C++ Club UK Meeting 139

Gleb Dolgich

2021-11-11

Contents

Visual Studio 2022 released 2
C++ Developer Roadmap 2
Using semantic versioning for the C++ standard 3
Break ABI to save C++ 3
What is an ABI, and Why is Breaking it Bad 7
ABI verification 10
    ABI Compliance Checker 10
    Libabigail 11
Breaking C++ ABI in the standard is neither necessary nor sufficient 11
“Should we break the ABI” is the wrong question 12
Ignore `nodiscard` values 13
Gosh Darn Member Function Pointers! 13
Reddit 14

Visual Studio 2022 released

Microsoft just announced Visual Studio 2022 on their blog and posted the keynote video. See also: Reddit

Notable features for us C++ developers are:

- Hot reload (yes, for C++!)
- Remote testing on Linux
- The IDE is finally 64-bit
- msvc160 is binary-compatible back to msvc140! (oh boy do I have more on that shortly)

The VS2022 roadmap says:

> Visual Studio 2022 will add support for C++20 language features that simplify management of large code bases. We are integrating support for CMake, Linux, and WSL to make it easier to create and debug cross-platform apps.

C++ Developer Roadmap

A translation from the original in Russian contains links to useful articles on C++ development and a link to the roadmap itself, which is a mindmap. It looks very impressive. The mindmap includes both technical and ‘soft’ (or ‘people’) skills, because we are people and it’s never just tech. The technical skill set is divided into levels which are in turn divided into steps. This should come handy to any aspiring C++ developer. Redditors are impressed.

Using semantic versioning for the C++ standard

A redditor asks if semantic versioning would work for the C++ standard. In other words, would it be possible to allow ABI breaks in C++ 2.0.0 versus C++ 1.x.y? I think this reminds me of the C++ Epochs proposal which didn't have enough support in the Committee.

Some solutions proposed in the thread require a time machine (why don't we have a smaller standard library) or completely ignore the reality of large scale C++ development (recompiling is trivial).

Some say they would freeze at the currently used standard if ABI were broken again:

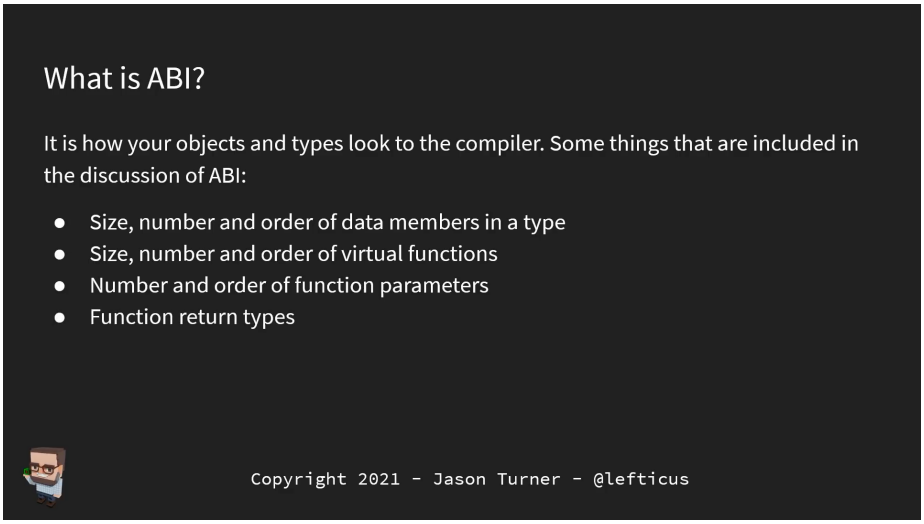
> I can say for sure that where I work all 40 teams would freeze on C++17 if they broke ABI again. We are still dealing with the ABI issues introduced by C++11.

Someone posted a link to an interesting list of differences explaining how C++20 breaks source compatibility.

Break ABI to save C++

Jason Turner posted a video called Break ABI to save C++ in which he does a high-level overview of "why ABI stability is probably making your code worse right now".

First Jason reminds us what is ABI: it's how the program looks to the compiler, including the following:



He then clarifies that function return type is not part of the mangled function signature, so if it changes, the function consumer will get the

wrong value back. More Godbolt illustrations follow, on what happens if you change virtual function order without recompiling the user code, or change number of member variables or their order.

Jason mentions Visual Studio and its ABI freeze. It made my life much easier when migrating several large libraries with lots of dependencies to newer versions of Visual Studio, but I can appreciate frustration expressed by many, including STL himself.

To look at things actually lost to stable ABI, Jason refers to the blog post by Corentin Jabot dated 24/02/2020 and rather dramatically called The Day The Standard Library Died. In the post Corentin lists what has been broken and cannot be easily fixed because of the decision by the committee to not break ABI. Some of the items in the list are:

- slow associative containers
- slow `std::regex` (Corentin notes that it's currently faster to launch PHP to perform a regex match that to do it with `std::regex`)
- `std::regex` lack of UTF-8 support
- non-conformant implementations for the sake of stability
- rejected changes to `error_code`
- `unique_ptr` not fitting into a register (to make it truly zero-overhead)
- `shared_ptr` improvements

ABI discussions in Prague showed that WG21 is not in favour of an ABI break in C++23 but is open to it in a future version of C++. There is no promise of ABI stability forever, and the committee prioritizes performance over stability.

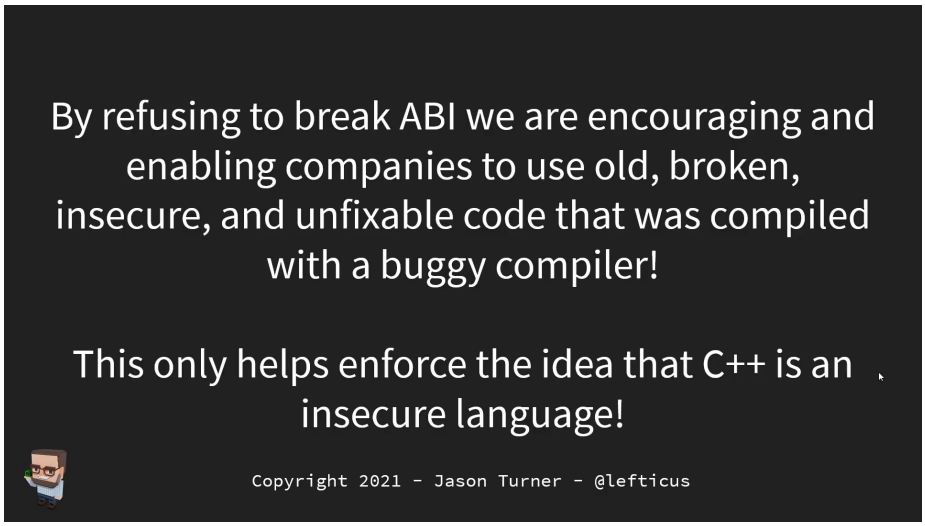
Corentin states:

> I believe, quite strongly, that not breaking ABI in 23 is the worst mistake the committee ever made.

Jason makes an interesting claim: he heard rumours of committee members quitting because of the ABI stance.

As we saw a few meetings back, Microsoft have said that because of ABI stability promise in Visual Studio they are not going to support some C++20 features, like the `[[no_unique_address]]` attribute.

Jason explains why we have ABI stability: because companies demand it as they want to keep using libraries that cannot be recompiled. He then tries very hard to convince the viewer that companies lika that are wrong and should get their act together.



Yes, theoretically I agree, old libraries are bad and insecure, and weren't built using modern compilers, so ideally we should get rid of them, or upgrade them to use modern C++, or at least recompile them. But in practice upgrading a legacy codebase to get rid of an old library can be a daunting proposition and a very expensive exercise costing millions.

I suppose one solution could be keeping the legacy codebase at the same C++ standard as the critical dependency that cannot be rebuilt. But this is a pessimistic approach — ABI stability allows us to improve the entire codebase except *this one dependency*, which is still better than the alternative.

Jason says:

> To save C++ we must break ABI in the standard library

But what if it kills C++ instead? Hmmm, choices...

His major point here is to discourage use of libraries of unknown quality. (*I have a surprise for you: that's most libraries.*)

On the reasons of why we haven't broken ABI, Jason states:

> We don't want to splinter the C++ community because of large companies refusing to update compilers and deal with ABI breakage.
>
> Newsflash: Large companies already cling to old compilers because large companies move slowly!

This argument puzzles me. So what, we can totally disregard large companies that haven't yet upgraded their compilers and stop them from moving forward at all, and that is somehow OK? (BTW, not *all* large companies move slowly. Also the way Jason generalizes companies like that based on his teaching experience sounds a bit too simplistic to me.)

Jason's solutions to the problem of old binary library dependency are (1) reimplement it (may be difficult or impossible like in case of old hardware), or (2) create a C interface for the library (hey, why don't you add another level of indirection!)

His informal polls indicated that *everyone* is OK with breaking ABI.

Sigh. It mush be nice to live in a theoretical world. Or in Google, where everything is recompiled all the time.

Curiously, Jason also mentioned a similar discussion on the lack of ABI stability in Rust:

- Rust doesn't indeed have a stable ABI, because developers want to have the freedom of changing implementations of the built-in types.
- Rust toolchain links the Rust standard library *statically* into every binary built with Rust.
- Developers can expose a stable C API from their Rust libraries.

The video gathered a lot of comments on YouTube. I have a personal rule not to read YouTube comments, but in this case many of them were quite insightful and even had some additional information. Here is Jason's comment on his video:

> What does it mean to the world if `std::vector` is 1% faster? How much electricity, computer time, and money would be saved?

(*Well done for invoking the environment card. To be fair though, C++ is already one of the top most environment-friendly languages, even when used for mining cryptocurrency!*)

There is a reality check from Robert Douglas:

> In the medical field subject to FDA oversight, moving a library forward to a new compiler ran the risk of requiring that library to go through clinical validation, again, potentially costing millions. For a small company who couldn't readily afford such, there was a lot of pressure to make sure compiler changes to critical pieces of code were suuuuuuuuper rare. Without ABI compatibility, the cost of using newer tools reached 7 digits.

Most commenters seem to think that if someone wants to use an old library and keep ABI stable, they automatically don't update their compilers. I think this conclusion is not always correct. Stable ABI allows gradual upgrades and improvements within large legacy codebases, which is a good thing IMHO.

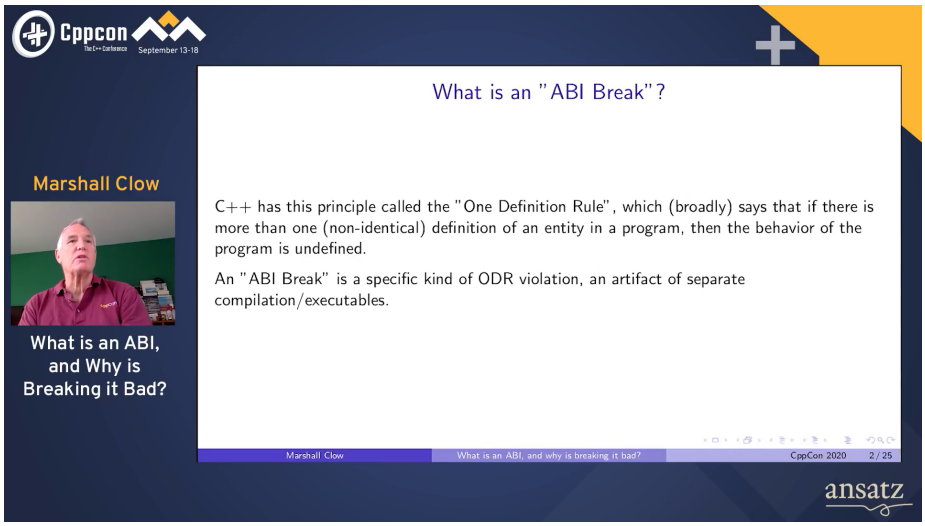
Read comments for more bad takes. I don't have the strength.

What is an ABI, and Why is Breaking it Bad

One of the comments in the Reddit thread above mentioned a talk by Marshall Clow at CppCon 2020 called What is an ABI, and Why is Breaking it Bad?.

My alternative title for this video would be "How things work in the real world."

Marshall Clow explains what is an ABI break:

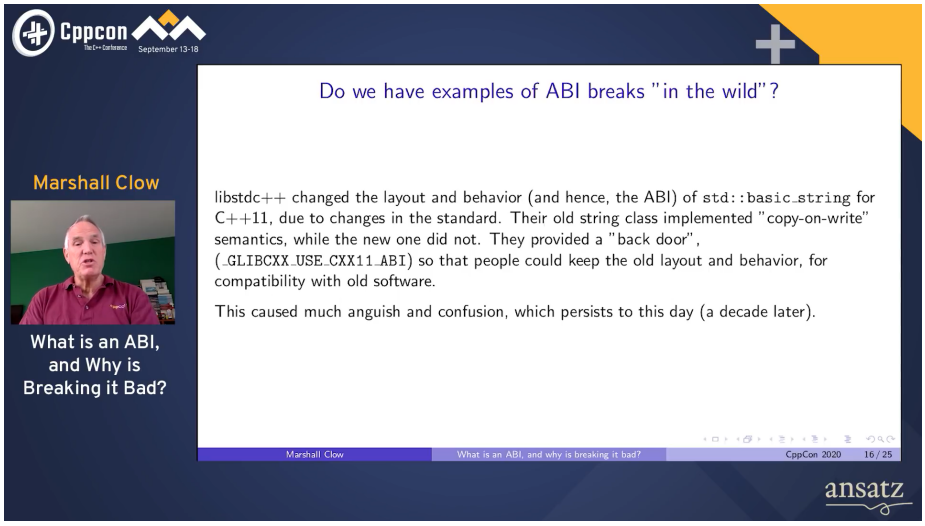


He says that ABI is defined by the platform and not by the C++ Standard.

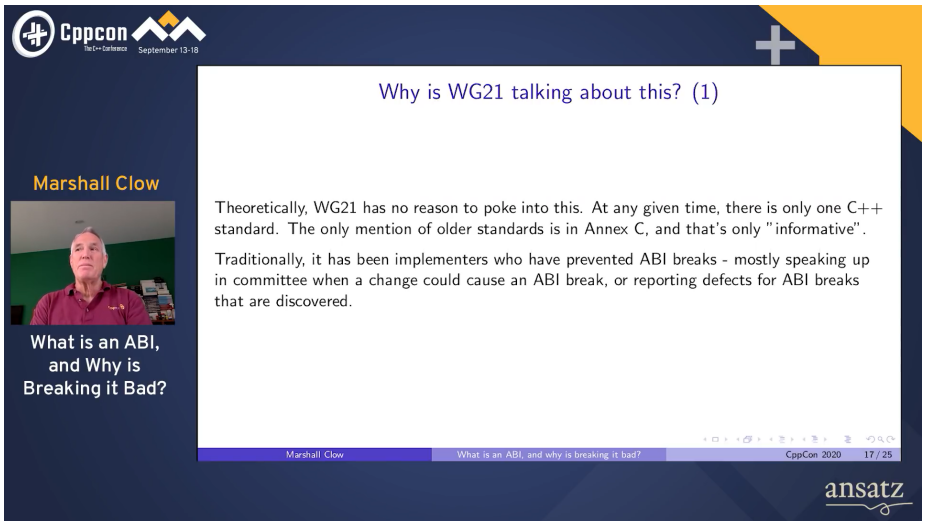
In addition to the list of potential ABI breaks that Jason shows above, Marshall adds:

- different definitions of a class (an ODR violation), for example, in two different headers (or due to different preprocessor macros), or in two different versions of the same library
- different `#pragma pack` for different class definitions between versions

As a cautionary tale, Marshall reminds us of the infamous `std::string` ABI break in C++11 which was still causing grief 10 years after the change:



What does WG21 have to do with ABI?



Some members, says Marshall, want to allow ABI breaks in the Standard, and started a meta-discussion about it. One proposed way of allowing an ABI break is to request implementors to change the name mangling scheme. This would mean that code compiled with, say, `-std=c++26` wouldn't link with object files compiled with a previous standard. Solution: "fat" binaries (Apple have been using them for ages).

Vendors would have to ship different binaries for different C++ standards from then on, which would quickly defy the purpose of shared libraries and explode the testing burden.

How do you detect such an incompatibility? It is only "materialized" at load time, when "the system" is assembled for a particular program. Any binary that uses C++ internally could be affected. Example: you update Adobe Photoshop and none of your plugins work, because they use an old ABI. (*Hello, Firefox.*)

Marshall says in the Q&A session after the video that a safe solution to fixing the existing problem is to introduce new better versions of library facilities and deprecate the old ones, just like Java does. Case in point: CTRE potentially replacing `std::regex`.

Someone asked about C++ Epochs, to which Marshall replied that he is not a fan because that would fracture the user base, leaving a bunch of people behind, and making it harder to advance towards the new stuff at their own pace. (*See: the above C++11 std::string debacle.*) Others said that epochs would require a centralised repository of epoch information, which is impossible to achieve because C++ has multiple compiler vendors. Also, in C++ template instantiation happens at 3 times, and every time the

epoch must match, which would be difficult to ensure.

The YouTube comments don't give much food for optimism. The first comment goes:

> It's sad to see the amount of effort some people will put into rationalizing why progress can't happen.

Can we just link everything statically please, like Rust?

And then the two camps collided and Marshall Clow got invited to a CppCast to talk to Jason Turner and Rob Irving. The episode is titled ABI Stability with Marshall Clow, it was released in May 2021, and in it Marshall reiterates the points from his earlier talk, perhaps a little bit more strongly. He says:

> Some people *do* want to break all C++ programs.

and

> I've heard people say "Users who won't rebuild their software every 3 years are holding C++ back".

Marshall says that it's unrealistic to require users to rebuild their software from source, and to even ask them to keep using their old software after there has been an ABI break, because their OS vendor might release a new update which breaks the old software (as an example, Apple doesn't version their `libc++`). To him, breaking the ABI and deciding to allow user programs to just crash is a non-starter.

Marshall gives an example of a company doing it right: Apple keeps shipping the old compatible version of `libc++` for legacy systems but switches to the new, more efficient version when they create a new platform and define a new ABI. They did it when they introduced 64-bit iOS and they did it again for M1-based Macs.

Marshall says he'd like to see a way forward to evolve the standard library, and currently it seems to be following the Java example and deprecating/replacing old inefficient classes with new and better ones.

ABI verification

This makes sense for library and platform vendors, but not for end users. The tools are useful, but are not enough to have a complete guarantee.

ABI Compliance Checker

Niall Douglas mentioned this tool. It's a free and open source utility that checks for ABI breaks between two libraries in binary or source code form.

You can incorporate it into your CI and be warned if your changes broke ABI.

The tool detects a range of binary and source compatibility problems:

- changes in memory layout due to added/removed/moved class members or changes of their type
- changes in vtable layout due to added/removed/moved virtual functions or added pure virtual functions
- changes in enumerations
- removed symbols
- changes in function parameters, their names, or default values
- changes in attributes (`const`, `volatile`, `static`)
- changes in macro definition values
- changes in base classes
- changes in class member access levels

For more information read the article Policies/Binary Compatibility Issues With C++ by KDE developers.

Libabigail

Libabigail is a library for building custom ABI checking tools based on comparing DWARF information. There is an article at Red Hat blog describing the process.

Breaking C++ ABI in the standard is neither necessary nor sufficient

Redditor **kalmoc** posted this:

> TL;DR:
>
> It is not necessary, because standard library providers don't need the standard's permission to break ABI in order to fixup/improve the performance of e.g. `regex` or `unordered_map`.
>
> It is not sufficient, because the list of things that only affect ABI, but still need the standard's permission is so small that it won't help C++ in any significant way. Similar for the other way round for the list of things the standard can't add just because it would require the tool vendors to break ABI which they refuse.

The discussion leaned towards "the committee is stagnant, we need to break ABI" and "the committee first needs to accept there is an ABI". As Marshall Clow said earlier, ABI is in the purview of platforms and not the C++ standard.

“Should we break the ABI” is the wrong question

Jussi Pakkanen, the creator of Meson build system, posted this on his blog in May 2021. He wrote:

> The ongoing battle on breaking C++’s ABI seems to be gathering steam again. In a nutshell there are two sets of people in this debate. The first set wants to break ABI to gain performance and get rid of bugs, whereas the second set of people want to preserve the ABI to keep old programs working. Both sides have dug their heels in the ground and refuse to budge.
>
> However debating whether the ABI should be broken or not is not really the issue. A more productive question would be “if we do the break, how do we keep both the old and new systems working at the same time during a transition period”.

As a solution, Jussi is advocating a multi-architecture support currently used in Debian to support 32- and 64-bit binaries at the same time.

The problem with this approach is that while it works for a limited set of incompatible binaries, it would be impractical to provide separate libraries for every potential ABI-breaking version of the standard.

To solve that, Jussi proposes to use the ABI versioning fields in the ELF header, completely ignoring operating systems that don’t use the ELF format, like Windows or macOS.

Jussi mentions Rust, its lack of a stable ABI. He says that attempts to force Rust to commit to a stable ABI have so far failed. One of the commenters replied that Rust is a bad example since it has a declared goal to not support shared libraries at all.

In the Reddit thread, a Microsoft STL developer commented:

> The problem is that implementers would have to maintain both versions for a very, very long time. It’s not just creating parallel installs (easy) but making sure even people on the old ABI get new features and can update their compiler and standard library.
>
> If you just make a parallel installation and then completely abandon it that’s... not really improving the situation.

This led to more replies like “why do we care about *those people* not being able to upgrade their compilers” and “you should be able to always recompile the world” and so on, which I’m just going to ignore because I can.

But enough of that.

Ignore `nodiscard` values

Jonathan O'Connor tweeted:

> C++ Trick of the Week: To ignore nodiscard return values:
>
> `std::ignore = someNodiscardFunction();`
>
> `std::ignore` has an assignment operator that accepts any type. It's quick and easy to write, and easy to search for, when you want to clean up the code properly.

Credit for this tip goes to Johann Studanski.

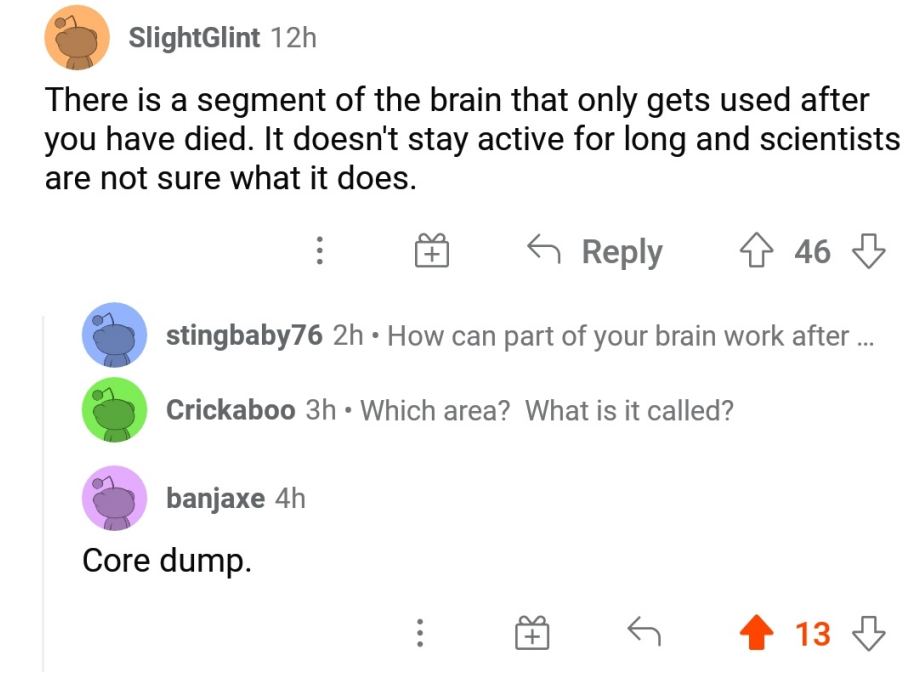
Gosh Darn Member Function Pointers!

If you don't know the member function pointer syntax by heart, this website may prove useful (scroll to the bottom for a more memorable URL!) There are examples of the following:

- how to invoke a member function pointer
- how do i declare a member function pointer
- the verbose syntax for the above

I'm pretty sure even expert C++ programmers google it from time to time, so this website and its memorable URLs will come handy!

Reddit



Someone on Reddit also suggested it's the brain destructor.