Could We Handle an ABI Break Today?

LUIS CARO CAMPOS

Could We Handle an ABI Break Today?

Luis Caro Campos
JFrog

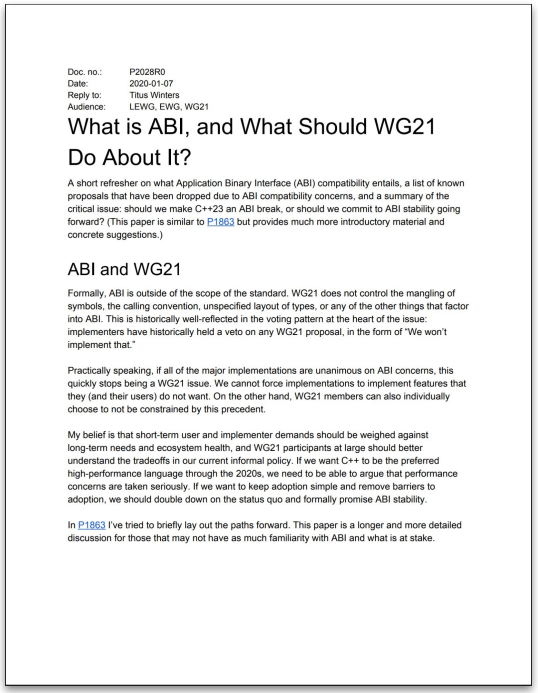
What this talk is NOT about

- Whether or not we should have an ABI break
- Benefits or disadvantages of breaking ABI
- Arguments in favor or against

What this talk is about

- Intro and context
- Initially: The practical consequences of a potential ABI break, ie “we have an ABI break, now what”
- But actually:
  - How we’ve dealt with C++ ABI breaks in the past
  - How we STILL deal with link incompatibilities very frequently
  - How tooling has advanced in the past 10 years to make the next transition easier

What is “ABI”?



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.

Application **B**inary **I**nterface

“How entities built in one translation unit interact with entities from another”

- Mangled names
- Size in bytes / alignment
- Semantics / binary representation
- Calling conventions, parameter passing in functions

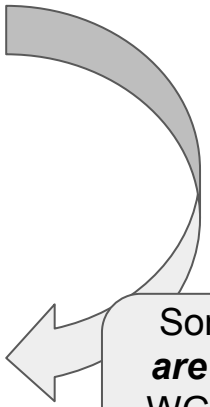
“What is ABI, and What Should WG21 Do About It?” by Titus Winters, 2020

ABI and the C++ Standard

- **ABI is outside of scope** - platform and vendor specific
- In scope for the C++ standard:
  - The C++ language specification
  - The C++ standard library
    - **Templates**
    - **Classes**
    - **Functions**
    - **Constants**
    - **Macros**
    - **…**

ABI and the C++ Standard - cont'd

- **ABI is outside of scope** - platform and vendor specific
- In scope for the C++ standard:
  - The C++ language specification
  - The C++ standard library
    - **Templates, classes, functions, constants, macros …**
- But **the following can break ABI**:
  - Changes to class/struct definitions
    - Adding, removing, or changing order of virtual functions
    - Adding or removing member variables
    - Adding or removing a base class
  - Changes to function signatures
  - Changes to namespaces (adding to, removing from, inlining…)
  - …



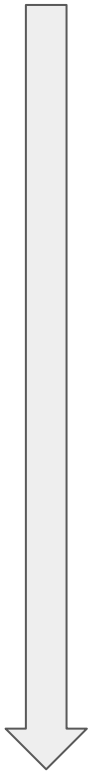
Some changes that ***are*** in the purview of WG21 could result in compiler vendors having to break ABI

Breaking the ABI

- If ABI is platform and vendor specific …

  … so is the way of handling and signaling incompatibilities

ABI break from a developers perspective



Obvious linker error - when mixing objects with different ABIs E.g. “You’re mixing objects with different ABIs!!!”

Non-evident linker error

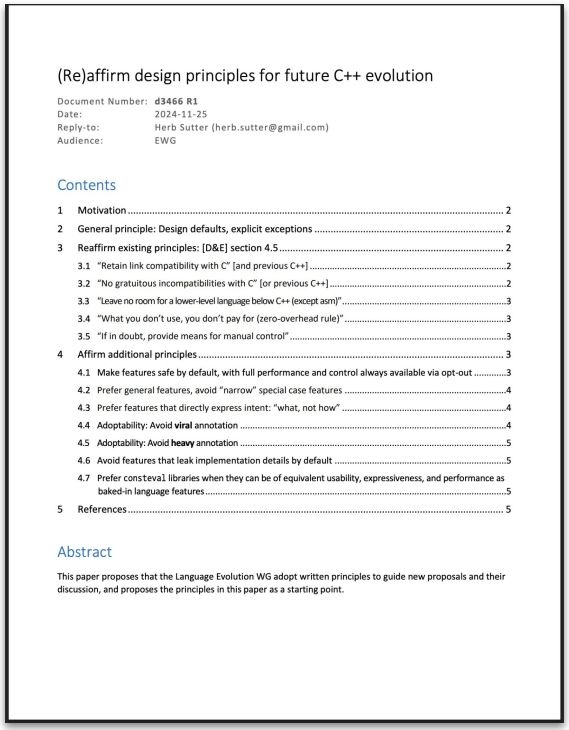
```
undefined reference to `std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::basic_string()'
```

Runtime error - e.g. segfault. Incompatibility was not detected early, program may crash

No error - program does not crash, but does not behave correctly

Less preferable

(Re)affirm design principles for future C++ evolution



(Re)affirm design principles for future C++ evolution

Document Number: **d3466 R1**
Date: 2024-11-25
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: EWG

Contents

Abstract

This paper proposes that the Language Evolution WG adopt written principles to guide new proposals and their discussion, and proposes the principles in this paper as a starting point.

"(Re)affirm design principles for future C++ evolution" by Herb Sutter

Language Evolution Working Group (EWG)

November 2024

(Re)affirming design principles for future C++ evolution

3.1 "Retain link compatibility with C" [and previous C++]

This is under "Use traditional (dumb) linkers" but the main point that still applies.

100% seamless friction-free link compatibility with older C++ should be a default requirement. We can decide to take an ABI breaking change on a case by case basis (or, potentially, even wholesale) but should do that with explicit discussion and document the reasons why.

Example: We should not require wrappers/thunks/adapters to use a previous standard's standard library.

Example: We should not make a change we know requires an ABI break without explicitly approving it as an exception. For example, in C++11 we knew we made an API changes to `basic_string` that would require an ABI break on some implementations.

3.2 "No gratuitous incompatibilities with C" [or previous C++]

100% seamless friction-free source compatibility with older C++ must always be *available*.

Example: We can adopt "editions" or an alternate syntax, as long as there is a mode where existing syntax is available. For example, we added `using` aliases which subsumed all uses of `typedef`, and we could in future embrace a syntax mode where `using` was valid and `typedef` was rejected in that mode, but only if a mode that allowed all existing code including `typedef` was still available. (See also 2.5.)

Example: We should not bifurcate the standard library, such as to have two competing `vector` types or two `span` types (e.g., the existing type, and a different type for safe code) which would create difficulties composing code that uses the two types especially in function signatures.

- Link compatibility with previous C++
- Source compatibility with older C++
- No competing types in the standard library

Retain link compatibility with previous C++

What does this mean?

Retain link compatibility with previous C++?

I should be able to compile my code with C++26 and link *binaries* that were built with C++23

I should be able to compile my code with C++26 with today's compiler, and link *binaries* built with C++17 with a previous version of the compiler

I should be able to compile my code with C++26 and link a *binaries* built from C++ sources in 2003

I should be able to build my code with C++17 and link a *binaries* that was built with C++26 but has a C++17 compatible API

**binary: object file, static library or shared library*

Past C++ ABI breaks

MSVC ABI breaks

Binary files (.obj, .lib, .dll) created with one major version of the Visual C++ compiler were generally **not linkable with or usable by code compiled with a different major version**.

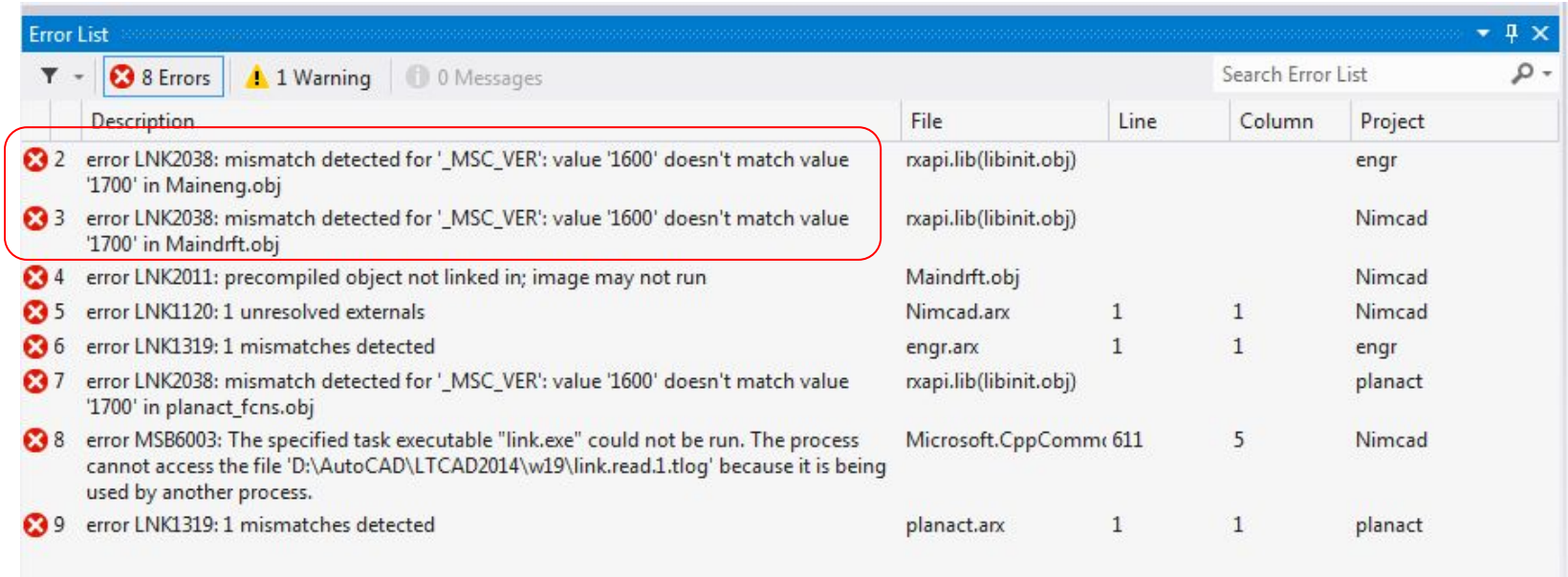
This changed with Visual Studio 2015 - which retains binary compatibility in the v14x versions of the toolset.

MSVC ABI Breaks (cont'd)

Product Name	Version Number	Release Year
Visual Studio 6.0	6.0	1998
Visual Studio .NET (2002)	7.0	2002
Visual Studio .NET 2003	7.1	2003
Visual Studio 2005	8.0	2005
Visual Studio 2008	9.0	2007
Visual Studio 2010	10.0	2010
Visual Studio 2012	11.0	2012
Visual Studio 2013	12.0	2013
Visual Studio 2015	14.0	2015

Breaking releases roughly every 2-4 years

MSVC ABI Breaks (cont'd)



	Description	File	Line	Column	Project
2	error LNK2038: mismatch detected for '_MSC_VER': value '1600' doesn't match value '1700' in Maineng.obj	rxapi.lib(libinit.obj)			engr
3	error LNK2038: mismatch detected for '_MSC_VER': value '1600' doesn't match value '1700' in Maindrft.obj	rxapi.lib(libinit.obj)			Nimcad
4	error LNK2011: precompiled object not linked in; image may not run	Maindrft.obj			Nimcad
5	error LNK1120: 1 unresolved externals	Nimcad.arx	1	1	Nimcad
6	error LNK1319: 1 mismatches detected	engr.arx	1	1	engr
7	error LNK2038: mismatch detected for '_MSC_VER': value '1600' doesn't match value '1700' in planact_fcns.obj	rxapi.lib(libinit.obj)			planact
8	error MSB6003: The specified task executable "link.exe" could not be run. The process cannot access the file 'D:\AutoCAD\LTCAD2014\w19\link.read.1.tlog' because it is being used by another process.	Microsoft.CppComm(	611	5	Nimcad
9	error LNK1319: 1 mismatches detected	planact.arx	1	1	planact

Trying to link objects built with Visual Studio 2010 and Visual Studio 2012

From https://stackoverflow.com/q/43184433

MSVC ABI Breaks (cont'd)

product name	VC (Version Code)	marketing year	_MSC_VER	_MSC_FULL_VER	runtime library version
Visual Studio 2010 [10.0]	10.0	2010	1600	160030319	10
Visual Studio 2010 SP1 [10.0]	10.0	2010	1600	160040219	10
Visual Studio 2012 [11.0]	11.0	2012	1700	170050727	11
Visual Studio 2012 Update 1 [11.0]	11.0	2012	1700	170051106	11
Visual Studio 2012 Update 2 [11.0]	11.0	2012	1700	170060315	11
Visual Studio 2012 Update 3 [11.0]	11.0	2012	1700	170060610	11
Visual Studio 2012 Update 4 [11.0]	11.0	2012	1700	170061030	11
Visual Studio 2012 November CTP [11.0]	11.0	2012	1700	170051025	11
Visual Studio 2013 Preview [12.0]	12.0	2013	1800	180020617	12
Visual Studio 2013 RC [12.0]	12.0	2013	1800	180020827	12

From https://en.wikipedia.org/wiki/Microsoft_Visual_C++

MSVC ABI Breaks (cont'd)

Set the OpenCV environment variable and add it to the systems path

First, we set an environment variable to make our work easier. This will hold the build directory of our OpenCV library that we use in our projects. Start up a command window and enter:

```
setx -m OPENCV_DIR D:\OpenCV\Build\x86\vc11     (suggested for Visual Studio 2012 - 32 bit Windows)
setx -m OPENCV_DIR D:\OpenCV\Build\x64\vc11     (suggested for Visual Studio 2012 - 64 bit Windows)

setx -m OPENCV_DIR D:\OpenCV\Build\x86\vc12     (suggested for Visual Studio 2013 - 32 bit Windows)
setx -m OPENCV_DIR D:\OpenCV\Build\x64\vc12     (suggested for Visual Studio 2013 - 64 bit Windows)

setx -m OPENCV_DIR D:\OpenCV\Build\x64\vc14     (suggested for Visual Studio 2015 - 64 bit Windows)
```

Here the directory is where you have your OpenCV binaries (*extracted* or *built*). You can have different platform (e.g. x64 instead of x86) or compiler type, so substitute appropriate value. Inside this, you should have two folders called *lib* and *bin*. The -m should be added if you wish to make the settings computer wise, instead of user wise.

From
https://docs.opencv.org/3.4/d3/d52/tutorial_windows_install.html

C++11 ABI changes in `libstdc++`

- GCC5 adds **new implementations** for `std::string` and `std::list`
  - ABI incompatible with the previous
  - `std::string` was copy-on-write in old ABI, and small string optimization with new ABI
- Rather than a wholesale ABI break, implementers decided that libstdc++ would support both ABIs
  - Using nested namespaces

✅ Source compatible
No changes required to user code

✅ Backwards binary compatibility
A program using the new ABI can link libraries built with the earlier ABI - libstdc++ has symbols for both

⚠️ Linking objects that pass `std::string` or `std::list` will not link with code that uses old ABI

libstdc++11 linking errors

I'm compiling C++ programs with gcc `version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.11)`. Everything is fine at compile time.

I then link those programs with a library that was built with `gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3)`.

I then get undefined reference link errors:

```
foo.cpp.o:(.rodata._ZTV46RigidBodyGRVelocityProjectionCollisionResolver[_ZTV46RigidBodyGRVelocityProjectionCollisionResolver]+0x18): undefined reference to
`RigidBodyGRVelocityProjectionCollisionResolver::getName[abi:cxx11]() const'
```

From https://stackoverflow.com/q/55406770

libstdc++11 linking errors

undefined references to std::__cxx11

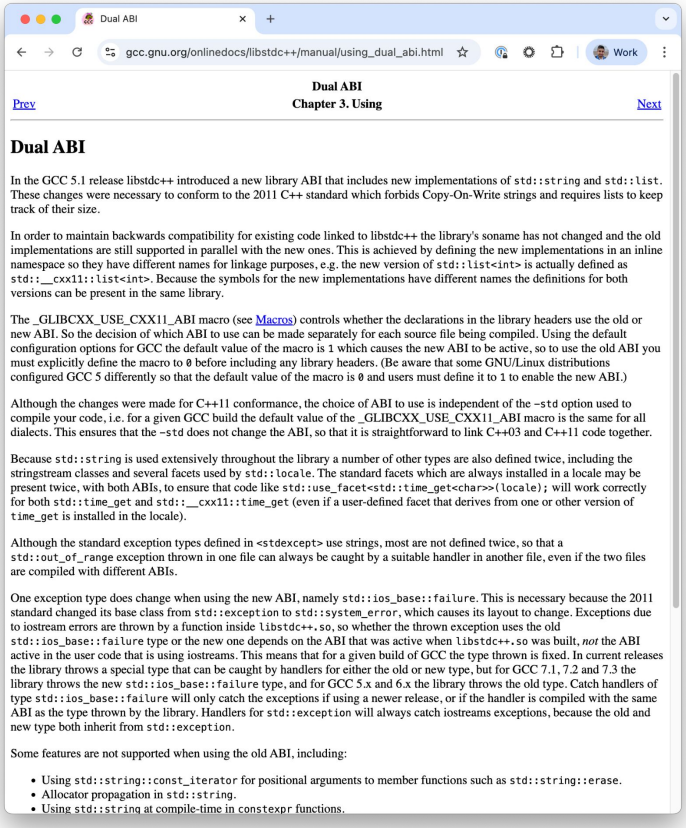
It is not a bug, if packages fail with undefined references to std::__cxx11 or [abi:cxx11] like

```
cmGlobalGenerator.cxx:(.text+0x12781): undefined reference to `Json::Value::Value(std::__cxx11::basic_string<char, std::char_traits<char>, st
d::allocator<char> > const&)'
```

It means you need to **rebuild the package** that provided that symbol first.

From
https://wiki.gentoo.org/wiki/Upgrading_from_gcc-4.x_to_gcc-5.x

libstdc++ new ABI: the fine print



"Although the changes were made for C++11 conformance, the choice of ABI to use is independent of the `-std` option used to compile your code, i.e. ***for a given GCC build*** the default value of the _GLIBCXX_USE_CXX11_ABI macro is the same for all dialects. This ensures that the `-std` does not change the ABI, so that it is straightforward to link C++03 and C++11 code together."

GCC docs at
https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html

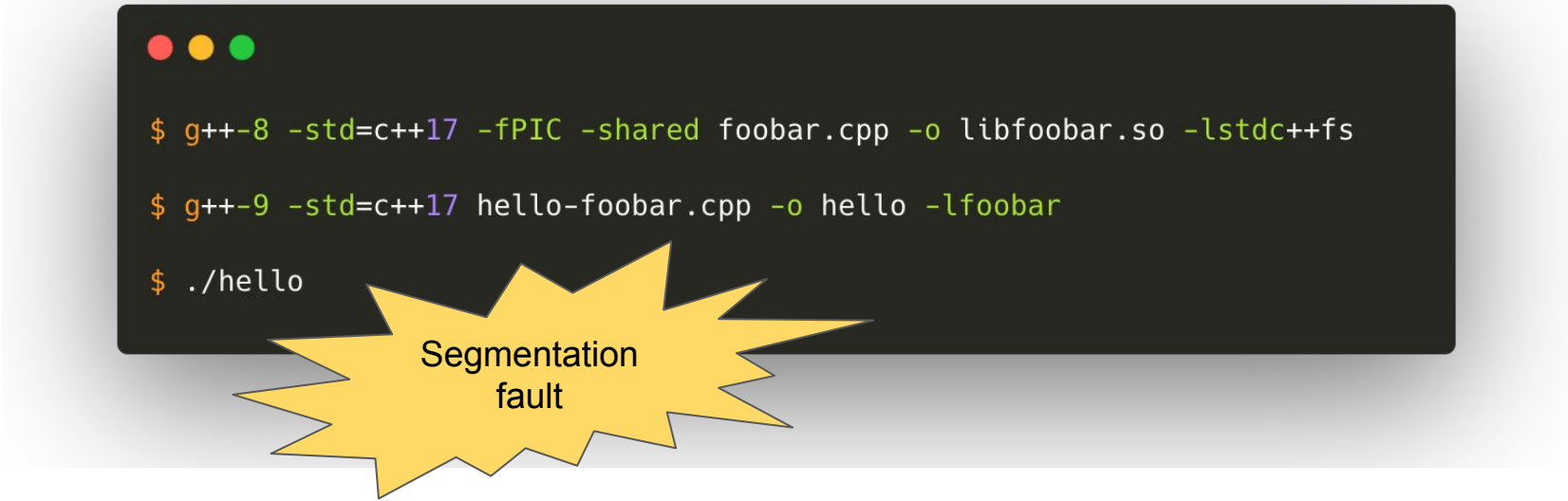
⚠️ Two different builds of the same gcc version can be ABI incompatible…

stdlibc++ and C++17’s std::filesystem

GCC version	Compiler flag	Header	Namespace	Link
gcc-7	`-std=c++1z`	`#include <experimental/filesystem>`	`std::experimental::filesystem`	`-lstdc++fs`
gcc-8	`-std=c++17`	`#include <filesystem>`	`std::filesystem`	`-lstdc++fs`
gcc-9	`-std=c++17`	`#include <filesystem>`	`std::filesystem`	`(-lstdc++, implicit)`

std::filesystem

std::filesystem



gcc and C++ standards: the finer print

C++17 Support in GCC

GCC has almost full support for the 2017 revision of the C++ standard. The status of C++17 library features is described in the library documentation.

C++17 mode is the default since GCC 11; it can be explicitly selected with the `-std=c++17` command-line flag, or `-std=gnu++17` to enable GNU extensions as well. Some C++17 features are available since GCC 5, but support was experimental and the ABI of C++17 features was not stable until GCC 9.

Raise your hand:

Are you using C++20 and gcc?

gcc and c++20

> ## C++20 Support in GCC
>
> GCC has experimental support for the previous revision of the C++ standard, which was published in 2020. The status of C++20 library features is described in the library documentation.
>
> C++20 features are available since GCC 8. To enable C++20 support, add the command-line parameter `-std=c++20` (use `-std=c++2a` in GCC 9 and earlier) to your `g++` command lin
> Or, to enable GNU extensions in addition to C++20 features, add `-std=gnu++20`.
>
> **Important**: Because the ISO C++20 standard is recent, GCC's support is **experimental**.

As of September 2025, with gcc 15 being the most recent release, **C++20 support is still experimental**

Less recent ABI breaks

- `libstdc++.so.6` was released with gcc 3.4 in 2004
  - ABI incompatible with previous
  - Code had to be rebuilt with new version against new runtime
  - `libstdc++.so.5` could coexist in the system, but both could not be loaded in the same process
- macOS (then OS X) transitioned from libstdc++ (gcc) to libc++ (llvm based)
  - Libraries linked with libstdc++ where incompatible with libraries linked with libc++

Other link incompatibilities

Position independent executables

- The gcc builds in some Linux distros started to default to generating programs built as **Position Independent Executables**
- Takes advantage of Address Space Layout Randomization (ASLR), which protects against return-to-text and memory corruption attacks
- A problem for pre-existing static libraries linked into executables

glibc 2.31 removal of math finite functions

Link errors with -ffast-math (-ffinite-math-only) and glibc 2.31

Asked 5 years ago Modified 5 years ago Viewed 2k times

Recently, glibc (namely with glibc 2.31, included in Ubuntu 20.04) seems to have removed families of functions like `__exp_finite()`.

These functions were used when compiling with gcc's option `-ffinite-math-only` (or `-ffast-math`, which enables the said option).

My problem is that I have *compiled closed sources* static libraries provided by third parties which have been presumably compiled with this flag and those libraries generate linking errors to missing math functions like `__exp_finite()`.

glibc 2.31 removal of math finite (cont'd)

> So you have some .o or static libraries that are compiled with the old
> headers from previous versions of glibc and then trying to link with
> the latest version of glibc?
> That is itself is not a supported use case IIRC for backwards
> compability of glibc.
>
> Thanks,
> Andrew Pinski
>
> ---
>
> - Previous message (by thread): Math changes in glibc 2.31
> - Next message (by thread): Math changes in glibc 2.31
> - **Messages sorted by:** [date] [thread] [subject] [author]
>
> ---

From https://sourceware.org/pipermail/libc-alpha/2020-May/113816.html

✅ shared libraries linked against glibc will work with newer glibc

❌ static libraries not guaranteed to be linkable with newer glibc?

Past ABI breaks - summary

- MSVC versions had “wholesale” ABI breaks roughly every 2-4 years
  - But have been ABI stable since Visual Studio 2015
- GCC’s libstdc++ last “broke” ABI with the 2015 gcc5 release
  - But gcc C++ ABI for new standards remains unstable for a while (c++20 is still experimental)
- Binary objects (.o, .a, .so) built with gcc for Linux are not always *link compatible*:
  - Across distros (each may have made different libstdc++ ABI decisions)
  - Across distros due to glibc versions
  - Across different versions of g++, especially while support is “experimental” (years!)
  - When using static libraries (glibc ABI, binutils versions, etc)

glibc runtime

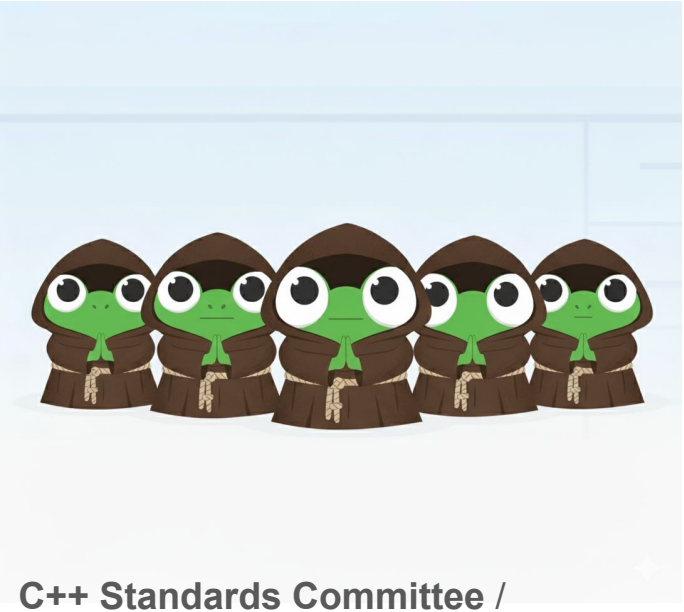
- glibc 2.36 removed the `DT_HASH` symbol hashtable - in favor of `DT_GNU_HASH`
- Not strictly part of the ABI - but still a runtime break
- This broke certain video games anti-cheat components

Win32 Is The Only Stable ABI on Linux

2022-08-15, by ivyl

Current C++ ABI issues

Retain link compatibility with previous C++



C++ Standards Committee / compiler vendors: we must retain link compatibility with previous C++ to avoid disruptive ABI breaks

Library authors: I understand your concern but I do not have the same concern

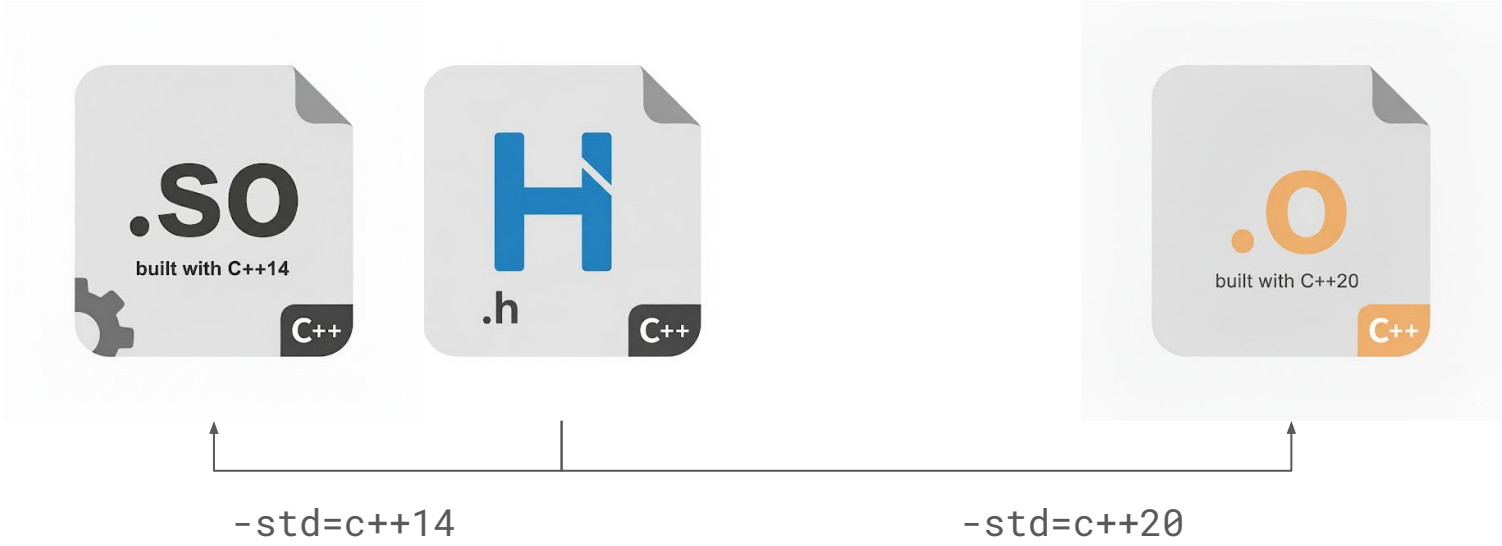
Conditional APIs… lead to incompatible ABIs

```
#if __cplusplus >= 201703L
  // API for C++17 or newer
#else
  // API for older standards
#endif
```

```
#if __has_include(<span>)
...
#elif __has_include(<gsl/span>)
...
#endif
```

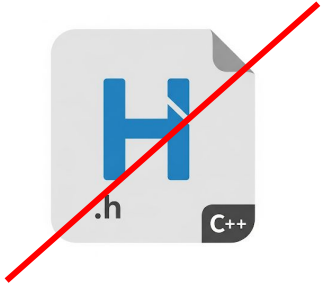
```
#    if defined(__cpp_lib_string_view)
using StringView = std::string_view;
#    else
using StringView = std::string;
#    endif
```

One library, different compilation modes



✅ source compatible with the public header
❌ link error: missing / undefined references

Dealing with polyfills



Header with compile-time
macro conditionals

**Header with hard-coded
options written at library build time,
or alternatively,
Transitive C++ flags that ensure a
given value of consumers**

The header will use the same “substitutes” for all C++ modes

✅ Link compatible

Example: abseil

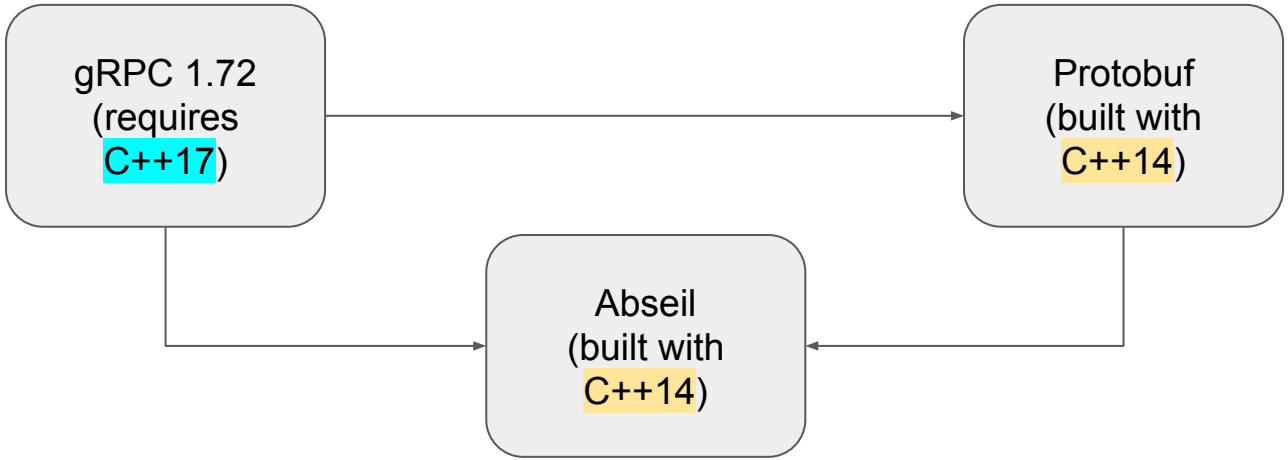
```
// -----------------------------------------------------------------------------
// File: options.h
// -----------------------------------------------------------------------------
//
// This file contains Abseil configuration options for setting specific
// implementations instead of letting Abseil determine which implementation to
// use at compile-time. Setting these options may be useful for package or build
// managers who wish to guarantee ABI stability within binary builds (which are
// otherwise difficult to enforce).

// package is compiled with C++14

#define ABSL_OPTION_USE_STD_ANY 0
#define ABSL_OPTION_USE_STD_VARIANT 0
#define ABSL_OPTION_USE_STD_STRING_VIEW 0
#define ABSL_OPTION_USE_STD_OPTIONAL 0
```

Example: abseil (cont'd)

```
#ifdef ABSL_USES_STD_VARIANT

#include <variant>

// use the c++17 std::variant
namespace absl {
    using std::variant;
    using std::monostate;
    using std::visit;
    using std::holds_alternative;
    using std::get;
    (...)
}

#else

namespace absl {
    template <typename... Ts>
    class variant;
    // ...
    // Implementation of custom variant class
    // ...
}

#endif
```

Will use this implementation because the macro in `options.h` is hardcoded to choose this

gRPC, protobuf and abseil



“C++ should be link-compatible with older C++”

gRPC, protobuf and abseil (cont'd)

```
src/core/call/request_buffer.cc:27:34: error: no matching function for call to
'std::variant<grpc_core::RequestBuffer::Buffering, grpc_core::RequestBuffer::Buffered,
grpc_core::RequestBuffer::Streaming,
grpc_core::RequestBuffer::Cancelled>::variant(absl::lts_20250127::in_place_type_t<grpc_core::RequestBuffer::Bufferi
ng>)'
   27 | RequestBuffer::RequestBuffer() : state_(absl::in_place_type_t<Buffering>()) {}
      |                                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/util/useful.h:22,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/util/time.h:29,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/lib/resource_quota/periodic_update.h:24,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/lib/resource_quota/memory_quota.h:42,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/lib/resource_quota/arena.h:39,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/call/call_arena_allocator.h:24,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/call/call_spine.h:21,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/call/request_buffer.h:20,
                 from /root/.conan2/p/b/grpc0dfeefae40d5b/b/src/src/core/call/request_buffer.cc:15:
/usr/include/c++/13/variant:1482:9: note: candidate: 'template<long unsigned int _Np, class _Up, class ... _Args,
class _Tp, class> constexpr std::variant<_Types>::variant(std::in_place_index_t<_Np>, std::initializer_list<U>,
_Args&& ...) [with long unsigned int _Np = _Np; _Args = _Up; _Tp = {_Args ...}; <template-parameter-2-5> = _Tp;
_Types = {grpc_core::RequestBuffer::Buffering, grpc_core::RequestBuffer::Buffered,
grpc_core::RequestBuffer::Streaming, grpc_core::RequestBuffer::Cancelled}]'
 1482 |         variant(in_place_index_t<_Np>, initializer_list<_Up> __il,
```

gRPC, protobuf and abseil (cont'd)

Commit `bd13a27`

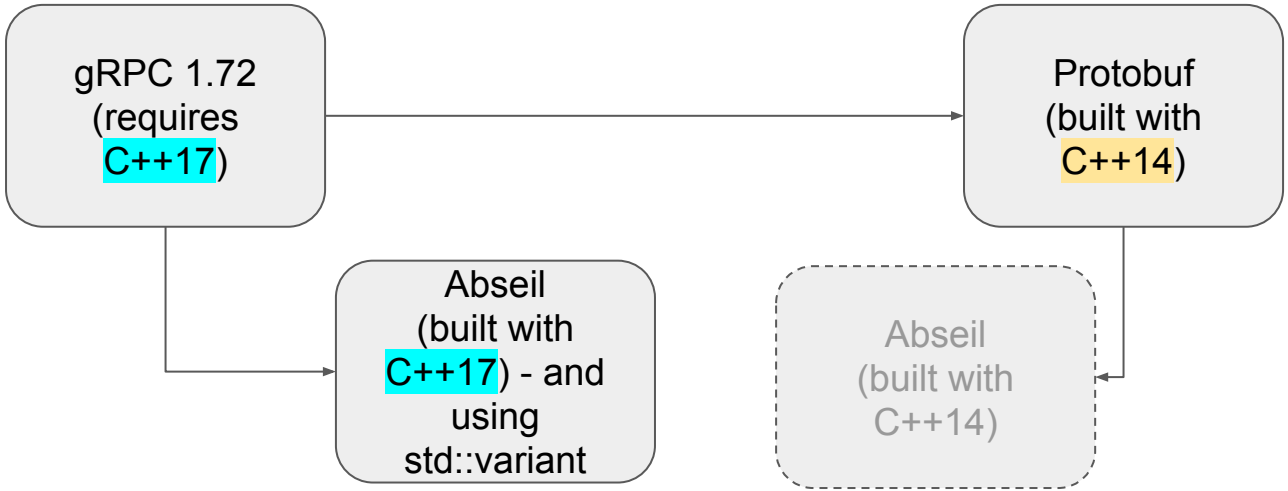
drfloob authored and **copybara-github** committed on Jan 9 · ✕ 2 / 3

```
[C++17] Replace absl::variant with std::variant (#38419)

absl::variant is a C++11 compatible shim for C++17's std::variant.
cpp/blob/2f016c4575a1e65e1b231d8db75e67ced21dfc14/absl/types/varia

Closes #38419

COPYBARA_INTEGRATE_REVIEW=#38419 from drfloob:cpp17-variant 914ffd4
PiperOrigin-RevId: 713817791
```

master · v1.75.0-pre1 ••• v1.71.0-pre2

✅ absl::variant would have worked with the hardcoded type in abseil
❌ std::variant may conflict with absl::variant implementation

gRPC, protobuf and abseil



“C++ should be link-compatible with older C++”

gRPC, protobuf and abseil

Abseil and gRPC

Abseil Option Configurations

Abseil provides a file (`options.h`) for static configuration of certain implementation details. Such static configurations are not often necessary but may be useful for providers of binaries/libraries which wish to ensure that their built-upon copies of Abseil are exactly the same.

Motivation

Abseil promises no ABI stability, even from one day to the next. Two different source revisions of Abseil cannot b they must be b the One Definit common probl from source.

However, while always possibl options mecha implementatio

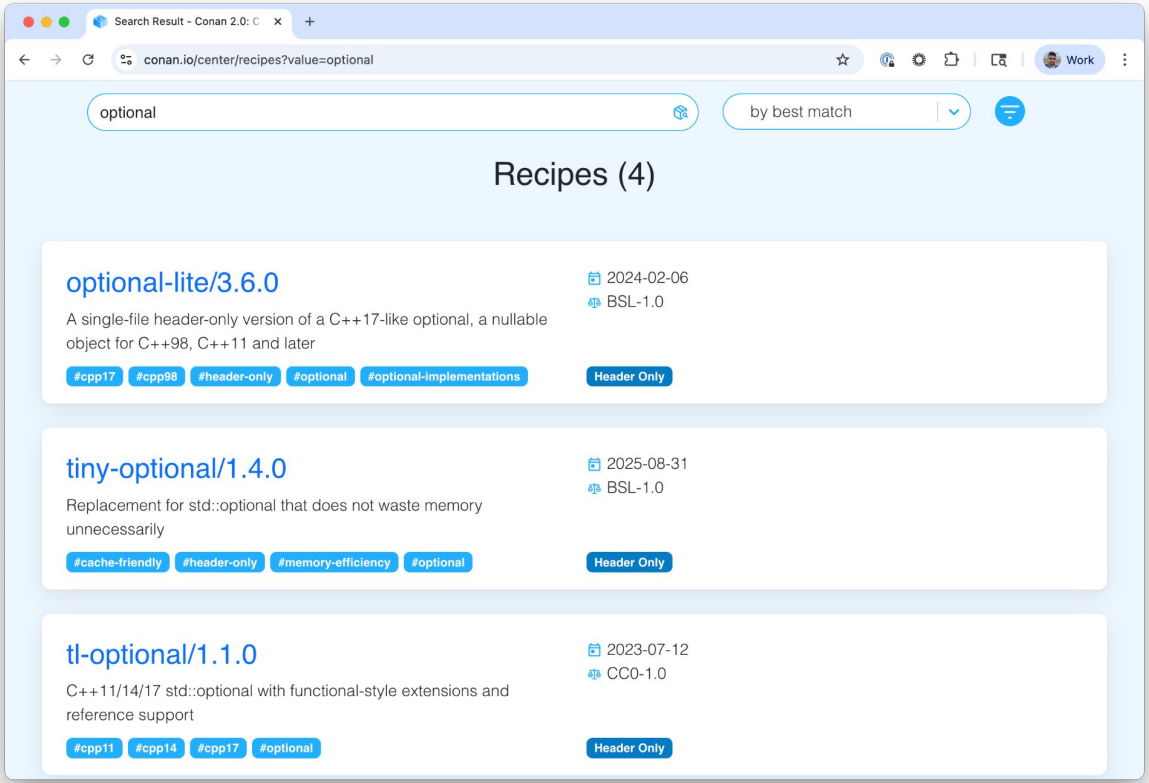
grpc / **BUILDING.md** ↑ Top

Preview | Code | Blame | 291 lines (219 loc) · 11.7 KB | Raw

Consistent standard C++ version

To avoid build errors when building gRPC and its dependencies (especially from gRPC C++ 1.70 onwards), ensure all CMake builds use the same C++ standard (at least C++17). This is crucial because gRPC C++ 1.70 requires at least C++17, and Abseil, a gRPC dependency, provides different APIs based on the C++ version. For instance, `absl::string_view` is implemented differently before and after C++17. Using a consistent C++ version prevents inconsistencies and ensures compatibility across the project.

Not just abseil…!



Not just abseil…! (cont'd)

README BSL-1.0 license

In a nutshell

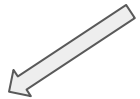
optional lite is a single-file header-only library to represent optional (nullable) objects and pass them by value. The library aims to provide a C++17-like optional for use with C++98 and later. If available, std::optional is used. There's also a simpler version, *optional bare*. Unlike *optional lite*, *optional bare* is limited to default-constructible and copyable types.

From https://github.com/martinmoene/optional-lite

Static constexpr data members

```cpp
#pragma once

struct A {
    static constexpr int x{ -1 };
};

void f(const int &x);
```

lib.h



No longer required in C++17

```cpp
#include "lib.h"

constexpr int A::x;

void f(const int& x) {}
```

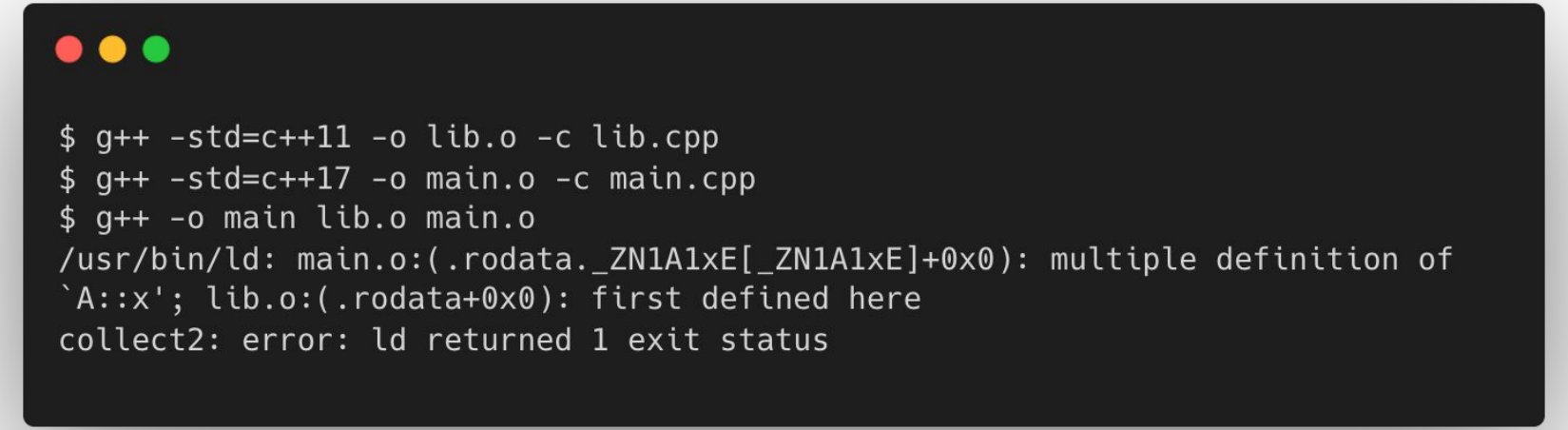
lib.cpp

```cpp
#include "lib.h"

int main() {
    f(A::x);
    return 0;
}
```

main.cpp

From https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101957

Static constexpr data members



```
$ g++ -std=c++11 -o lib.o -c lib.cpp
$ g++ -std=c++17 -o main.o -c main.cpp
$ g++ -o main lib.o main.o
/usr/bin/ld: main.o:(.rodata._ZN1A1xE[_ZN1A1xE]+0x0): multiple definition of
`A::x'; lib.o:(.rodata+0x0): first defined here
collect2: error: ld returned 1 exit status
```

From https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101957

MSVC runtimes

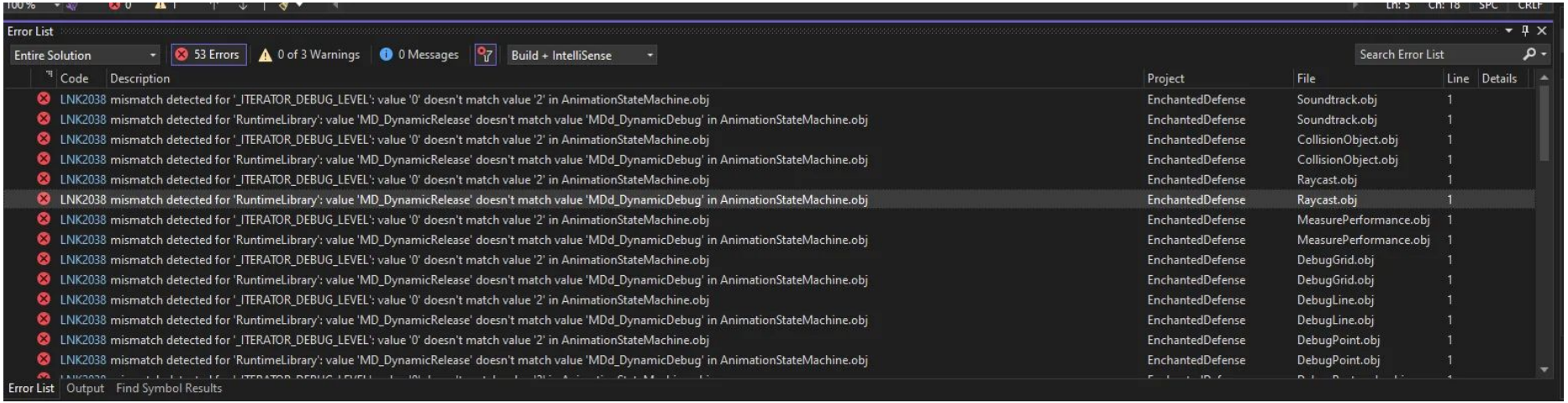
- Unlike what we're used to for libstdc++ and libc++
- MSVC has 4 different runtime libraries

/MD (dynamic)	/MT (static)
/MDd (dynamic, debug)	/MT (static, debug)

All modules passed to a given invocation of the linker must have been compiled with the same runtime library compiler option

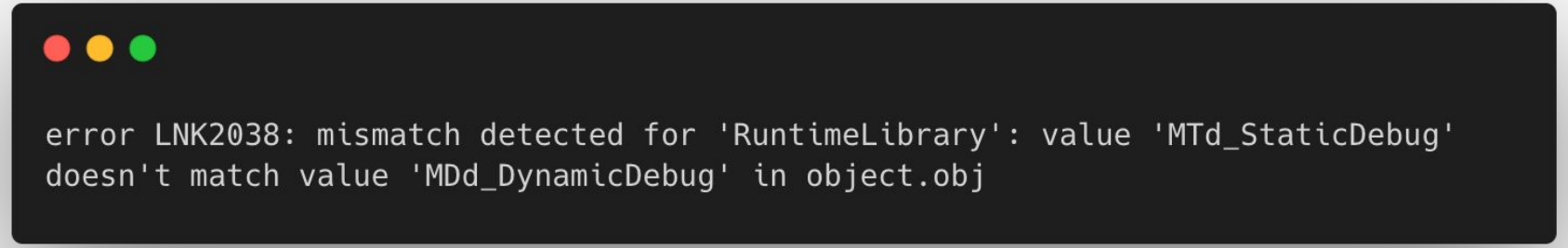
Visual C++ documentation

MSVC runtimes - link errors



From https://www.reddit.com/r/Cplusplus/comments/19c74m6/error_iterator_debug_level_value_0_doesnt_match/

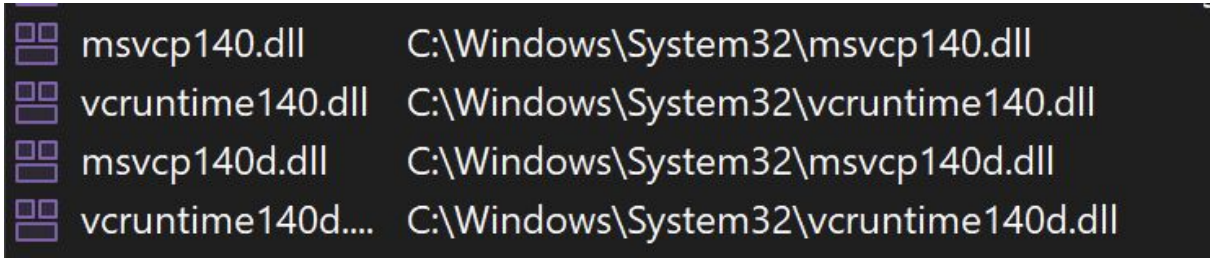
MSVC runtimes - link errors



MSVC runtimes - runtime mismatch

It’s possible to link programs with DLLs that were built for a different runtime, e.g

- Program with /MD
- Library with /MDd (debug runtime)



Depending on specific function calls, the program can malfunction, even without crashing

MSVC v14x toolsets

- Visual Studio 2015: v140
- Visual Studio 2017: v141
- Visual Studio 2019: v142
- Visual Studio 2022: v143

Each is backwards compatible with the previous

Linking only works with objects created by the same version or earlier versions

MSVC v14x toolsets

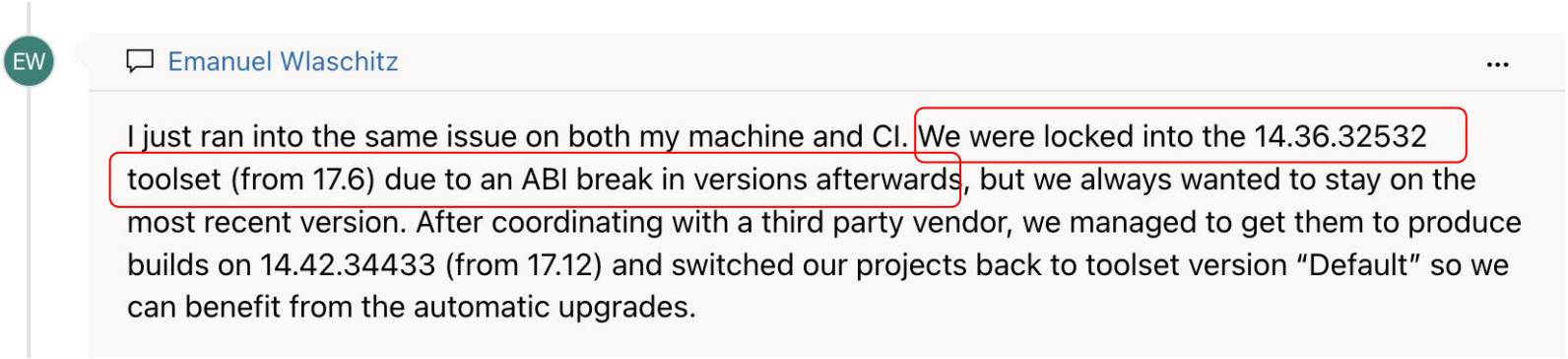
Linking must be done with the **most recent version**

```
absl_log_internal_message.lib(log_message.cc.obj) : error LNK2019: unresolved external symbol
__std_find_last_of_trivial_pos_1 referenced in function "unsign ed __int64 __cdecl
std::_Find_last_of_pos_vectorized<char,char>(char const * const,unsigned __int64,char const *
const,unsigned __int64)" (??$_Find_last_of_p os_vectorized@DD@std@@YA_KQEBD_K01@Z) [protoc.vcxproj]
absl_strings.lib(str_replace.cc.obj) : error LNK2019: unresolved external symbol __std_search_1
referenced in function "class std::vector<struct absl::lts_20
250127::strings_internal::ViableSubstitution,class std::allocator<struct
absl::lts_20250127::strings_internal::ViableSubstitution> > __cdecl absl::lts_202501
27::strings_internal::FindSubstitutions<class std::initializer_list<struct std::pair<class
std::basic_string_view<char,struct std::char_traits<char> >,class
std::basic_string_view<char,struct std::char_traits<char> > > > >(class
std::basic_string_view<char,struct std::char_traits<char> >,class std::initializer_li st<struct
std::pair<class std::basic_string_view<char,struct std::char_traits<char> >,class
std::basic_string_view<char,struct std::char_traits<char> > > > c onst &)" (??$FindSubstitutions@V?
$initializer_list@U?$pair@V?$basic_string_view@DU?
$char_traits@D@std@@@std@@V12@@std@@@std@@@strings_internal@lts_20250127@a bsl@@YA?AV?
$vector@UViableSubstitution@strings_internal@lts_20250127@absl@@V?
$allocator@UViableSubstitution@strings_internal@lts_20250127@absl@@@std@@@std@@V ?$basic_string_view@DU?
$char_traits@D@std@@@4@AEBV?$initializer_list@U?$pair@V?$basic_string_view@DU?
$char_traits@D@std@@@std@@V12@@std@@@4@@Z)(f, seed, [])
}
```

MSVC v14x toolsets

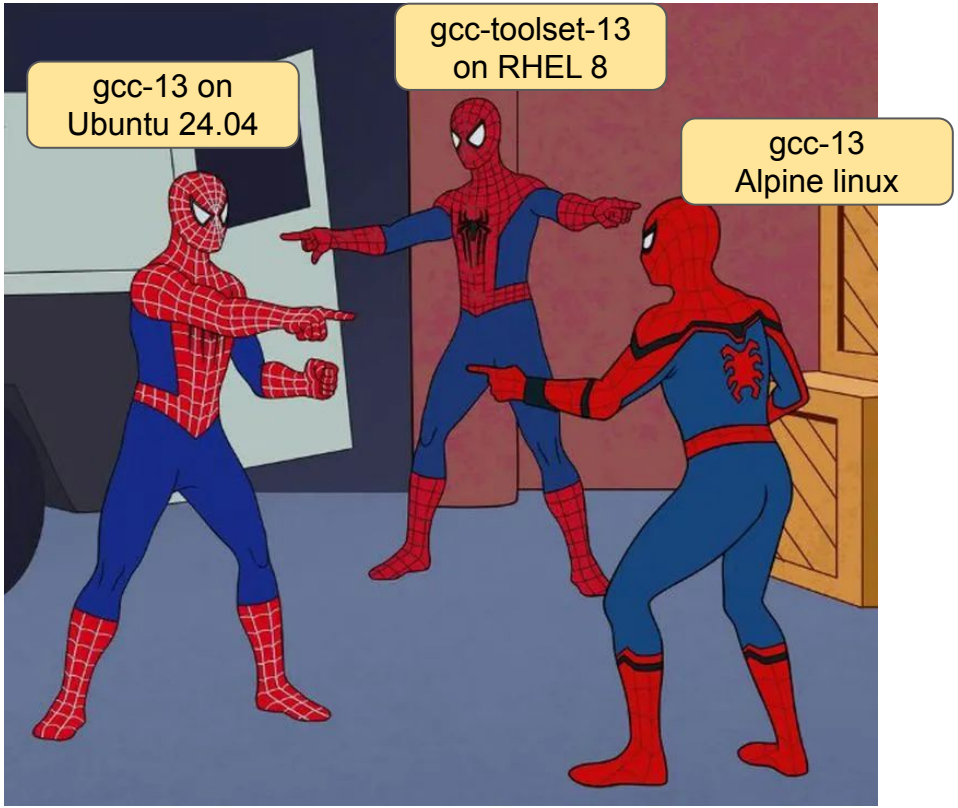
Each is backwards compatible with the previous

Except when it isn't

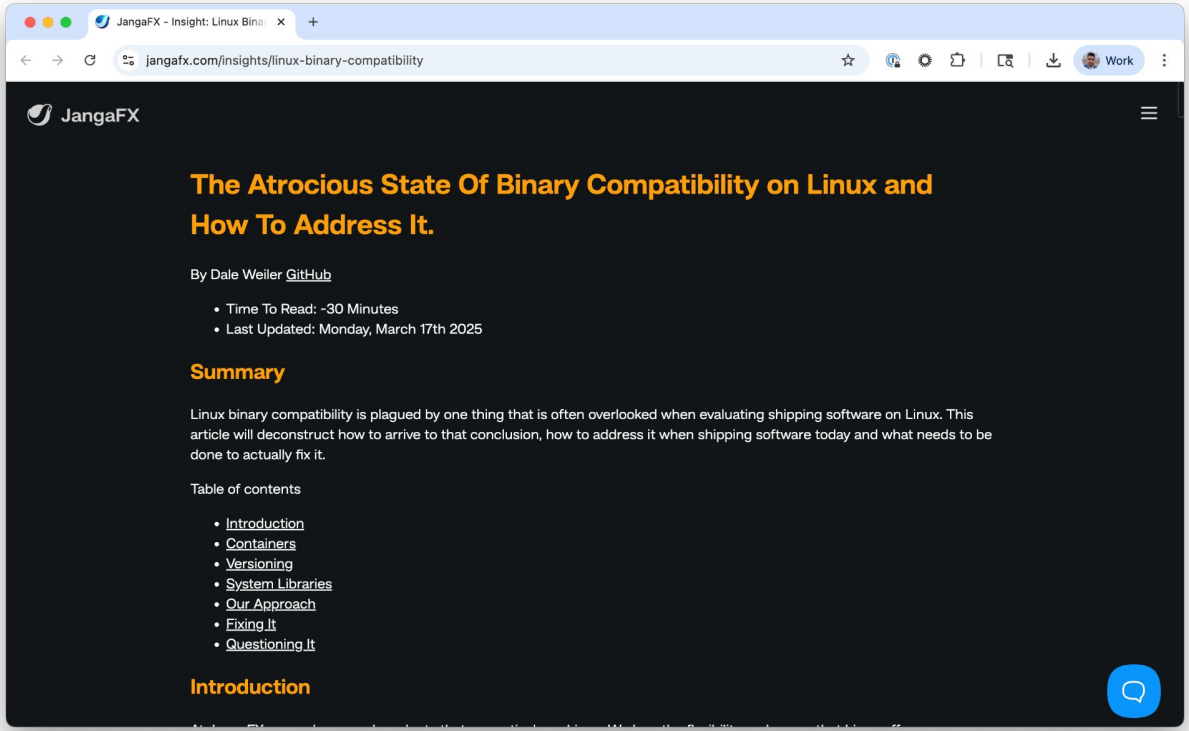


From
https://developercommunity.visualstudio.com/t/MicrosoftVCToolsVersionv143defaulttx/10041951

g++ compatibility on Linux



Binary compatibility on Linux



From https://jangafx.com/insights/linux-binary-compatibility

Current ABI incompatibilities - summary

- Macro conditionals / polyfills cause ABI incompatibilities between libraries
  - Some are intentional and documented (e.g. Abseil)
  - Some not so much
- There are always corner cases where ABI compatibility is not preserved even when documented to work
  - gcc static constexpr members pre-c++17
  - User reports with MSVC versions
- We successfully (?) deal multiple, incompatible MSVC runtimes
- Developers still face cryptic linker errors for all of the above
- Libraries built against libstdc++ for a given gcc version are not necessarily link or runtime compatible

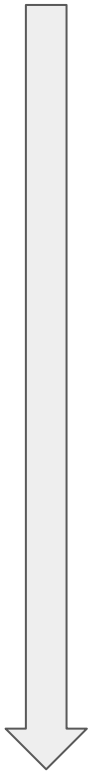
Future ABI breaks

- MSVC STL:
  - STL vNext
  - https://github.com/microsoft/STL/wiki/vNext-Planning
  - Several users are actually requesting an ABI break!
- LLVM libc++:
  - `LIBCXX_ABI_UNSTABLE` can be used at build time to include latest ABI changes (not set in stone)
- GNU libstdc++:
  - https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html
  - There have been 10+ ABI breaks since the gcc 5.x “c++11 ABI” !

https://github.com/microsoft/STL/wiki/vNext-Planning

Dealing with incompatible ABIs

ABI break from a developers perspective



Obvious linker error - when mixing objects with different ABIs E.g. “You’re mixing objects with different ABIs!!!”

Non-evident linker error

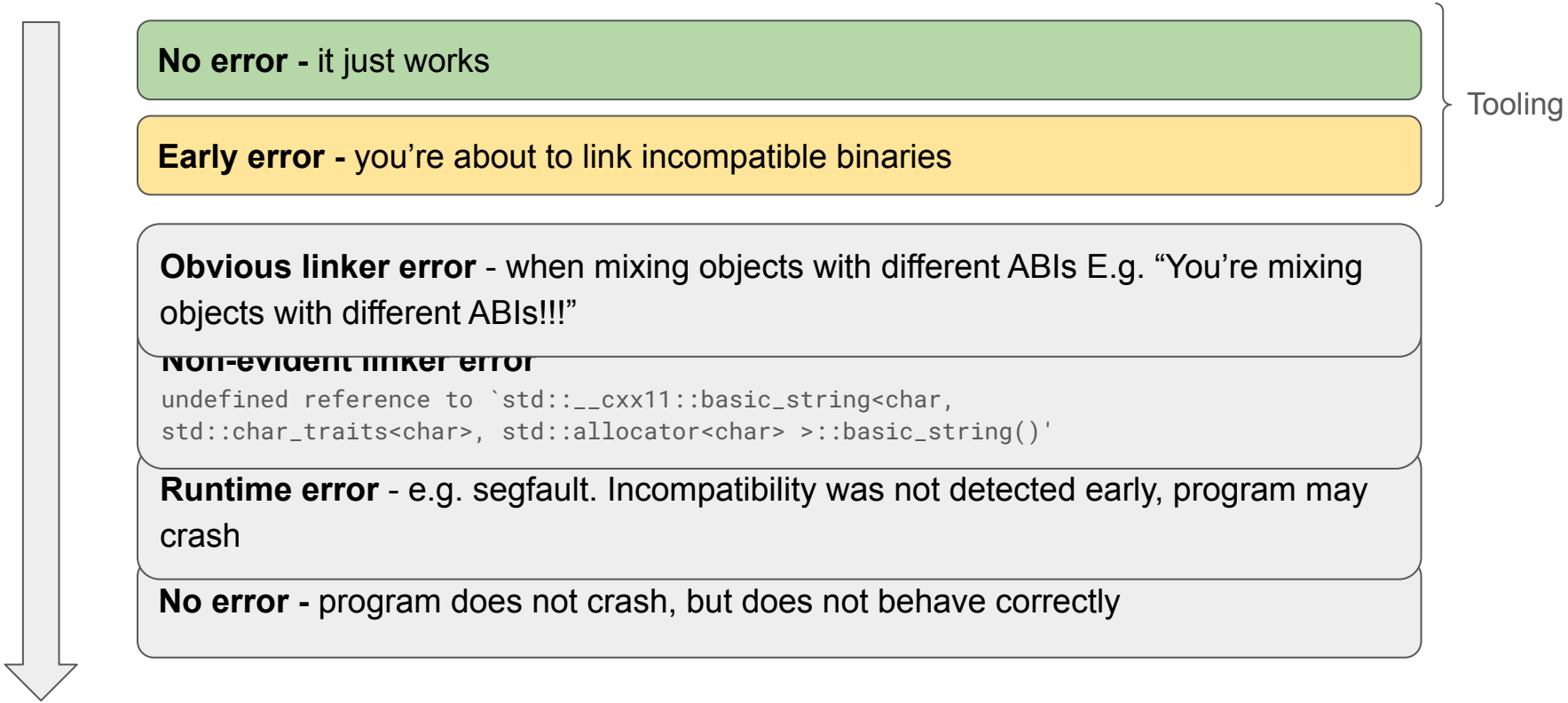
```
undefined reference to `std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::basic_string()'
```

Runtime error - e.g. segfault. Incompatibility was not detected early, program may crash

No error - program does not crash, but does not behave correctly

Less preferable

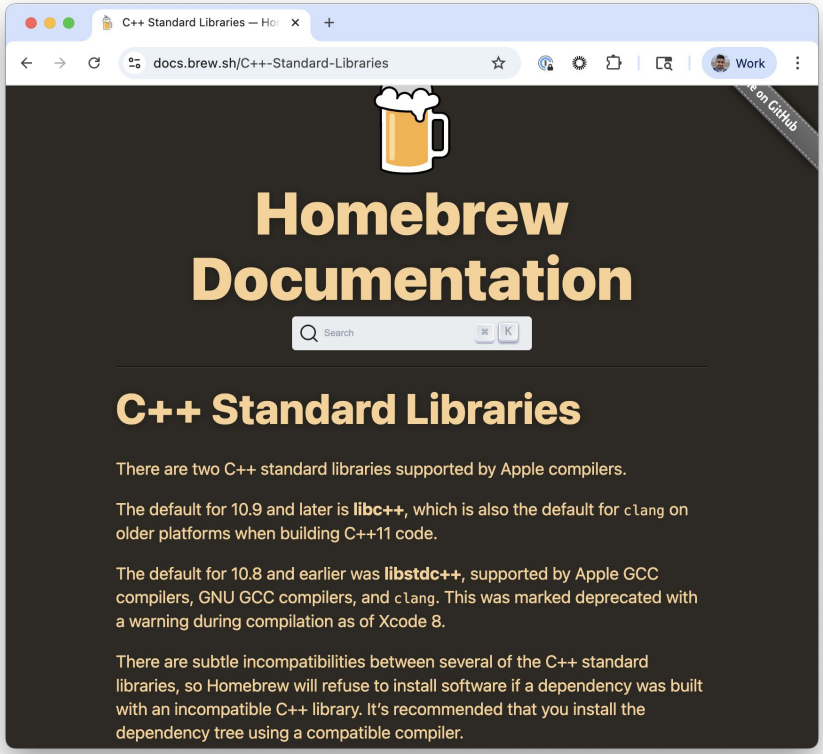
ABI break from a developers perspective



Tooling to the rescue

- **Good**
  package manager or build system will alert of an incompatibility before exposing user to obscure linker errors
- **Better**
  incompatible binaries are not even considered, only compatible binaries are offered
- **Best**
  Missing, compatible binaries are built from source if requested
- **Bonus**
  Support custom “ABIs” with narrow compatibility guarantees: LTO, ASAN, libc++ on Windows…

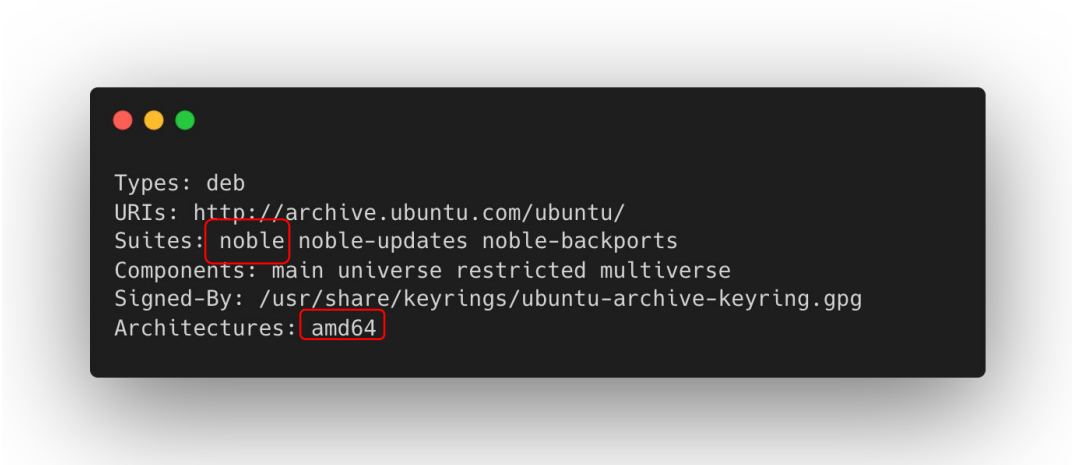
Homebrew



"Homebrew will refuse to install software if a dependency was built with an incompatible C++ library."

ca. 2013

Linux distro package managers, e.g. apt

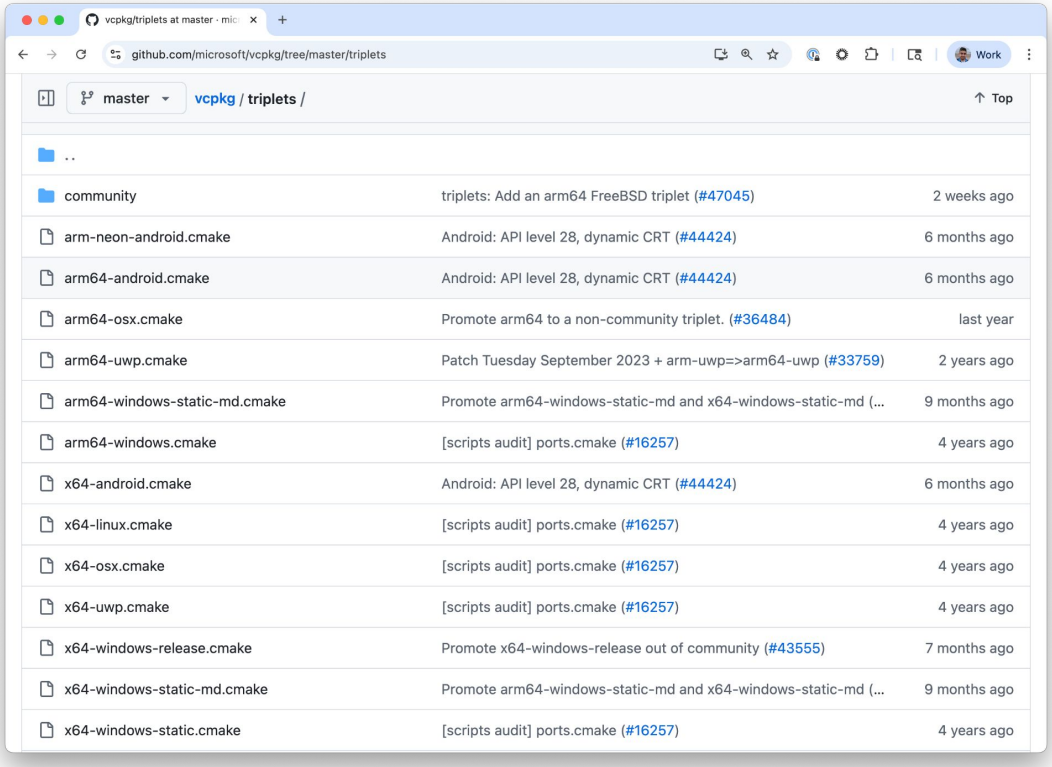


Distro + version codename + arch define a “compatibility island”:

Ubuntu Noble:

- gcc 13 (defaults to c++17)
- glibc 2.39

vcpkg triplets



Triplets are used to abstract a target configuration (cpu, OS, compiler, runtime)

Can be customized, e.g. ASAN, LTO…

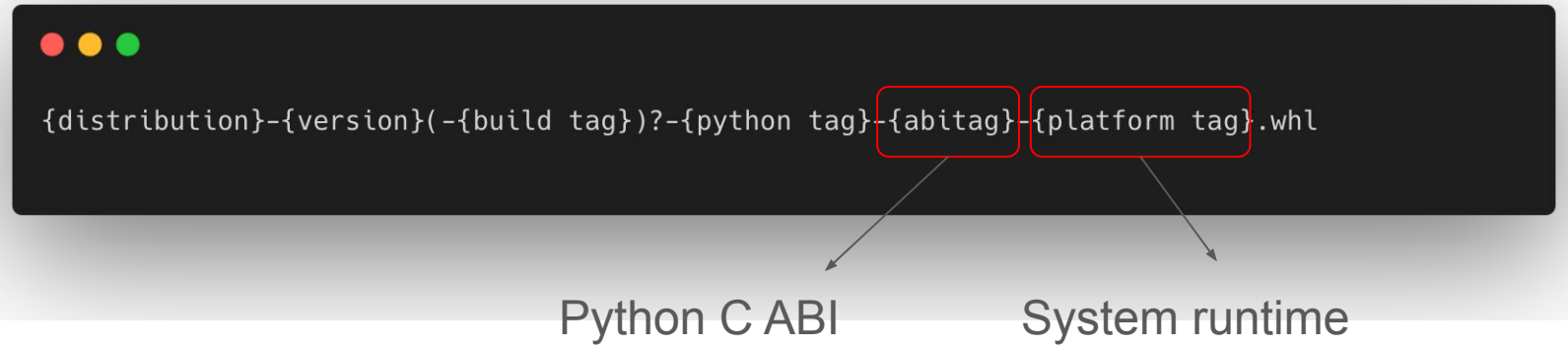
… or if a newer C++ standard required a newer ABI

vcpkg binary caching

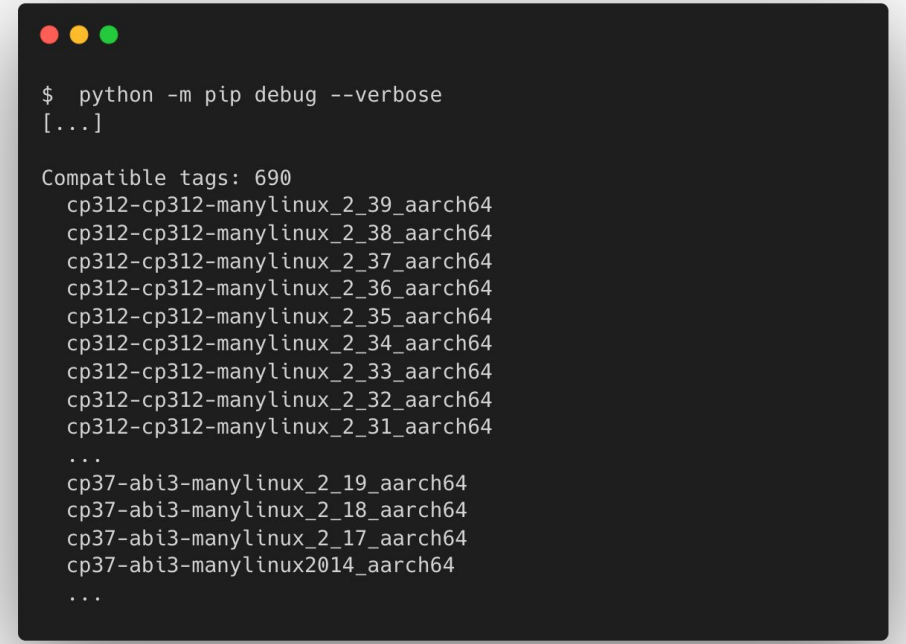
https://learn.microsoft.com/en-us/vcpkg/users/binarycaching

Python and pip

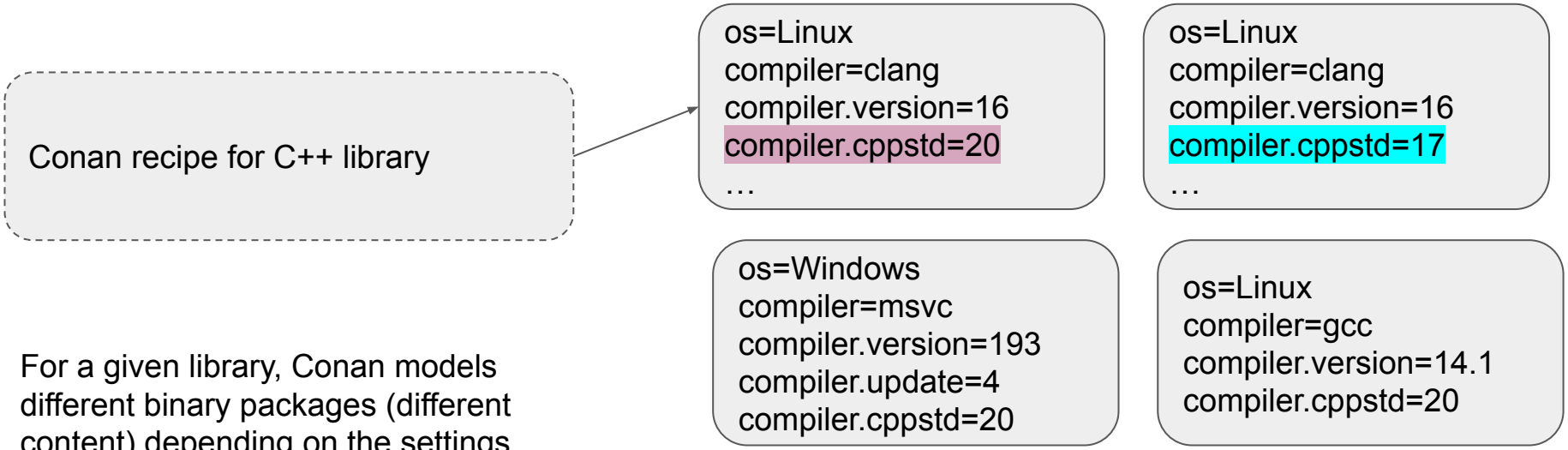
- Python modules can interact with loadable modules
  - Can make calls to C++ via C interfaces
- pip can install binary modules (rather than pure python) and is has some exposure to ABI concerns



Python and pip



Conan



For a given library, Conan models different binary packages (different content) depending on the settings specified in the input profile.

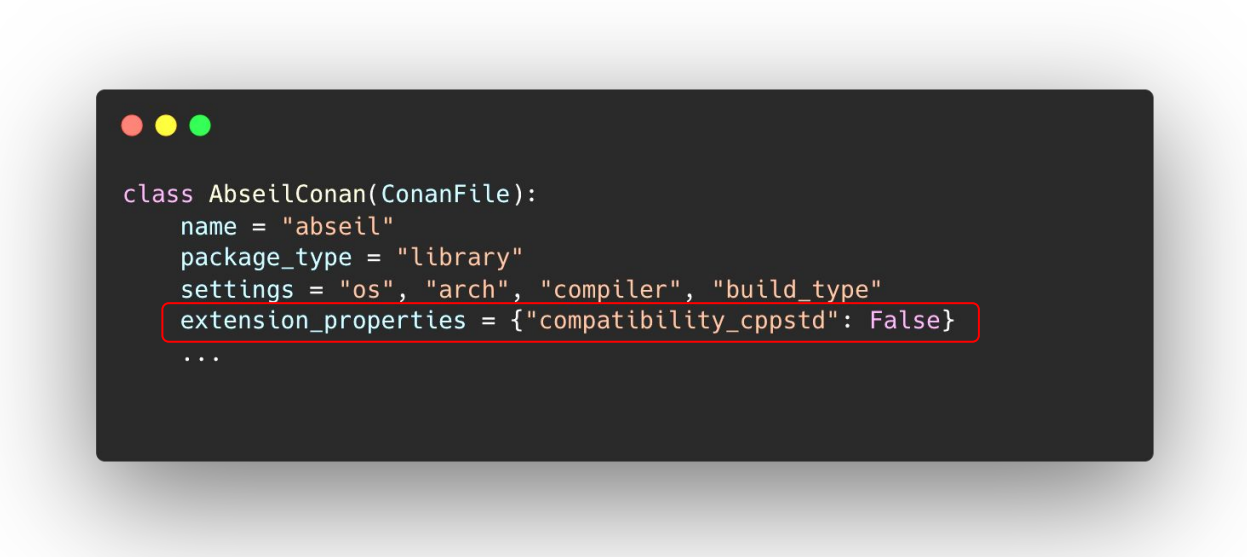
Conan 2 - `compiler.cppstd` setting

Conan compatibility fallback

```
$ conan install --require=boost/[*]

======== Input profiles ========
Profile host:
[settings]
arch=armv8
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=13
os=Linux

[...]

======== Computing necessary packages ========
boost/1.88.0: Main binary package '23db33171d5741f5e5a340b40f35a80f73611b4b' missing
boost/1.88.0: Checking 7 compatible configurations
boost/1.88.0: Compatible configurations not found in cache, checking servers
boost/1.88.0: '683c3d8d0b9f2523c882f161e98c9036b6a73e64': compiler.cppstd=14
boost/1.88.0: '02c54f3e19d6613c7b897b62fe89a7c61ae12a70': compiler.cppstd=gnu14
boost/1.88.0: '1c8eee7edf49cb8e031b804e01bd6573b99af56e': compiler.cppstd=17
boost/1.88.0: 'f3e696a9da1e5c340fed021f34d94a56a5312c9f': compiler.cppstd=20
boost/1.88.0: 'cb10c80b5ad330556f11704d158384f65a9b8a2e': compiler.cppstd=gnu20
boost/1.88.0: '338a35bac06adb183e4c6f4efe3266a342ac3753': compiler.cppstd=23
boost/1.88.0: '13128f873de0be0215e67c7a4b6ff93cbb87a69e': compiler.cppstd=gnu23
```

- By default, only binaries for the same compiler version are considered
- But binaries with with other C++ standard modes are considered as compatible
- If a new ABI came about around any variable (compiler version, or C++ standard), Conan can be configured to treat these as incompatible

Conan - Abseil, protobuf and gRPC



```
class AbseilConan(ConanFile):
    name = "abseil"
    package_type = "library"
    settings = "os", "arch", "compiler", "build_type"
    extension_properties = {"compatibility_cppstd": False}
    ...
```

Abseil recipe - Disable the compatibility extension
Only consider binaries built with the exact same C++ standard as requested

Conan - Abseil, protobuf and gRPC (cont'd)

```
class Protobuf(ConanFile):
    name = "protobuf"
    package_type = "library"
    settings = "os", "arch", "compiler", "build_type"

    def validate(self):
        abseil_cppstd = self.dependencies.host['abseil'].info.settings.compiler.cppstd
        if abseil_cppstd != self.settings.compiler.cppstd:
            raise ConanInvalidConfiguration("Protobuf and abseil must be built with the same compiler.cppstd setting")
```

Prevent creating binary packages of Protobuf where the C++ standard does not match the one that abseil was built with

Conan - Abseil, protobuf and gRPC (cont'd)

- Before:
  - Assume any build of abseil is compatible
  - Potential for compiler and linker errors
- After:
  - Support multiple ABI-incompatible builds of Abseil
  - User will get one of the following:
    - "Missing binary for the requested configuration"
    - **"`--build=missing`" - just works without link errors** 😃
    - Incompatible settings error (only for users forcing a mismatch manually)

What has improved in the past 10 years

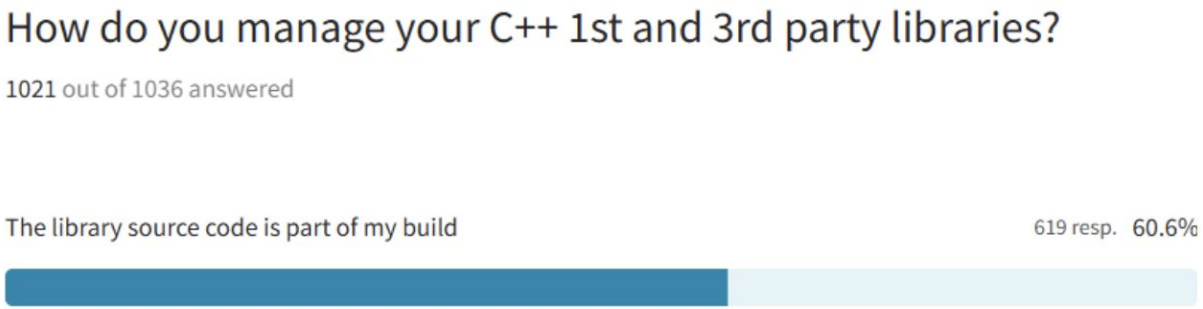
- It’s easier than ever to build a dependency graph from source in a consistent way:
  - Conan (2015)
  - Vcpkg (2016)
  - Bazel
  - CMake `FetchContent` (2018)
- Easier for software vendors to build software using different compiler/versions/platforms:
  - Docker! (first released in 2013)
  - Visual Studio license changes
  - Visual Studio installer (ability to install earlier toolsets)

Thoughts and conclusions

- The last “big” ABI breaks happened 10-13 years ago
  - Visual Studio 2015
  - libstdc++ C++11 ABI
  - macOS libc++ migration
- But disruptive link incompatibilities never went away
  - Difficult to diagnose from linker errors alone
- Longterm backwards compatibility is possible but comes with tradeoffs
  - E.g. don’t use PIE executables, or force the pre-C++11 libstdc++ ABI
- Tools have evolved in the past 10 years to work around the problem altogether
  - By rebuilding from source!
  - And making it easier to build binaries to target multiple configurations

Building from source

- The “solution” to most link incompatibilities (incl. ABI breaks) is to (re)build from source.
- Not everybody can do this - e.g. closed source libraries
  - Software vendors should have access to the same tool to support their users
  - Have a good agreement with your supplier!



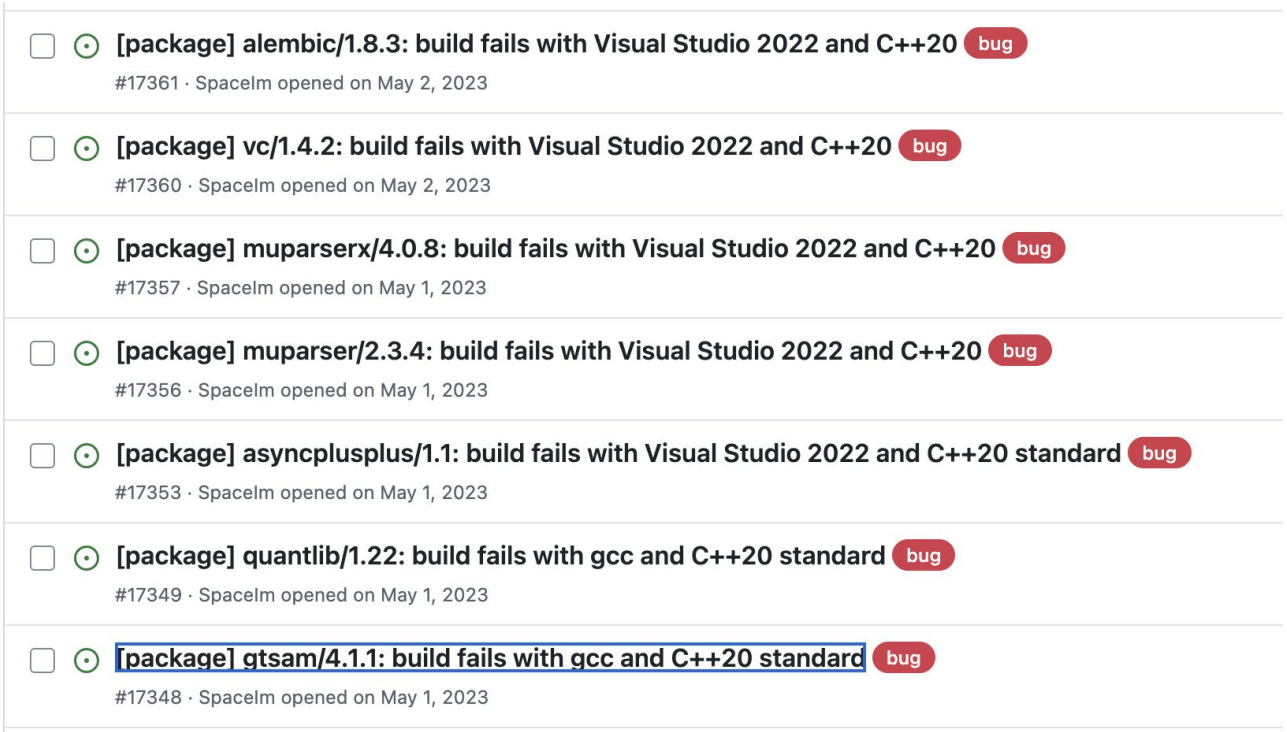
2025 Annual C++ Developer Survey "Lite"

Building from source (cont'd)

3.2 "No gratuitous incompatibilities with C" [or previous C++]

100% seamless friction-free source compatibility with older C++ must always be *available*.

Building from source (cont'd)



Building from source (cont’d)

Changing /std:c++14 to /std:c++20 – How Hard Could It Be?

Monday September 15, 2025 15:15 - 16:15 MDT

Summit 2/3

by Keith Stockdale

Thank you

Any questions?

Could We Handle an ABI Break Today?

Luis Caro Campos
JFrog