status quo is that two source files `a.cpp` and `b.cpp` can only be linked together if they are compiled with both:

- the same version of the same compiler, or another compiler with a compatibility mode; and
- compatible switch settings, since most C++ compilers offer incompatible switch settings where even compiling two files with the same version of the same compiler will not link successfully.

This is a longstanding source of problems, including but not limited to:

- *It creates common FAQs, such as:* "Why can't I link object files created using two compilers?" "Why can't I link object files created using different versions of the same compiler?" "Why can't I use `std::string` in my public function signature?" "Why can't I link two object files compiled with identical switch settings but different versions of the same compiler, just because both use (different versions of) the compiler vendor's own in-the-box `std::` library?"
- *It makes sharing binary C++ libraries more difficult:* To ship a C++ library in binary form for a given platform requires building it with possibly dozens of popular combinations of switch settings for the popular compiler(s) on that platform, and then may not cover all combinations. Alternatively, one can wrap the library in that platform's stable C ABI, which brings us to…
- *It is a valid reason to use C:* This is (the) one area where C is superior to C++. Among programs and programmers who would otherwise use C++, the top reason to use C appears to be the inability to publish an API with a stable binary ABI, including that it can be linked to from C, C++, and other languages' foreign function interfaces (FFIs) such as Java JNI and .NET PInvoke. In particular…
- *It therefore creates ongoing security problems:* The fact that C is the only de facto ABI-stable lingua franca continues to encourage type- and memory-unsafe C APIs that traffick in things like error-prone pointer/length pairs instead of more strongly typed and still highly efficient abstractions, including but not limited to `std::string` or the new `string_view`.
- *It is a perennial area for vendor-specific workarounds:* The development of technologies like COM and CORBA were largely motivated by the desire to use abstractions like classes and virtual functions in an ABI-stable API.

Finally, it is deeply ironic that C++ actually has always supported a way to publish an API with a stable binary ABI—by resorting to the C subset of C++ via `extern "C"`. We can and must do better.

Goals, Non-Goals and Constraints

The primary goal is to:

- Enable writing portable C++ code that will be compiled to an object file that can successfully:
    - link with object files created by other conforming C++ compilers, or different versions of the same compiler, on the same platform; and
    - be called using FFIs from other language compilers/runtimes on the same platform, such as Java JNI and .NET PInvoke, if those FFI implementations support the (single) C++ ABI for that platform.

Here a target *platform* has the same meaning as de facto in C, namely the combination of:

- *Operating system:* Such as Windows, OS X, Android, iOS, etc. The OS determines things like the binary format (such as PE, COFF, and ELF), the meaning of pointers (such as the legality of stealing low bits and the rules for high bits above bit 48), and more.
- *Processor family:* Such as x86 or ARM. This affects the instruction set targeted by code generation, alignment, endian-ness, and more.
- *Bitness:* Such as 32-bit or 64-bit. This affects the size and interpretation of pointers and references, `size_t`, and `ptrdiff_t`, and more.

This largely maps to "object files that could reasonably be expected to be part of a single executing program," since a given single running program will be running on a given OS and processor at a time.

Examples and corollaries:

- It should be possible to ship a single (not "fat") compiled binary library for Windows x86 32-bit whose interface uses C++ features such as classes, virtual functions, and overloading, and have that be usable with code compiled by other compilers for Windows x86 32-bit.
- It should be possible for an operating system API to expose a well abstracted modern C++ API that uses features like classes, virtual functions, and overloading, and also meets normal OS API requirements for ABI stability.

Notable non-goals and constraints:

- We need not, and should not, try to guarantee some "universal C++ ABI" whereby a single object file could link with other object files on different platforms, such as 32-bit ARM Linux and 64-bit x86 Windows. That would defeat the original purpose of C and C++ to be a portable language that can generate executable code that is competitive with handcrafted platform-specific code on a given platform.
- We must not change or restrict anything that is expressible today in platform-specific ways. Any new capabilities must be purely additive. All of the various incompatible compiler option flavors and standard library implementation flavors that are allowed today must still be preserved, both for compatibility and because most of them exist for good reasons (sometimes you do want packing, sometimes you do want a non-default calling convention, etc.).
- We should accommodate that some vendors, such as Microsoft, feel they must be able to break the standard library ABI on every major release (see next section).
- We should accommodate that some vendors, such as GCC, feel they must *not* break the standard library ABI (see next section).

Existing Practice

Examples of existing practice include:

- The Common Vendor ABI (Itanium C++ ABI) is a step in this direction, to specify an ABI for the language on some platforms. It is supported today by compilers such as GCC and EDG. It does not specify an ABI for the standard library, so this is necessary but insufficient to, for example, use `std::string` on a stable API boundary.
- Microsoft VC++ likewise has long had a de facto stable, though undocumented, ABI for the language. It does not have a stable ABI for the standard library, but rather intentionally breaks ABI

compatibility on every major release, for example in order to allow continuous improvements to the implementation and to quickly implement a new standard library that contains ABI-breaking changes.

- GCC, in addition to implementing the Itanium ABI, has long supported a stable ABI for the libstdc++ library, at the cost of flexibility to change and improve the library. For example, for some years GCC has attempted to ship a C++11-conforming `basic_string` that does not perform copy on write, but even the recently released GCC 4.9 still has not been able to ship a conforming `basic_string` mainly because of the binary compatibility issue.

Approach and Proposal

This section proposes an approach and strawman syntax to make a complete per-platform C++ ABI that includes two major parts: the language ABI, and the standard library ABI.

Language ABI: extern "abi"

A *language ABI* defines how object code will be generated for given C++ source code, so that object files that use C++ types and features, such as class types and virtual or overloaded functions, can be mixed and communicate on a stable ABI boundary. This includes a full specification of:

- object layout for both built-in types and user-defined types, including but not limited to alignment, padding, virtual base class subobjects, member pointers, virtual tables, and RTTI;
- function name mangling and calling convention;
- exception handling mechanisms; and
- linkage information, such as object file formats.

The programmer needs to distinguish between code compiled "the usual way"—whatever that is today, using the current rich variety of nonportable compiler options—and code compiled to the target platform's C++ language ABI.

As noted in the Overview, C++ has always supported a way to distinguish code compiled to the target platform's C ABI: `extern "C"`.

Therefore we propose the syntax: `extern "abi"`.

> **NOTE:** This is a strawman placeholder name only; see Q&A.

The syntax shall be usable wherever `extern "C"` is permitted today, including on individual declarations and `extern` blocks. For example:

```
extern "abi" {

    template<class T>                       // can use templates
    class gadget { ... };                   // can use class types

    class widget {
    public:
        virtual int add( gadget<int>& );    // can use virtual functions
```

```
        ...
    };

    bool overload( widget& );              // can use overloads
    bool overload( int, gadget<float>*& );

}
```

The meaning of `extern "abi"` shall be *implementation-defined*, which means it is required to be documented, but by the OS platform owner rather than by each C++ compiler implementation. (See *"Implementation-Defined": By the OS Platform* Owner on page 7.) As today, `extern` would be part of the type of a function, and therefore of the type of a pointer to function; see further notes about function pointer conversions in Q&A.

Standard Library ABI: std::abi::

A *standard library ABI* defines a stable implementation of standard library types, so that object files can be mixed and communicate using `std::` types, such as `string` and `vector<int>`, on a stable ABI boundary. This means providing the complete header implementation of every standard library type, containing all declarations and most implementations, except only for separately compiled function bodies which can be provided in a binary library.

The programmer needs to be able to distinguish between code that uses "the usual `std::` library" (whatever that is today, which usually is different for different C++ implementations on the same platform and could even be a user-provided standard library if they are not using the one that came with their compiler) and code that uses the target platform's C++ standard library ABI.

C++ already has a way to designate the C++ standard library types and functions: `namespace std`.

For symmetry with `extern "abi"`, we propose the namespace `std::abi`.

> **NOTE:** This is a strawman placeholder name only. See Q&A.

This gives us two distinct namespaces for two distinct things:

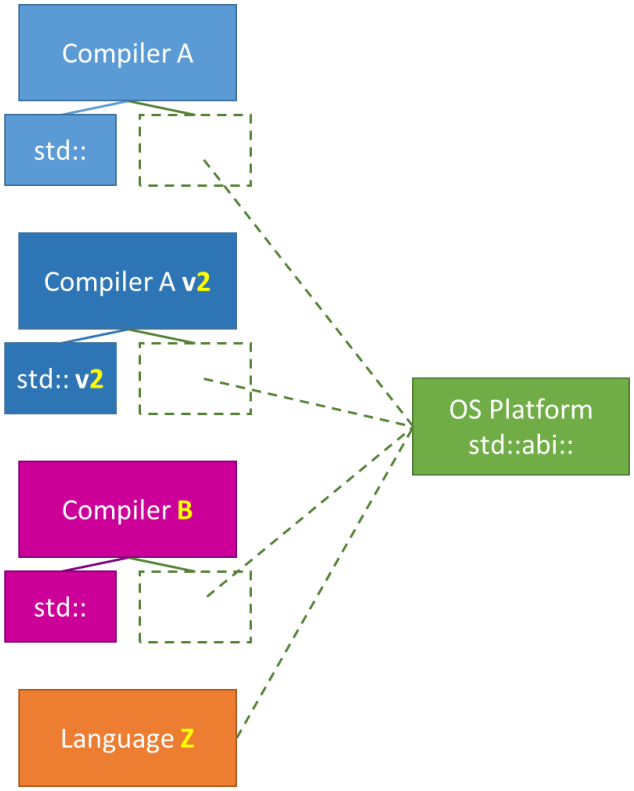
- `std` contains "the C++ standard library implementation that can change/evolve." This is provided *by each C++ implementer*, exactly as today.
- `std::abi` contains "the C++ standard library implementation that is binary stable." This is provided *by each OS platform*, however it wants, and is shared by all compilers that target that platform. A likely choice is to make this the release build of a snapshot of the OS platform owner's own C++ product's implementation of `std`, taken at the time they support the C++ ABI.

For convenience and safety, in an `extern "abi"` block `std::` means `std::abi::`.

In the standard, the namespace `std::abi` is specified to contain exactly the same types and functions as `std` plus a list of explicit differences; initially this list would be either empty, or list as omitted from `std::abi` those entities in `std` that are already deprecated.

This means that a standard type or function in both namespaces, such as `std::vector` and `std::abi::vector`, would have the same standard specification. However, it is up to the C++ `std` implementation whether any given `std` and `std::abi` types are the same or distinct. We have a precedent

for this situation in the current standard with `::iterator` and `::const_iterator` for the `set`-like containers. Adapting the existing rules stated in 23.2.4 [associative.reqmts]/6, we now require:



- For a given type or function XXX, it is unspecified whether `std::XXX` and `std::abi::XXX` are the same type or function. A `std` implementer might prefer for some cases to make them be the same, such as to guarantee no conversions happen for an efficient library boundary, and for other cases to make them be distinct, such as to provide their own implementation in `std` or to replace a previous shared implementation with an improved implementation in `std`.
- For a given type XXX, `std::XXX` and `std::abi::XXX` are explicitly convertible to each other as long as XXX is copyable. If they are the same type, the conversions are no-ops. If they are not the same type, the `explicit` conversions are provided by `std::XXX`.

Corollaries:

- Portable user code cannot overload on `std::XXX` type and `std::abi::XXX`, because they might be the same type. As a QoI matter, as for the `set` iterator types, compilers could elect to provide warnings if they notice code (such as with overloads, specializations, etc.) that would behave differently if the types collapsed.
- Implementers of `std` can elect to overload on `std::XXX` type and `std::abi::XXX` if they know their types are different.
- A C++ compiler could choose to not contain its own separate implementation of `std` at all, but simply use `std::abi` through an inline namespace.

This helps both the committee maintain the standard, and C++ library implementers maintain their implementations.

First, as the committee maintains the standard:

- When the committee adds a new library feature, it adds the feature as usual to `std`, where it is automatically picked up also in `std::abi`.
- When the committee makes a library binary breaking change, it applies the change to `std` only, and lists it as a difference between `std` and `std::abi` (possibly in Annex C?).

So initially the difference in specification is zero, but as we take binary breaking changes there will be a small cumulative difference consisting of those changes only. As a side effect, because `std::abi` is specified as "what's in `std` plus a list of explicit differences," we automatically document exactly the set of binary breaking changes in a single place.

Second, as C++ implementers maintain their standard library implementations:

- When a C++ implementation implements a new standard library feature: The OS platform's own C++ compiler product adds the feature to both `std` and `std::abi`, typically with a single implementation using `inline namespace`. Other C++ compiler products add the feature to their own `std` only.
- When a C++ implementation implements a binary breaking change, whether to conform to a new standard or just to change or improve their own implementation: Each C++ compiler product makes the change in their own `std` only.

Distinguishing `std` and `std::abi` directly addresses a current tension:

- Today, GCC lags in implementing `std` conformance because it tries to make `std` itself stable and so can't take breaking changes to track conformance. If we clearly separate the "stable" `std::abi` and "latest" `std` standard libraries, GCC can stop trying to make `std` itself stable, and still ship a stable `std::abi` as today while additionally shipping a more current and conforming `std`.
- Today, Microsoft is an example in the opposite direction: VC++ breaks binary compatibility deliberately on every major release and so gets the flexibility to track the standard library's breaking changes and improve its own implementation, but it doesn't have a way to use `std` types in ABI-stable APIs. If we clearly separate the "stable" `std::abi` and "latest" `std` standard libraries, VC++ could additionally ship a stable `std::abi` while still shipping its current ever-updated `std` as today.

This approach allows portable C++ code to explicit use ABI-stable types, and to use standard library types in ABI-stable APIs. For example:

```
int count_commas( std::string );       // we can say which
int count_periods( std::abi::string ); // one we want to use

extern "abi" {  // inside this block, std:: => std::abi::

    int count_words( std::string );    // takes a std::abi::string

}
```

"Implementation-Defined": By the OS Platform Owner

Because the ABI is per "platform," it is the OS platform owner, not every C++ implementation, who is responsible for defining what the C++ ABI means on its platform. For example, for the platform "Windows x86 32-bit," Microsoft's Windows team (possibly delegating to their native compiler team) would be ultimately responsible for specifying the Windows C++ ABI.

All C++ compilers that target the OS platform would then support the OS platform's C++ ABI in addition to their normal custom compiler modes and options.

Also, all non-C++ languages that want to support calling C++ libraries and APIs would likewise support the OS platform's C++ ABI as a FFI target.

Migrating Existing Code to Expose a Stable ABI: /extern:abi or similar

It is expected that compilers also add a compiler option, such as a switch, to compile an entire translation unit as though wrapped in an `extern "abi"` block. We could optionally mandate that an implementation provide such a mode.

This would enable arbitrary existing C++ code, such as any existing C++ library *in toto*, to be exposed with a stable ABI for a given platform just by recompiling it (once per target platform).

Beyond the Basic ABI: Helping Developers Design ABI-Safe Libraries

Additionally, developers often would like more guidance on how to design a library such that it will have a backward (and sometimes forward) compatible ABI with other versions of the same library. That seems to be at least in part because the standard doesn't define what kinds of API evolution maintain a stable binary interface—the standard says all evolution causes undefined behavior through the ODR, and then some implementations make it "work in practice" without really documenting their guarantees—and it's genuinely hard to evolve a C++ API in an ABI-compatible way even given those implementation characteristics. For example, you can't add a virtual function to a base interface if it has a derived interface that adds a virtual function. You can't add an optional parameter to a C++ function since it changes the mangling. You can't add a field at the end of a small struct passed or returned by value because that will change the calling convention.

My view is that two things are needed here in the standard:

- The standard should state more explicitly what things will or won't break the ODR. For example: When can you add a virtual function, or a template parameter? When can you change a default argument? When can you add an explicit `override`? Lawrence Crowl and others have proposed this in the past.
- The standard should consider adding the ability to explicitly make writing ABI-safe C++ types and functions easier for important common cases. For example, a perennial developer request is how to opt into hiding a class's data member implementation, not only as a compilation firewall to improve compilation times but also to enable the ability to change the object implementation without affecting link compatibility. C++ developers have long written a Pimpl or similar idiom by hand, with ample opportunity for pitfalls and repetitive boilerplate code. If there is interest, it would be useful to consider a proposal for a language extension to directly supporting "Pimpl'ing" a type, for example allowing a class definition in the header to omit the class's private data members (and possibly private functions, which would affect overload resolution and has direct overlap with the modules effort) and have those provided separately in the implementation file.

Q&A

Q: Are there other `extern`/namespace names? A: Yes, lots of room for bikeshedding.

The goal from this paper is to get discussion going on the general issue and one possible approach for a solution. But yes, if we decide we want this bike shed, there are many colors we could paint it, such as:

- `extern "cpp"` and `std::cpp::` (at least it's more search-friendly…);
- `extern "cxx"` and `std::cxx::` (following the Itanium ABI approach…);
- `extern "OS"` and `std::OS::`;
- `extern "platform"` and `std::platform::`;
- or something else.

Q: Could you provide definitions for common terms? A: Sure.

API means the source-level declarations advertised by a library, such as the set of function declarations in the library's headers.

ABI means the binary link-level implementation details of using the library, such as object layout, vtable layout, function calling conventions including exception handling mechanisms, etc.

Stable API means that existing calling code can still correctly compile against a newer version of the library. There are no source breaking changes. It is possible to write a stable API in portable Standard C++.

Stable ABI means that existing linkable calling object files can still correctly link with a newer version of the library. There are no binary breaking changes. This proposal intends to make it possible to write a stable ABI in portable Standard C++.

Q: Does this interact with modules? A: It's related but distinct: source compilation vs. binary linking.

The ISO C++ modules proposals are about improving the way source code is compiled, specifically to replace/improve the `#include`-based build model.

This proposal is about enabling the way binary object files are linked, specifically to specify a stable link target.

There is some overlap, specifically in the places where 'source' and 'binary'/separate-compilation cross, namely templates. This should be coordinated.

Q: Is this related to COM? CORBA? A: No, it's orthogonal.

No, this is just "C++ as it is today compiled in a predictable way." It does not create a new type system, but just defines a stable target (basically a specific set of compilation switches plus a specific `std` implementation) for the native C++ libraries as they already exist, without doing anything more.

COM and CORBA are different and orthogonal. Those tools are not just for ABI stability, but also for implementation hiding and building pluggable services. They offer far more functionality (e.g., implementation hiding) and much less (e.g., lack of support for templates and inlining) than this proposal. They also incur overhead in the ABI boundary because they're essentially a different type system and provide additional abstractions and services.

This proposal does not attempt to do implementation hiding, service discovery, or any of the wonderful things that make COM/CORBA what they are. Developers can still use Pimpl or COM/CORBA, or C++ idioms like Pimpl, as usual, independently and in combination with this proposal.

Q: What about efficiency? A: This is normal C++ compilation and therefore normal C++ efficiency.

This is just "C++ as it is today compiled in a predictable way" including full optimization and inlining. It's about identifying a stable targetable binary format for normal C++ compilation, while still doing everything C++ does including exposing private implementation to the compiler of the calling code and the library code, so that the compiler can generate code to access private members directly instead of via (possibly virtual) functions.

Q: What about crossing the boundary? A: Low (possibly zero) and predictable cost.

Crossing any boundary potentially incurs cost, and using `abi::` types can incur conversion overheads on the boundary. This proposal allows implementations to choose on a case-by-case basis for any type or function XXX whether to optimize for making the boundary cheap (by using the same type for `std::XXX` and `std::abi::XXX` to eliminate conversions) or to optimize for "the internals" of a library (by using different and presumably better/improved/custom implementations for `std::XXX` and converting to/from `std::abi::XXX` on the boundary).

In an environment where `std::` and `std::abi` are the same, or the user code just uses std::abi everywhere, there is no overhead for conversions on the boundary but they might lose some optimizations or techniques that could be used in a distinct `std::` type.

Q: What about servicing (e.g., inline functions)? A: Same as today: recompile.

Like today, it should rarely be necessary to service the ABI-stable `std::abi` library in a way that would affect binary compatibility. But it does happen, for example for security fixes to inline functions, where some of the code is duplicated and compiled into the caller. Because all we're doing is specifying a fixed target for the same model as today, the answer in this model is exactly the same as today, and no worse than today: You need to ship an updated library and have calling code recompile if they want to take advantage of the fix.

Q: What if I think a language ABI isn't a problem? A: There's still the stdlib.

Some platforms have less language ABI diversity than others; developers on those platforms are likely to be less sensitive to the need for explicitly supporting a language ABI. It's pretty widely agreed that there is more of an explosion of configurations at the standard library level, and that is an interesting level that can be attacked independently of whether we decide to do something in the language for the language ABI.

Q: What does "stable" mean – how long is "forever"? A: Same as today: per OS generation.

An OS typically takes an ABI- breaking change every decade or so anyway, such as from OS 9 to OS X, Windows 95 to Windows NT, and Windows 32-bit to Windows 64-bit. In this proposal each is simply considered a distinct OS "platform" target, which is basically just documenting the status quo.

Q: Would this approach mandate the use of inline namespaces for `std`? A: Probably yes.

In particular, enabling a mode where `std::XXX` means `std::abi::XXX` is important to easily enable existing source code to be compiled without change to the ABI, and this probably requires selecting between a `std::normal` vs. `std::abi` as an inline namespace.

Consider the case where a developer statically links one object file partially using plain `std` (from compiler v1) and `std::abi`, with another object file partially using plain `std::` (from compiler v2) and `std::abi`, where the `std::abi` parts connect (e.g., one object file calls a function with a `std::abi` interface in the other object file), and the plain `std::` parts are implementation details. Inline namespaces are necessary to avoid ODR violations. They are also necessary to prohibit mixing, for example that object file A cannot call functions in object file B if their interfaces are plain `std::` and the versions mismatch; that is, inline namespaces work as a fine-grained mismatch detection mechanism.

Q: Can the Itanium ABI be a starting point for the language ABI? A: Yes, at least section headings.

For the language, the standard would specify a list of things shall be implementation-defined per platform. That standardized list should likely be a superset of the things captured in the "section headings" of the Itanium ABI.

Q: Would platform X use the Itanium ABI? A: Probably, if it does already.

That's up to each platform vendor. On platforms that already support the Itanium ABI today, presumably the C++ ABI they would choose to document would likely include a superset of the "section contents" of the Itanium ABI.

Q: Does this mean each OS platform vendor would publish their C++ language ABI? A: Yes.

If the standard specifies an implementation-defined language ABI, then to conform each OS vendor would need to define the implementation of the ABI for their platform. That would likely be (one of the) current ABI(s) that vendor already has now, or possibly a new one if they use this opportunity to take a breaking change to fix issues/cruft in their current ABI(s).

Q: Could `std::abi` contain just a few fundamental types, instead of everything in `std`? A: Yes, but it would be limiting, it's hard to draw a line, and it's easy to take it all.

Yes, `std::abi` could contain a subset of `std`. This was raised by several reviewers. One same response, paraphrased:

> 'Do we need `map` and `deque` in stable ABIs? Do we really need `std::num_get<char>`?! Or could we make do with `vector` and `string`? …
>
> 'Some of those essential types would be: `vector<T, allocator<T>>` (which also implies the standard allocator is one of the essentials); `basic_string<T, char_traits<T>, allocator<T>>` (implying `char_traits` too); `unique_ptr<T, default_delete<T>>` (and maybe `shared_ptr<T>`); `function<T>`; `pair<T, U>` and `tuple<T...>`; `mutex`; probably `atomic<T>`; the exceptions in `<stdexcept>`.'

I think there are three reasons not to define a subset.

First, it's limiting. Even the question above uses the term "make do with."

Second, it's hard to draw the line because there would be more useful things just beyond any given boundary. For example, using the above suggestions:

- If `mutex`, why not `unique_lock` which can be passed around and so makes sense to put in library interfaces?

- If `unique_ptr<T, default_delete<T>>`, why not `unique_ptr<T, X>` where X is `extern "abi"`? Note that `unique_ptr` does seem to be getting use with custom deleters.
- If `vector<T, allocator<T>>`, why not a custom allocator?
- And even more so, why not `map<K,V>` or `unordered_map<K,V>` given that a lookup dictionary is a very common data structure?

Third, it's easy to take the entire contents of `std`. This proposal includes a mode whereby arbitrary existing code can be compiled as-is to the ABI, and every major compiler already ships with a standard library implementation that could be compiled in that mode. (Further, every major compiler vendor also already has the ability to for the user to select among different versions/builds of its standard library, and to use entirely a different standard library implementation than the one it ships with.)

Q: Who is the 'OS platform owner' who defines the C++ ABI for an OS like GNU/Linux that is not controlled by one vendor? A: Likely either or both of GCC/libstdc++ and Clang/libc++.

I see two natural options, either or both of which work.

Likely either the GNU/Linux C library team or the GCC team would probably choose GCC's g++/libstdc++ implementation as the platform C++ ABI for GNU/Linux.

Alternatively, some Linux distribution might decide to base themselves on LLVM's Clang/libc++ implementation. That too would be fine, it would just be a different target "platform" from other GNU/Linux distros and wouldn't be link-interoperable via `std::abi`.

Even if both options are chosen, that's fine and again seems to be just documenting the status quo we have anyway today of OS fragmentation choices. The situation is the same as today, and probably better.

Q: Does `extern "abi"` affect the type of a function and affect overloading? A: Probably yes.

It technically does for `extern "C"` today, but many compilers do not implement this.

If a function is declared `extern "abi"`, it would be useful to assign its address to an `extern "C++"` function pointer, or to an `extern "C"` function pointer. We should be able to support it as follows:

- If the compiler produces code with the same calling conventions to the target platform's language ABI, it just works – the compiler would product identical code generation as today.
- Otherwise, if the compiler produces code with different calling conventions to the target platform's language ABI, either by default or because of compiler options being used for a given piece of code, the compiler can generate an inline conversion wrapper and take the address of that. Note that this means portable code could *convert* pointers to function as above, but it cannot *compare* `extern "abi"` and non-`extern "abi"` pointers to functions.

Q: Would this proposal provide even more portability/stability than C? A: Yes.

C compiler options allow you to compile C code that doesn't use the platform's usual ABI, for example causing structures in system headers to be packed or using alternative calling conventions. Anything declared `extern "abi"` is an instruction to the compiler to ignore such compiler options and meet the platform ABI.

Acknowledgments

Thanks to Chandler Carruth, James Dennett, Gabriel Dos Reis, Artur Laksberg, Stephan T. Lavavej, Jason Merrill, Leonard Mosescu, Clark Nelson, Richard Smith, Bjarne Stroustrup, Jonathan Wakely, and Jeffrey Yasskin for their comments and feedback on this topic and/or on drafts of this paper.

References

The following is directly or indirectly related previous work.

Binary compatibility:

- EWG reflector thread, “Binary compatibility for C++1y” (October 2012)
- See c++std-ext-13626, 28, 35, 37, 39, 43, 46, 48, 77, 85, etc.

Modules:

- N3347: D. Vandevoorde, “Modules in C++ (Revision 6)” (January 2012)
- N4047: G. Dos Reis, “A Module System for C++” (forthcoming, May 2014)

Dynamic and shared libraries:

- N1400: M. Austern, Toward standardization of dynamic libraries (September 2002)
- N1418: P. Becker, Dynamic Libraries in C++, Notes from the Technical Session in Santa Cruz, Oct. 21, 2002 (November 2002)
- N1496: P. Becker, Draft Proposal for Dynamic Library Support in C++, Revision 1 (March 2003)
- N1976: B. Kosnik, Dynamic Shared Objects: Survey and Issues (April 2006)
- N2407: L. Crowl, C++ Dynamic Library Support (September 2007)