Rust ~~C++~~ ~~Swift~~ ABI Resilience?

Rust Nation UK

February 2025

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
Rust Tooling @ Microsoft

Abstract

Is ABI merely an artifact of implementation in native programming languages or should it be considered part of their design?
Some programming languages avoid this commitment, while others are still trying to figure out a path forward.

No, this is not an "ABI - Now or Never" talk. We're taking a different route, by following the design and evolution of the Swift ABI model and seeing what we can learn from it.

From ABI stability & dynamic linking to designing for ABI resilience - a journey through resilient type layout, reabstraction & materialization, resilience in library evolution and (opt-out) performance costs.

What can we learn from Swift's ABI resilience?
How does C++ navigate on this journey?
Can Rust be liberated from the ABI conundrum?

A while back,
at Meeting C++ conference,
in Berlin

I grabbed 2 small bottles of water from the cooler...
... and sat down in one of the afternoon sessions

A while back,
at Meeting C++ conference,
in Berlin

I grabbed 2 small bottles of water
from the cooler...
... and sat down in one of the
afternoon sessions

ABI - Now or Never

In Feb 2020, in Prague, the ISO C++ committee took a series of polls on whether to break ABI, and decided not to... sort of.

There was no applause 😐

"I'm not sure we fully understood what we did and the consequences it could have."

-- not so anonymous C++ committee member

- wg21.link/P2028
- wg21.link/P1863

C++ ABI - Now or Never

February 24, 2020

The Day The Standard Library Died

cor3ntin.github.io/posts/abi/

Design or Implementation Detail?

No, this is not an "ABI - Now or Never" talk 😁

Is ABI merely an artifact of implementation in native programming languages or should it be considered part of their design?

Some programming languages avoid this commitment, while others are still trying to figure out a path forward.

Design or Implementation Detail?

- ABI stability: benefits & risks
- Dynamic linking
- Rust/C++ interop
- Designing for ABI resilience
- Resilience in library evolution
- Performance costs and (opt-out) strategy

How does C++ navigate on this journey?

What can we learn from Swift's ABI resilience?

Can Rust be liberated from the ABI conundrum?

About me

Advanced Installer

Clang Power Tools

Oxidizer SDK

Visual C++

Rust Tooling

@ciura_victor

@ciura_victor@hachyderm.io

@ciuravictor.bsky.social

Disclaimer

I'm just an engineer, with some opinions on stuff...

What is ABI, anyway?

ABI can mean a lot of different things to different people.

Is it platform, hardware, calling conv, language, compilers, std library, your code?

At the end of the day it's a *catch-all* term for "implementation details" that at least two things need to agree on for everything to work.

What is ABI, anyway?

ABI stability isn't technically a property of a programming language.

It's really a property of a system and its toolchain.

ABI is something defined by the *platform*.

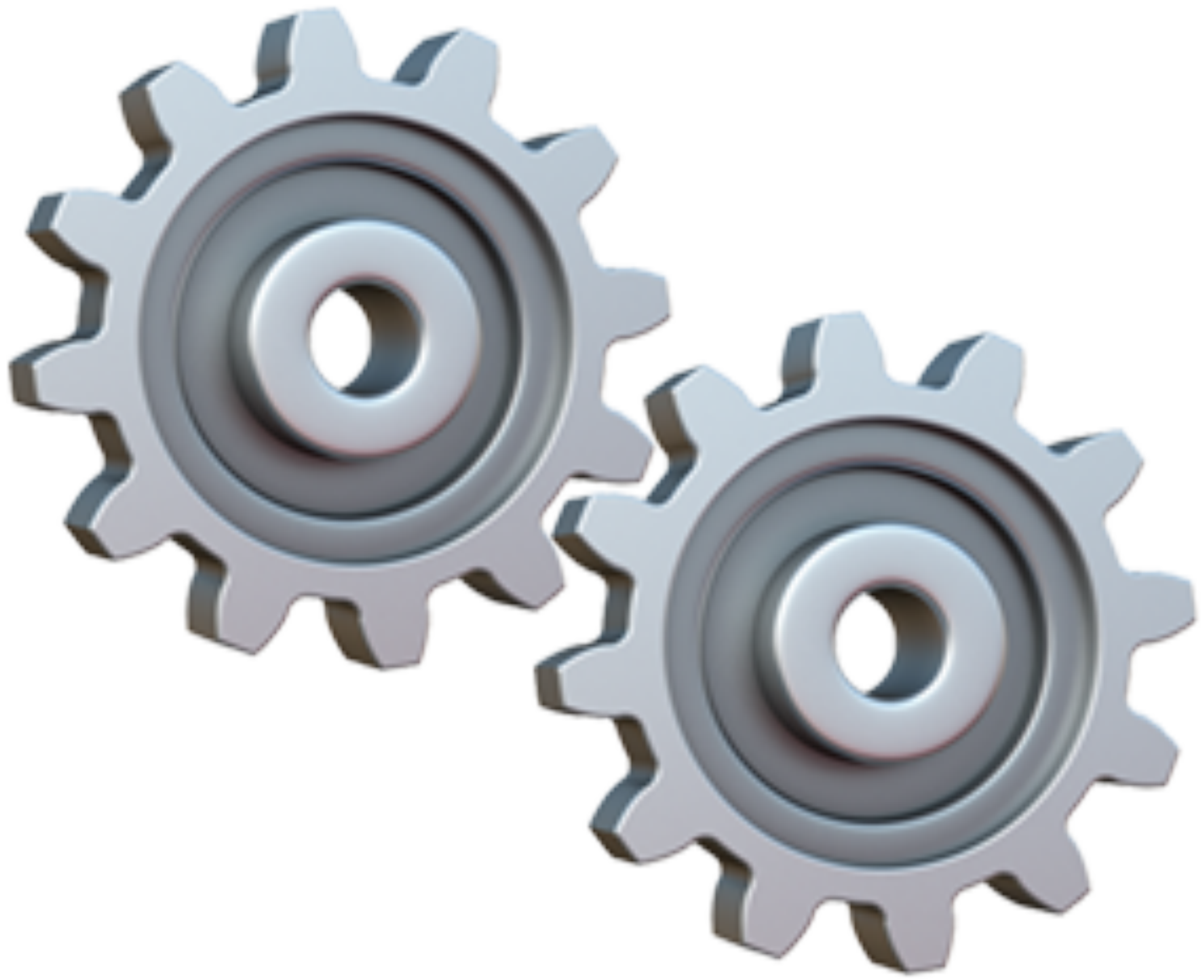
The platform owner can just require you to use a particular compiler toolchain that happens to implement their "stable" ABI.

If you care about *dynamic linking* (shared libraries).

What is ABI, anyway?

- Layout of types
  - size & alignment (stride)
  - offsets & types of fields
  - v-table entries
  - closures
- Calling conventions
- Name mangling (symbols)
- Metadata (if applicable)

ABI Stability - Why?



- You don't have to share the source code of your library
- You can use the most recent compiler for your library
- You don't have to recompile everything (full project visibility)
- Binaries can be shipped and updated independently (patches)
- Multiple programs can share the same library (incl. std lib)
- Plugins/extensions
- Language interop

ABI Stability - When?

- Don't shut the door on future compiler & library improvements
- Stabilizing the ABI (too early)™ might miss optimization opportunities
  - implement a faster custom calling convention
  - implement optimal structure layout
  - improve the way a std utility works

- NB. These are not impossible things!
  - They are just tough engineering problems
  - We need to invest a lot of time and brain power to solve them

ABI Stability - Evolution of Software Libraries

- Developers want to evolve their software libraries without breaking ABI
  - add new functionality
  - fix bugs
  - improve performance
- A lot of these activities can break ABI
  - add a field to a class
  - make changes affecting v-table
  - (re)use existing padding for a new field?

ABI Stability Scorecard

Can we have stable ABI, pretty please?

- Go: NO
- Rust: NO
- Carbon: NO
- Zig: NO
- C++: <always has been meme> 🤷‍♂️ ... but don't tell anyone!
- Swift: YES, since v5.0 (most important thing ever!)

ABI Stability

Zig natively supports C ABIs for *extern* things; which C ABI is used depends on the target you are compiling for (e.g. CPU architecture, operating system).

This allows for near-seamless interoperation with code that was not written in Zig; the usage of C ABIs is standard amongst programming languages.

Zig internally does not use an ABI, meaning code should explicitly conform to a C ABI where reproducible and defined binary-level behavior is needed.

ABI Stability

Go ABI specification

- Go's ABI defines the layout of data in memory and the conventions for calling between Go functions
- This ABI is unstable and will change between Go versions
- If you’re writing assembly code, please instead refer to Go’s assembly documentation, which describes Go’s stable ABI, known as ABI0
- Go uses a *common* ABI design across all architectures (instead of the *platform* ABI)
- All functions defined in Go source follow ABIInternal
  - however, ABIInternal and ABI0 functions are able to call each other through transparent ABI wrappers

ABI Stability

Carbon / non-goals 🙂

github.com/carbon-language/carbon-lang#language-goals

❝ We also have explicit non-goals for Carbon, notably including:

- a stable ABI for the entire language and library
- perfect backwards or forwards compatibility

Stability ∞

The greatest champion of ABI stability and dynamic linking:

C

That's plain old **C**, not Carbon, by the way :)

The C ABI

Many software ecosystems require both long-term ABI stability
<u>and</u> the ability to constantly evolve.

These systems tend to use **C** as the stable ABI

Evolving software components with a C ABI requires to manually and proactively introduce extra levels of indirection, to account for potential future changes.

pimpl

* one more level of indirection solves every problem, right? 😁

- COM interfaces
  - change API to hide implementation changes (break ABI)
    - `IWidgetSomething, IWidgetSomething2, IWidgetSomething3`
  - MIDL for interop
  - metadata

- Objective-C msg-send
  - ~unstructured data
  - type erasure / everything dynamic / indirections
  - swizzling, isa

STD ~ ABI

Consistency

Jonathan Müller @foonathan · Feb 3, 2020

What's the point of standard library containers if they can't give the best performance?

2 · 9

Titus Winters @TitusWinters · Feb 3, 2020

They're common and readily available? (Which does have some value.)

Committing to ABI is like admitting that the standard library is aiming to be McDonald's - It's everywhere, it's consistent, and it technically solves the problem.

5 · 3 · 18

JF Bastien 🔗 **@jfbastien@mastodon.social** @jfbastien · Feb 3, 2020

The rumors of STL pink slime are wildly overblown 🤡

1 · 5

May I have some ABI?

Sean Parent @SeanParent · Feb 4, 2020 ···

A stable ABI means you can link against the platform API, shared library APIs including the standard, evolve your product without breaking plugins. C++ needs a strategy on how to specify and maintain ABI compatibility - not some "one time break" for efficiency. See @SwiftLang

Chandler Carruth ···
@chandlerc1024

You could have an independent mechanism for accessing platform libraries. We (almost) have that on Linux with their C APIs & ABIs.

Pinning all of C++ (and its standard library) down with a stable ABI for the entire thing largely blocks evolving any of them for performance.

May I have some ABI?

Doug Gregor @dgregor79 · Feb 4, 2020

A stable C++ ABI is useless for platform APIs if it doesn't encompass the standard library. That said, you could have a compilation mode choose between resilience (library impl can change without breaking you) or fragility (performance without ABI stability)

Sean Parent @SeanParent · Feb 4, 2020

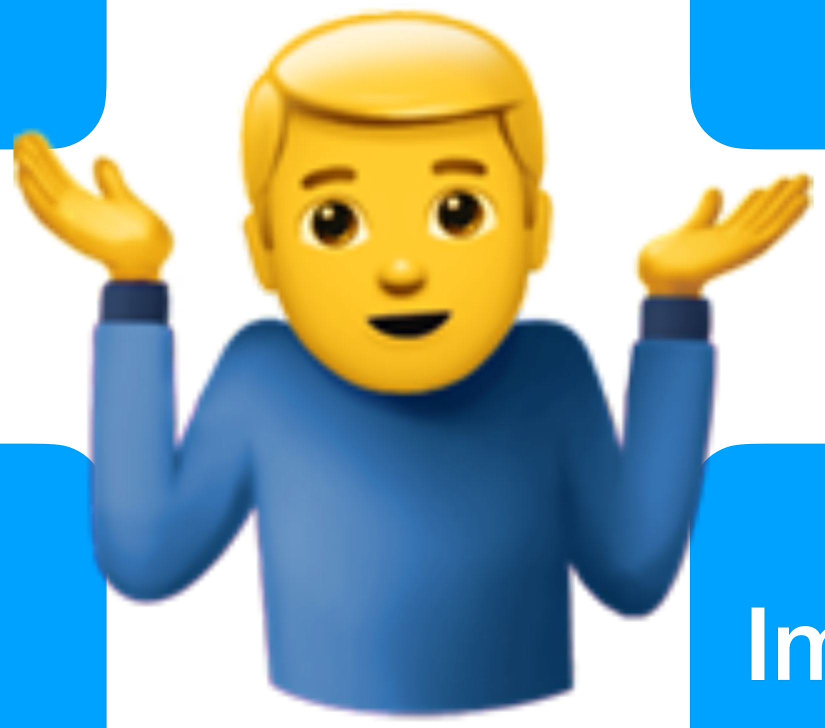
I didn't say lock everything. Define what can be used in an ABI stable interface, and how it is versioned. A single app needs to be able to link against multiple versions of the same lib without an ODR violation. C++ currently is _not_ ABI stable.

twitter.com/TitusWinters/status/1224351257479077889?s=20

C++ the king of mix signals and ambivalent behavior

C++ does not have an ABI resilience model (it's not stable)

The committee will reject any proposal that could cause ABI breaks in existing STL components



C++ will not officially commit to guaranteeing ABI stability

Implementors* will not change/improve library components if it would cause an ABI break for clients

- wg21.link/P2028
- wg21.link/P1863

The king of mix signals and ambivalent behavior

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

- WG21 is not in favor in an ABI break in C++23/26

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

- WG21 is not in favor in an ABI break in C++23/26
- WG21 is in favor of an ABI break in a future™ version of C++ (When?)

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

- WG21 is not in favor in an ABI break in C++23/26
- WG21 is in favor of an ABI break in a future™ version of C++ (When?)
- WG21 "will take time to consider" proposals requiring an ABI break (read as: ignore)

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

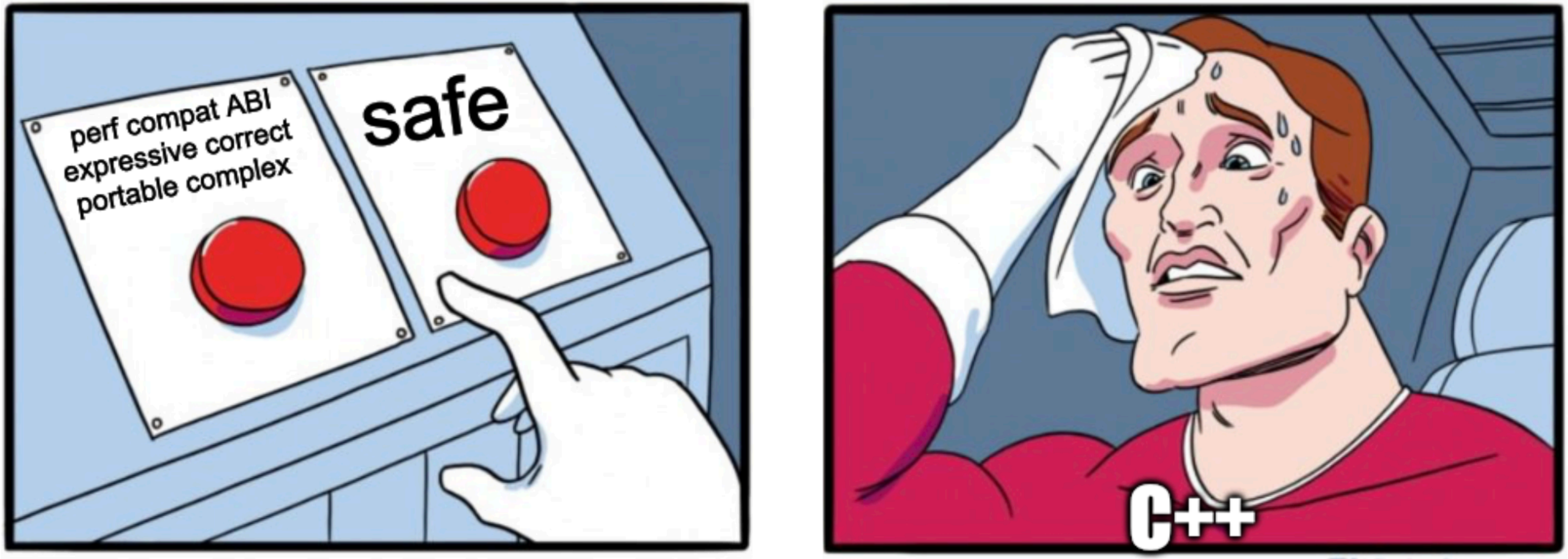
- WG21 is not in favor in an ABI break in C++23/26
- WG21 is in favor of an ABI break in a future™ version of C++ (When?)
- WG21 "will take time to consider" proposals requiring an ABI break (read as: ignore)
- WG21 will not promise stability forever

The king of mix signals and ambivalent behavior

ABI discussions in Prague (Feb 2020):

- WG21 is not in favor in an ABI break in C++23/26
- WG21 is in favor of an ABI break in a future™ version of C++ (When?)
- WG21 "will take time to consider" proposals requiring an ABI break (read as: ignore)
- WG21 will not promise stability forever
- WG21 wants to keep prioritizing performance over stability

The king of mix signals and ambivalent behavior



Is change even possible?

Quick recap: A "**lost decade**" pattern

MSVC 6 — ~12 years

- Shipped in **1998**
- "10 is the new 6" fanfare in **2010**

C99 `_Complex` and VLAs — ~12 years

- Added in **1999**
- Walked them back to "optional" in **2011**

C++11 `std::string` — ~11 years

- Banned RC for std::string in **2008/2010**
- Major Linux distro enabled it in **2019**

Python 3 — ~12 years

- Shipped 3.0 in **2008**
- 10% still using 2.x as of early **2020**

> If you don't build a strong backward compatibility bridge, expect to slow your adoption down by
>
> **~10 years**
>
> (absent other forces)

youtube.com/watch?v=8U3hI8XMm8c

Why do we want to break ABI

Why do we want to break ABI

Quality of implementation fixes:

Why do we want to break ABI

Quality of implementation fixes:

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)
- making std::unordered_map faster or swap the hash algorithm

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)
- making std::unordered_map faster or swap the hash algorithm
- better conformance: some implementations are intentionally not conforming for the sake of stability

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)
- making std::unordered_map faster or swap the hash algorithm
- better conformance: some implementations are intentionally not conforming for the sake of stability
- tweaks to string, vector, and other container layouts

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)
- making std::unordered_map faster or swap the hash algorithm
- better conformance: some implementations are intentionally not conforming for the sake of stability
- tweaks to string, vector, and other container layouts
- std::span, std::string_view, std::unique_ptr need to be spilled into registers for function calls (language changes needed => zero-overhead for x64 call conv.)

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)
- making std::unordered_map faster or swap the hash algorithm
- better conformance: some implementations are intentionally not conforming for the sake of stability
- tweaks to string, vector, and other container layouts
- std::span, std::string_view, std::unique_ptr need to be spilled into registers for function calls (language changes needed => zero-overhead for x64 call conv.)
- improving std::shared_ptr, eg. lock_exclusive()

Why do we want to break ABI

Quality of implementation fixes:

- making std::regex faster (also adding UTF-8 support)
- making std::unordered_map faster or swap the hash algorithm
- better conformance: some implementations are intentionally not conforming for the sake of stability
- tweaks to string, vector, and other container layouts
- std::span, std::string_view, std::unique_ptr need to be spilled into registers for function calls (language changes needed => zero-overhead for x64 call conv.)
- improving std::shared_ptr, eg. lock_exclusive()
- improving perf of std::mutex (std::shared_mutex is faster!)

Clang libc++ ABI stability

There is a path forward:

- libc++ aims to preserve a stable ABI to avoid subtle bugs

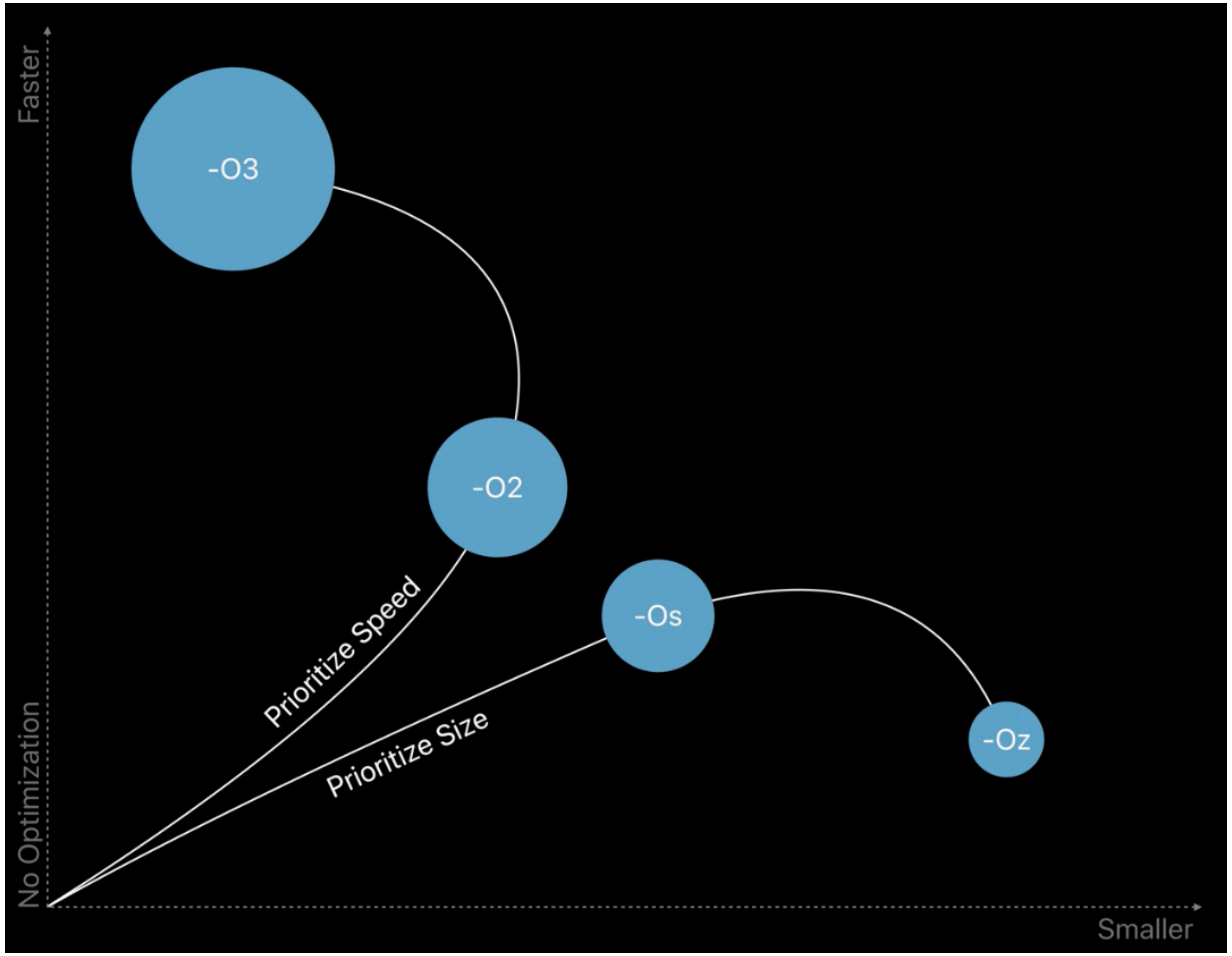
  (when code built under the old ABI is linked with code built under the new ABI)
- libc++ wants to make ABI-breaking improvements/fixes (user opt-in)
- libc++ allows specifying an ABI version at build time:
  - `LIBCXX_ABI_VERSION=`
  - 1 (stable/default); 2 (unstable/next); 3 (when 2 will be frozen)...
- always use the most cutting-edge, most unstable ABI: `LIBCXX_ABI_UNSTABLE`
- All or nothing! solution 😕

📖 Clang docs:
libcxx.llvm.org/DesignDocs/ABIVersioning.html

Design Choices

C++ / Rust / Carbon / ...	Swift
Fast code	Favor small* code
Heavy inlining	Outlining
CPU utilization/saturation	CPU power usage
Mostly* static linking (with occasional DLL madness)	**Dynamic linking (shared libraries)**

Outlining



LLVM Outlininer

–0z

Outlining:
Replacing repeated sequences of instructions with calls to equivalent functions.
(smaller code => icache)

Jessica Paquette "Reducing Code Size Using Outlining"
youtube.com/watch?v=yorld-WSOeU

Jessica Paquette, JF Bastien "What's New in Clang and LLVM"
developer.apple.com/videos/play/wwdc2019/409/

Swift who?

- Ahead-Of-Time (AOT) compiled, but has a large runtime library
- created to replace Objective-C on Apple's platforms (native interop with Obj-C)
- has classes and inheritance
- interfaces, generics, closures, enums with payloads
- Automatic Reference Counting (ARC)
- simple function-scoped mutable borrows (inout)
- emphasis on value semantics
- structs/primitives ("values") are "mutable xor shared" & stored inline
- classes are mutably shared and boxed (using ARC) -> reference semantics
- collections implement value semantics by being CoW (using ARC)

Swift

Language designed for Library Evolution

Swift was designed to explicitly account for a stable ABI.

Swift espouses a *principle of least regret* for public interfaces, ensuring that the implementation details of a software module do not create a binary-compatibility contract that prevents future evolution.

Swift

Language designed for Library Evolution

Principles for ABI-stable library evolution:

- make all promises explicit
- delineate what can and cannot change in a stable ABI
- provide a performance model that indirects only when necessary
- let the authors of libraries & consumers be in control

Doug Gregor
Implementing Language Support for
ABI-Stable Software Evolution in Swift and LLVM
youtube.com/watch?v=MgPBetJWkmc

Evolving a struct

```swift
public struct Person {
  public var name: String
  public let birthDate: Date?
  let id: Int
}
```

```swift
public struct Person {
  let id: Int
  public let birthDate: Date?
  public var name: String
}
```

```swift
public struct Person {
  let id: UUID
  public var birthDate: Date?
  public var name: String
}
```

```swift
public struct Person {
  let id: UUID
  public var birthDate: Date?
  public var name: String
  public var favoriteColor: Color?
}
```

- Person struct changes size when new fields are added
- Offset of fields changes whenever layout changes

Using the struct

```
import PersonLibrary
struct Classroom {
  var teacher: Person
  var students: [Person]

  func getTeacherName() -> String { teacher.name }
  var numStudents: Int { students.count }
}
```

array

offset

The Library

Type Layout should be as-if we had the whole program:

- *Person* library should layout the type <u>without</u> indirection
- Expose metadata with layout information:
  - size/alignment of type
  - offsets of each of the public fields

```
size_t Person_size = 32;
size_t Person_align = 8;
size_t Person_name_offset = 0;
size_t Person_birthDate_offset = 8;
```

Client/External Code

Client code (external) indirects through layout metadata

- Access a field:
  - read the metadata for the field offset
  - add that offset to the base object
  - cast the new pointer and load the field
- Store an instance on the stack:
  - read the metadata for instance size
  - emit alloca instruction, to setup as needed

Library Code

Library code (internal) eliminates all indirection

- Access a field:
  - ~~read the metadata for the field offset~~
  - add that offset to the base object
  - cast the new pointer and load the field
- Store an instance on the stack:
  - ~~read the metadata for instance size~~
  - emit alloca instruction, to setup as needed

LLVM dynamically-sized things

- LLVM's support for dynamically-sized things on the stack has been good for Swift
- Swift makes heavy use of this for of ABI-stable value types:
  - you have local variable of some struct defined in an ABI-stable library
  - so you don't know it's size until load time
- Dynamic allocs can handle this nicely (with minimal perf impact)
- C++ desperately want all objects to have compile-time-constant size
- The notion of sizeof/alignof being runtime values just grates against the whole C++ model :(

sfba.social/@dgregor79/111058162167016107

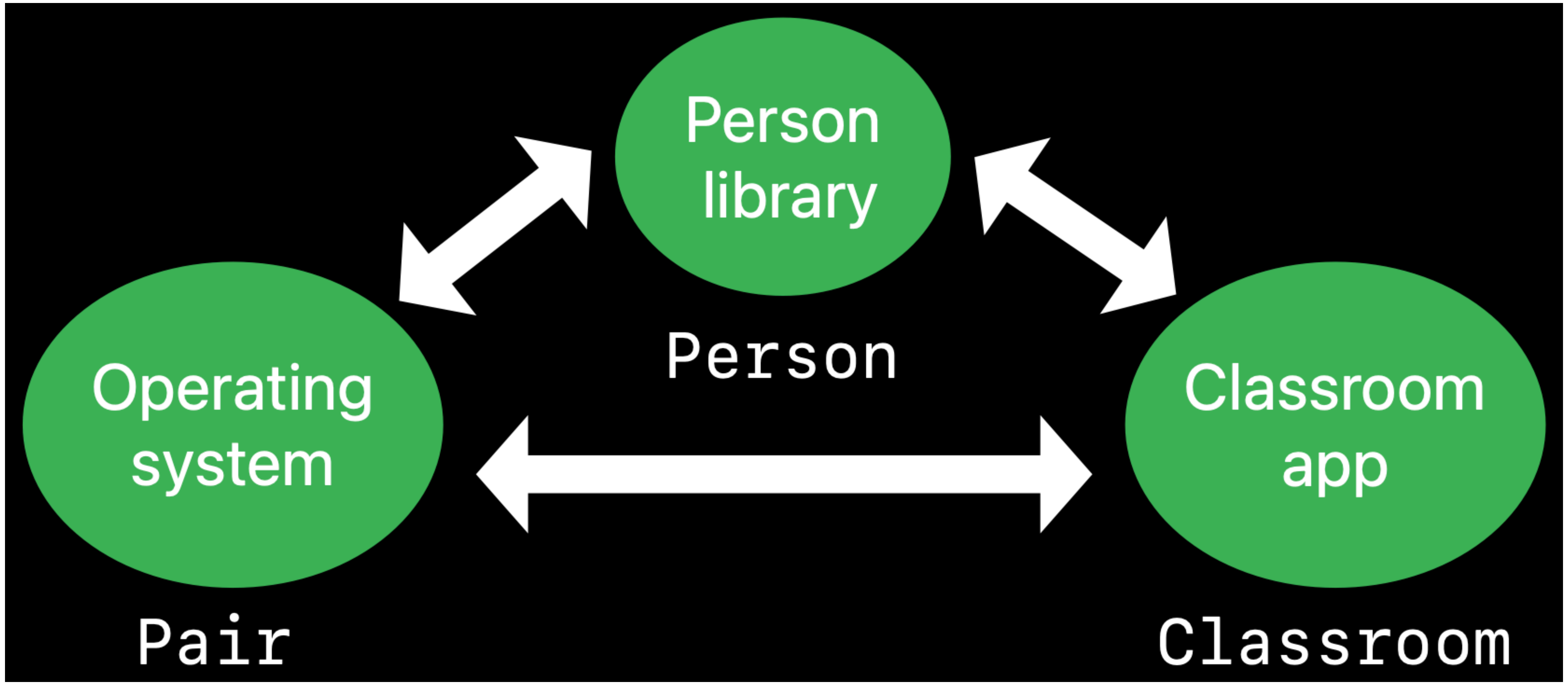
Resilience Domains

By explicitly modeling the boundaries between software modules that evolve separately vs. together:

- Swift is able introduce appropriate indirections across separately-evolved software modules

- while optimizing away that indirection within software modules that are always compiled together

Resilience Domains



A resilience domain contains code that will always be compiled together.

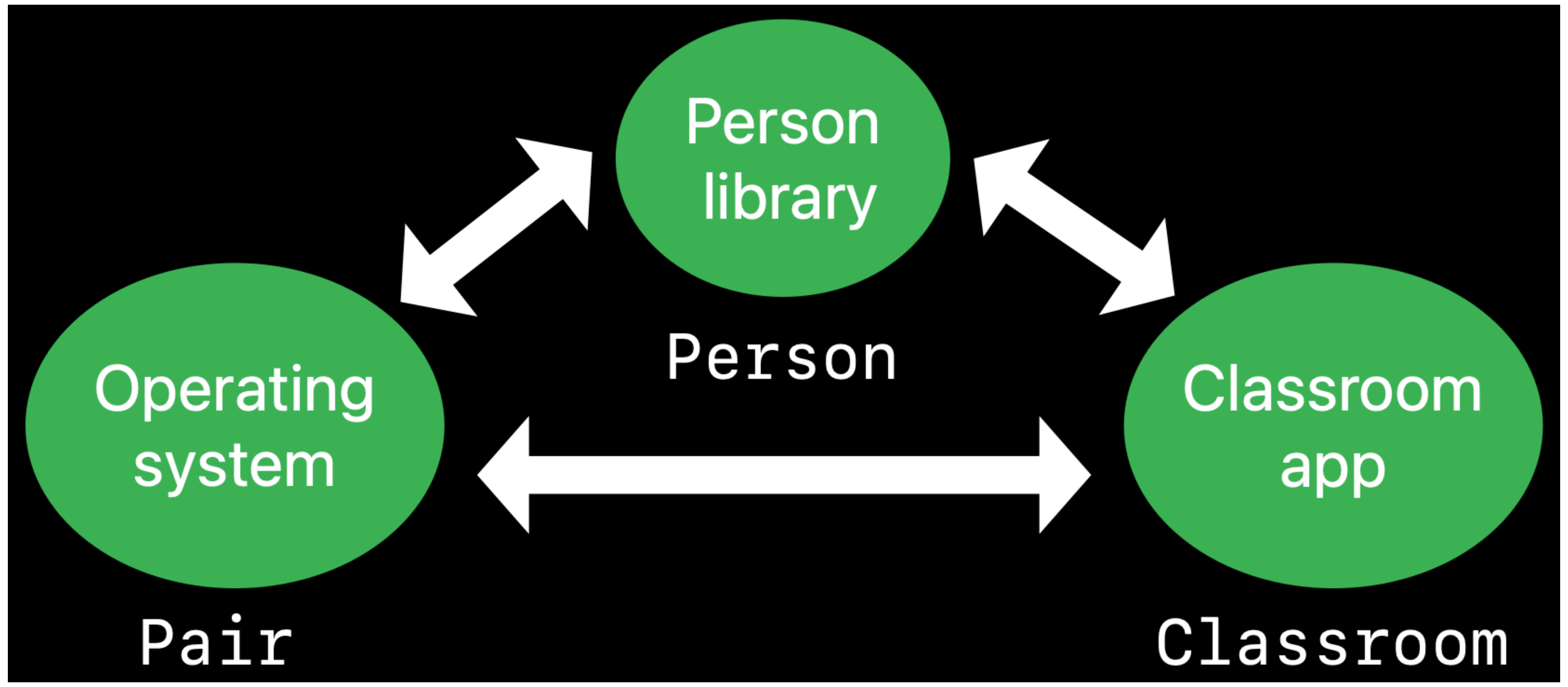
A program can be composed of many different resilience domains.

Resilience domains control where the costs of ABI stability are paid.

Resilience Domains

Optimization and Resilience Domains

- Across resilience domains => maintain stable ABI
- Within a resilience domain => all implementation details are fair game
  - no indirections (direct access, no computed metadata)
  - no guarantees made
- Optimizations need to be aware of resilience domain boundaries



Resilience Domains

What if there is only 1 resilience domain?

- There are no ABI-stable boundaries
  - all type layouts are *fixed* at compile time
  - stable ABI is completely irrelevant
- You don't pay for library evolution when you don't use it

Resilient Type Layout

By <u>default</u>, a type that is defined by a dylib has a resilient layout.

- size, alignment, stride of that type aren't statically known to the application
  - it must ask the dylib for that type's value witness table (at runtime!)
- value witness table is just the "vtable" of stuff you might want to know about any type
- this results in resilient types having to be "*boxed*" and passed around as a pointer
  - not quite... (<u>details</u> are interesting)
- inside the boundaries of the dylib
  - where all of its own implementation details are statically known
  - the type is handled as if it wasn't resilient (no indirections & perf costs)

Escape Hatches

Swift ABI resilience is the DEFAULT (for libraries).

You have to Opt-Out of Resilience, if you don't want it.

Escape Hatches

Trading future evolution for client performance:

- Explicit inline code exposed into the client
  - enables caller optimization, generic specialization
  - prevents any changes to the function’s semantics

```
@inline public func swapped()
{
}
```

Escape Hatches

Trading future evolution for client performance:

- Fixed-layout types promise never to change layout
  - enables layout of types in client code
  - gives-up ability to add/remove/reorder fields

```
@fixedLayout
public struct Pair<First, Second>
{
}
```

Famous last words: *"This type will never need to change"*
-- author unknown 😵

Swift Challenges

- Large runtime component (with compiler abilities)
  - Runtime type layout
  - Handling metadata at runtime
  - Witness tables & indirections
  - Generics<T> are particularly hard (monomorphization, reabstraction)
- Every language feature is a bit harder to design (resilient)
- Older Swift runtimes might not support new language features (OS targets)

Go in depth:
faultlore.com/blah/swift-abi/

Rust ABI Stability

Rust dev: "Can we have stable ABI?"

Rust dev: "We have stable ABI at home."

Rust ABI Stability

Rust dev: "Can we have stable ABI?"

Rust dev: "We have stable ABI at home."

Stable ABI at home: `#[repr(C)]`

The C ABI - Cost

Rust FFI itself is "zero cost" in that it has the same performance characteristics as C(++) calling C(++) code.

🐌 Where you can run into a cost is if you have to convert some of your internal data structures into a C-ABI friendly representation.

Eg.
If you use Rust strings String/&str but your FFI layer really wants UCS-2 strings (platform)
=> you'll pay the conversion cost in order to do the FFI itself

Rust ABI

Status quo: `repr(C)` - fake it, till you make it 😀

- Using the C calling convention for function definitions and calls `extern "C" fn`
- Using the C data layout for a type `#[repr(C)]`
- Definitions of C types like char, int, long, etc. `std::ffi::c_*`
- Exporting an item under a stable linking symbol `#[no_mangle]`
- Limited to C types, mostly
- No slices

```
u8, i64, c_int, c_char, ...
&T, &mut T
*const T, *mut T
struct
```

Rust ABI

The Future: calling convention and data layout

- Stable calling convention that supports common data types `extern "crabi" fn`
  - `&str` `&[u8]` `etc.`
- Standard data layout that supports enums (with data), etc. `#[repr(crabi)]`
  - `enum` `struct`
- Stable layout guarantees of common standard library types `#[repr(crabi)] in std`
  - `Option` `Result` `etc.`

crABI
github.com/joshtriplett/rfcs/blob/text/3470-crabi.md

Rust ABI

The Future: mechanism for exporting/importing, naming symbols and working with dynamic libraries

- Exporting items under stable linking symbols, supporting crates, modules, methods `#[export]`
- Use a crate as dynamic library, only importing the exported items `extern dyn crate`
- Cargo features for dynamically linking to Rust libraries `cargo dynamic deps`

Rust ABI

The Future: trait objects/vtables and typeid

- A standard data layout for dynamic trait objects (v-tables)
  - `&dyn T` `&mut dyn T` `Box<dyn T>`
- A way of dealing with types that depend on global state (eg. allocated objects)
  - `Box` `Vec`
- Stable typeid
  - `Any` `catch_unwind`
- Access to std structures like maps through dynamic std trait objects
  - `&dyn HashMap` `etc.`

Rust ABI

The Future: "*Don't stop me now!*" 🎶

- Turning parts of std into an opt-in dynamic library with a stable ABI (std as dylib)
- Tools to help with detect/maintaining ABI compatibility and tools to debug ABI issues
- Store signatures, data layouts in binaries (introspection)

ABI Cafe 🧩 ☕

faultlore.com/abi-cafe/book/

Pair Your Compilers At The ABI Café:
faultlore.com/blah/abi-puns/

The Shiny Future

If I were to guess, I would say the 🔮 future of Rust stable ABI

- is not "One To Rule Them All" 

but

- MANY (for better or for worse...)

Rust

~~C++~~

~~Swift~~ ABI Resilience?

Rust Nation UK

February 2025

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
Rust Tooling @ Microsoft