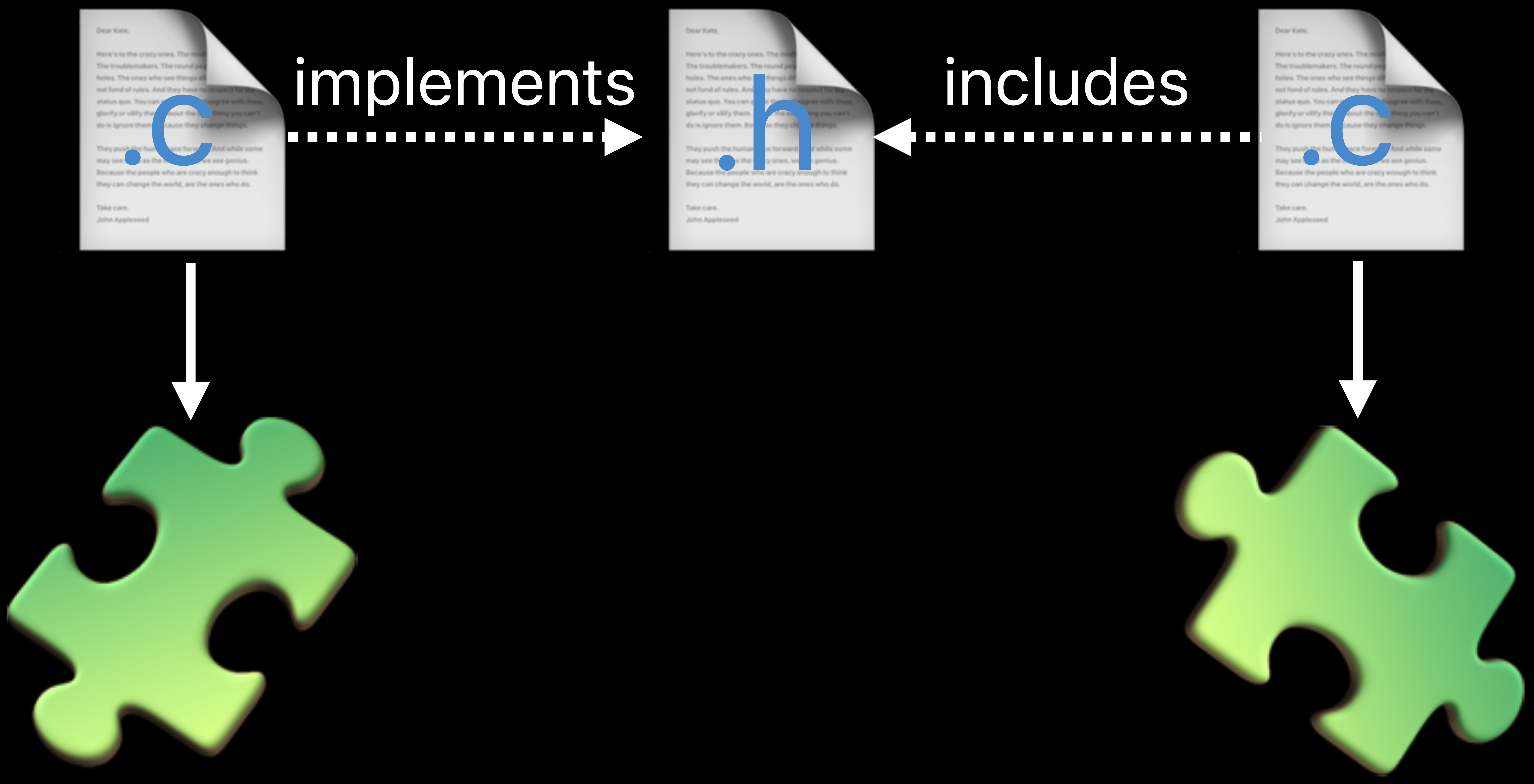
Implementing Language Support for ABI-Stable Software Evolution in Swift and LLVM

Doug Gregor

2022 LLVM Developers' Meeting | Apple, Inc. | November 8, 2022

What is an Application Binary Interface (ABI)?

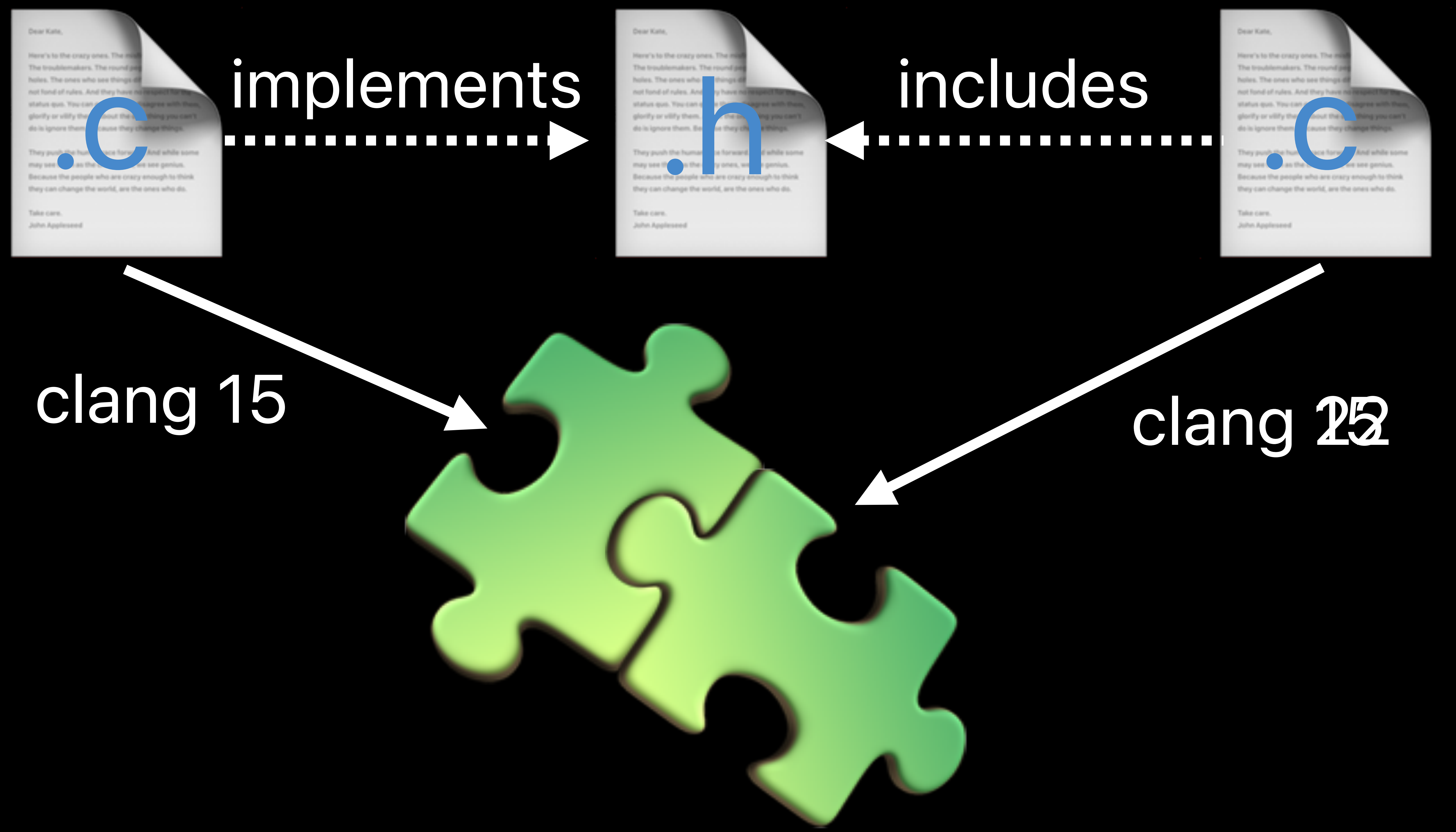
Binary compatibility between separately-compiled artifacts



Optional footnote

What is an Application Binary Interface (ABI)?

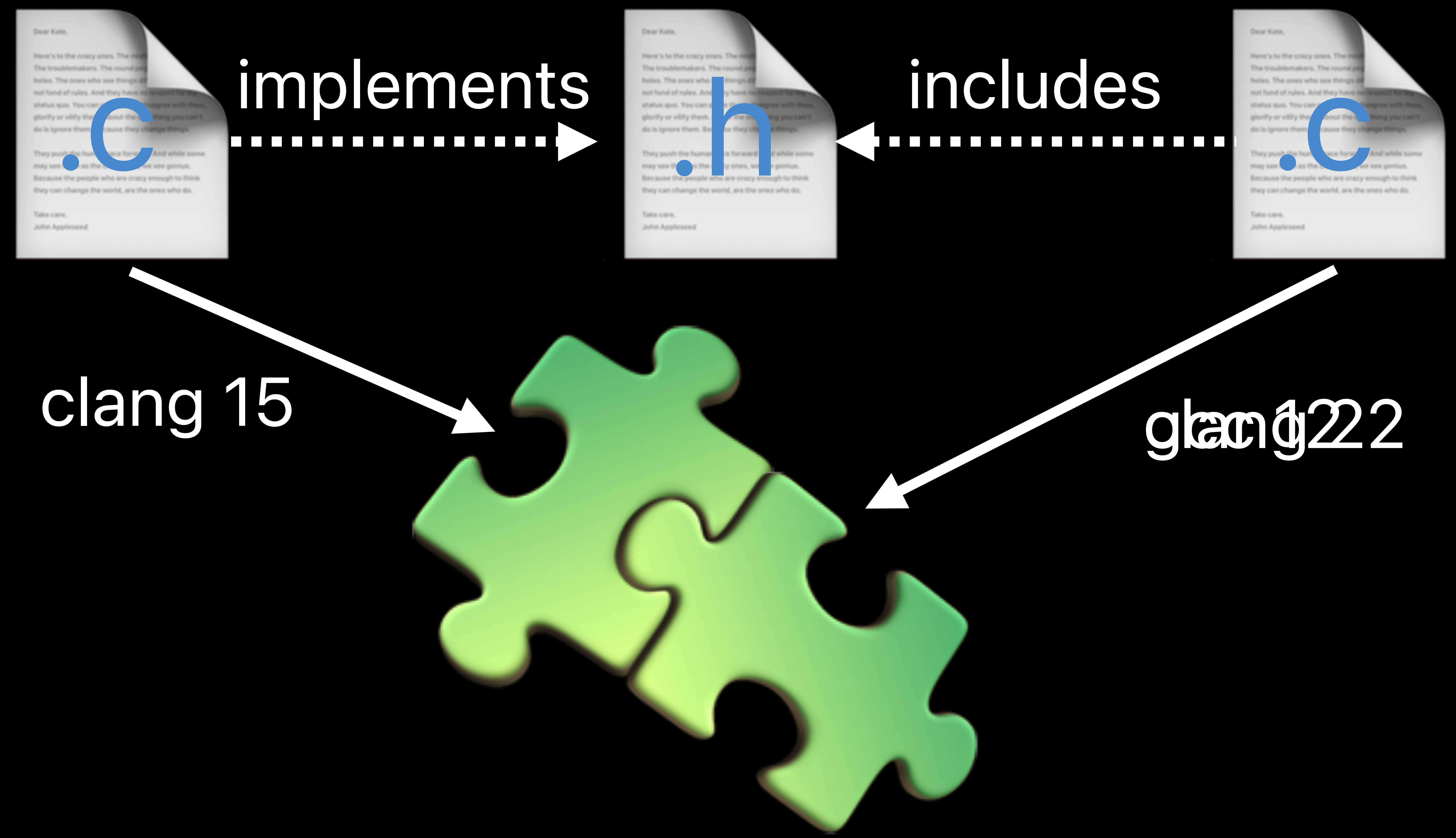
Binary compatibility between separately-compiled artifacts



Optional footnote

ABI Stability

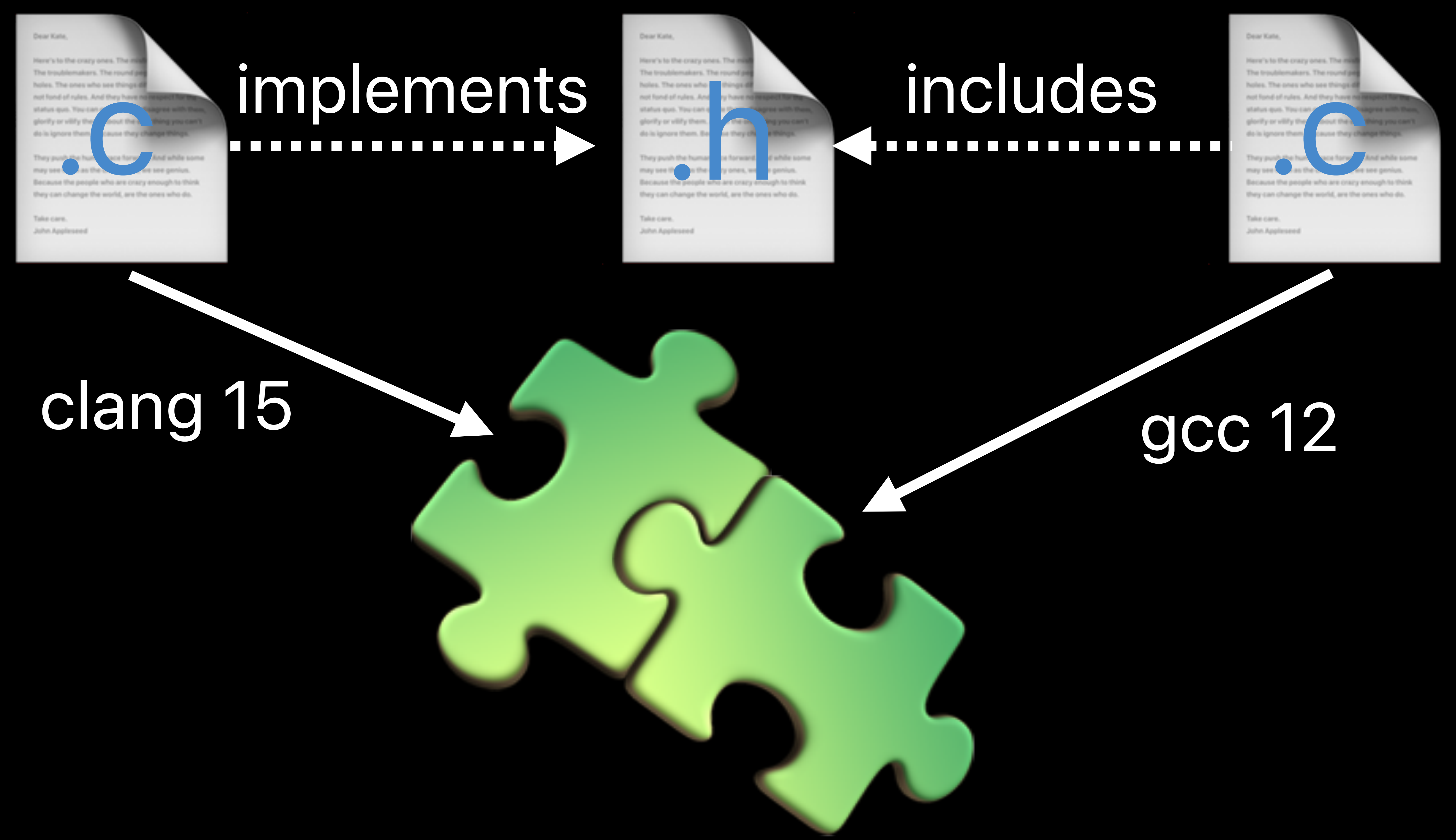
Binary compatibility across compiler versions



Optional footnote

ABI Standardization

Binary compatibility across different compilers



Optional footnote

Developer benefits of ABI stability / standardization

You don’t have to share the source code to your library

You can use the best compiler for your library

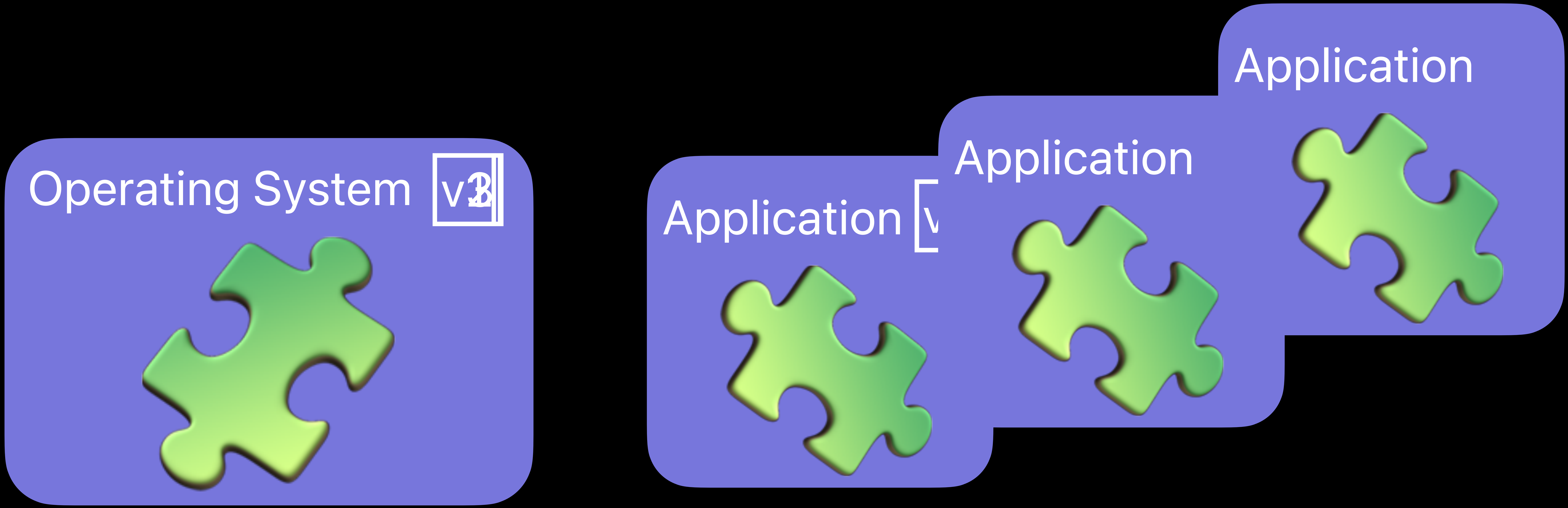
You don’t have to recompile the world

Optional footnote

Systemic benefits of ABI stability

Binary artifacts can be shipped and updated independently

Multiple programs can use the same shared library



Optional footnote

May I Have A Stable ABI, Please?

Go internal ABI specification

This document describes Go's internal application binary interface (ABI), known as ABIInternal. 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.

- Non-goals
  - Stable language and library ABI

NOPE

Zig natively supports C ABIs for `extern` things; which C ABI is used is dependant on the target which 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 make use of an ABI, meaning code should explicitly conform to a C ABI where reproducible and defined binary-level behaviour is needed.

Define a Rust ABI #600

Closed **steveklabnik** opened this issue on Jan 20, 2015 · 86 comments

Why Can’t I Have A Stable ABI?

What Goes Into An ABI?

Calling convention

Layout of types

- Size and alignment
- Offsets and types of every field
- Virtual table entries

Mangled names

Metadata

Optional footnote

Foreclosing On Future Compiler Optimizations

Stabilizing the ABI “too early” might miss optimizations

- Could implement a faster custom calling convention!
- Could implement optimal structure layout!
- Could change the way dynamic casting works!

These are solvable engineering problems

Optional footnote

Language ABI Stability Is An Engineering Problem

Language ABI Stability Is
Only Half Of the Solution

Evolution of Software Libraries

Developers want to evolve their software libraries without breaking ABI

- Add new functionality
- Fix bugs
- Improve performance

Most of these things break ABI!

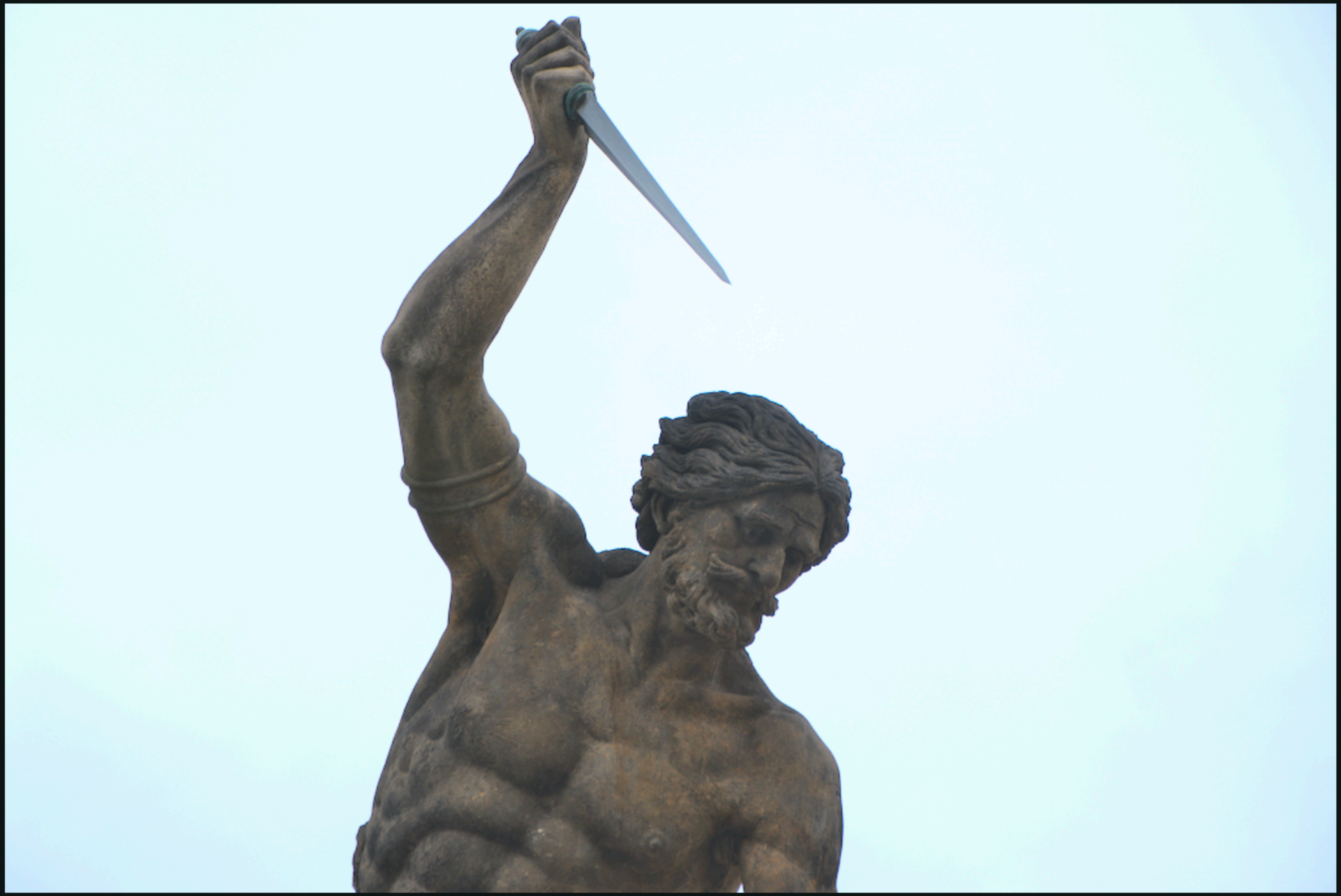
- Add a private field to a class?
- Add a new virtual function?
- Use some existing padding for that new field?

Optional footnote

C++ and ABI Stability

February 24, 2020

The Day The Standard Library Died



“All problems in computer science can be solved by another level of indirection”

— Attributed to David Wheeler

C++: The pImpl Idiom

```
// widget.h
class widget {
  struct impl;
  std::unique_ptr<impl> pImpl;
  // …
}

// widget.cpp
struct widget::impl {
  // implementation details
}
```

- ✅ Stable public type layout
- ✅ Can fix bugs
- ✅ Can add functionality
- ❌ Maintenance burden
- ❌ Not all features work
- ❌ Not the default
- ❌ Performance

Optional footnote

Designing a Language 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

Optional footnote

Evolving A Simple Struct

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

Evolving A Simple Struct

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

Evolving A Simple Struct

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

Evolving A Simple Struct

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

Evolving A Simple Struct

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

Optional footnote

Challenges For Compiling Client Code

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

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

Person struct changes size when new fields are added

Offset of fields changes whenever layout changes

Optional footnote

Optimize Data Layout, Indirect In The Code

Type Layout Should Be As-If You Had The Whole Program

Person library should layout the type without indirection

Expose metadata with layout information:

- Size/alignment of type
- Offsets of each of the public fields

Person
Offset 0: name Offset 8: birthDate Offset 24: id

Imagine the metadata in C:

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

Optional footnote

Client Code Indirects Through Layout Metadata

How to access a field?

- Read the metadata for the field offset (e.g., `Person_birthDate_offset`)
- Add that offset to the base object
- Cast the new pointer and load the field

How do I store an instance on the stack?

- Read the metadata for instance size (e.g., `Person_size`, `Person_align`)
- Emit an `alloca` instruction

Library Code Eliminates All Indirection

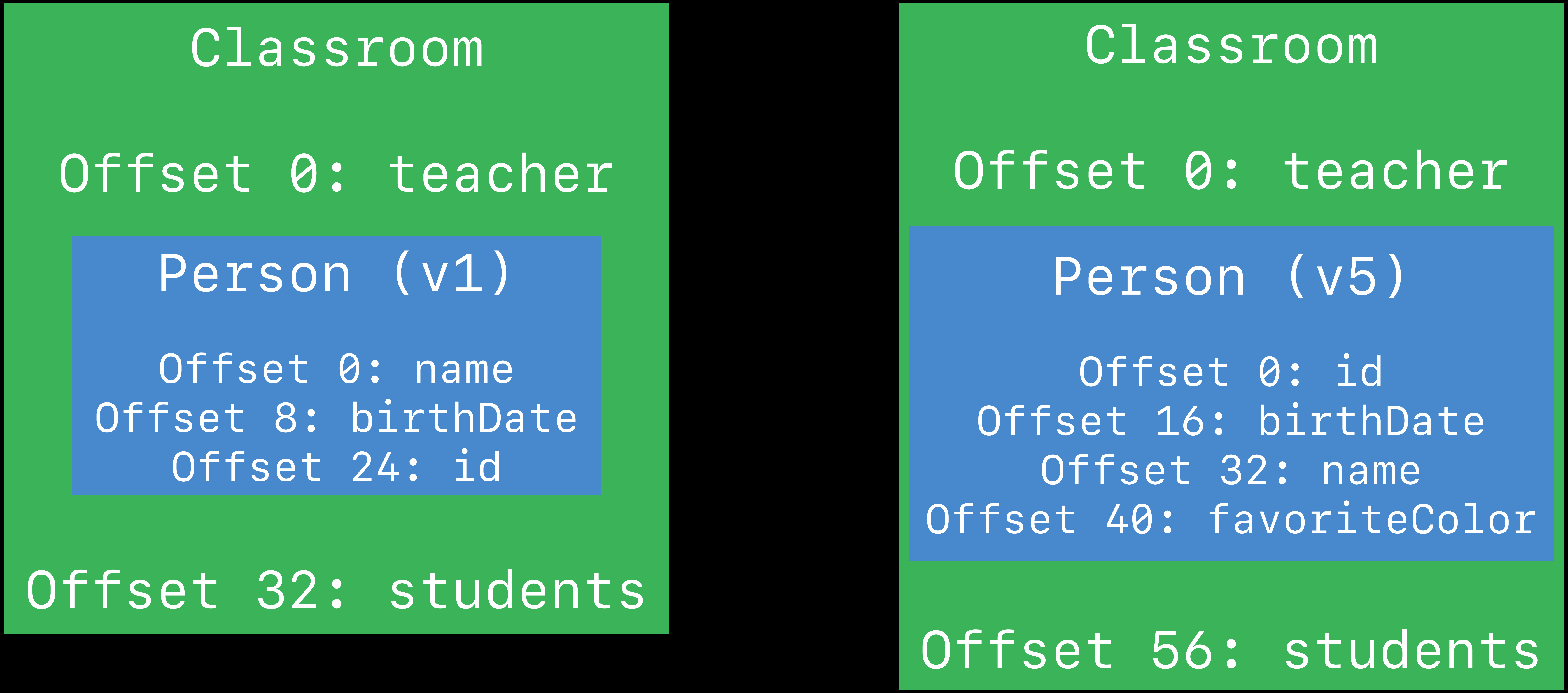
How to access a field?

- ~~Read the metadata for the field offset (e.g., `Person_birthDate_offset`)~~
- Add that offset to the base object
- Cast the new pointer and load the field

How do I store an instance on the stack?

- ~~Read the metadata for instance size (e.g., `Person_size`, `Person_align`)~~
- Emit an `alloca` instruction

Type Layout Can Occur After Compilation



Optional footnote

Generics Make Everything More Complicated

```
public struct Pair<First, Second> {
  public var first: First
  public var second: Second
}
```

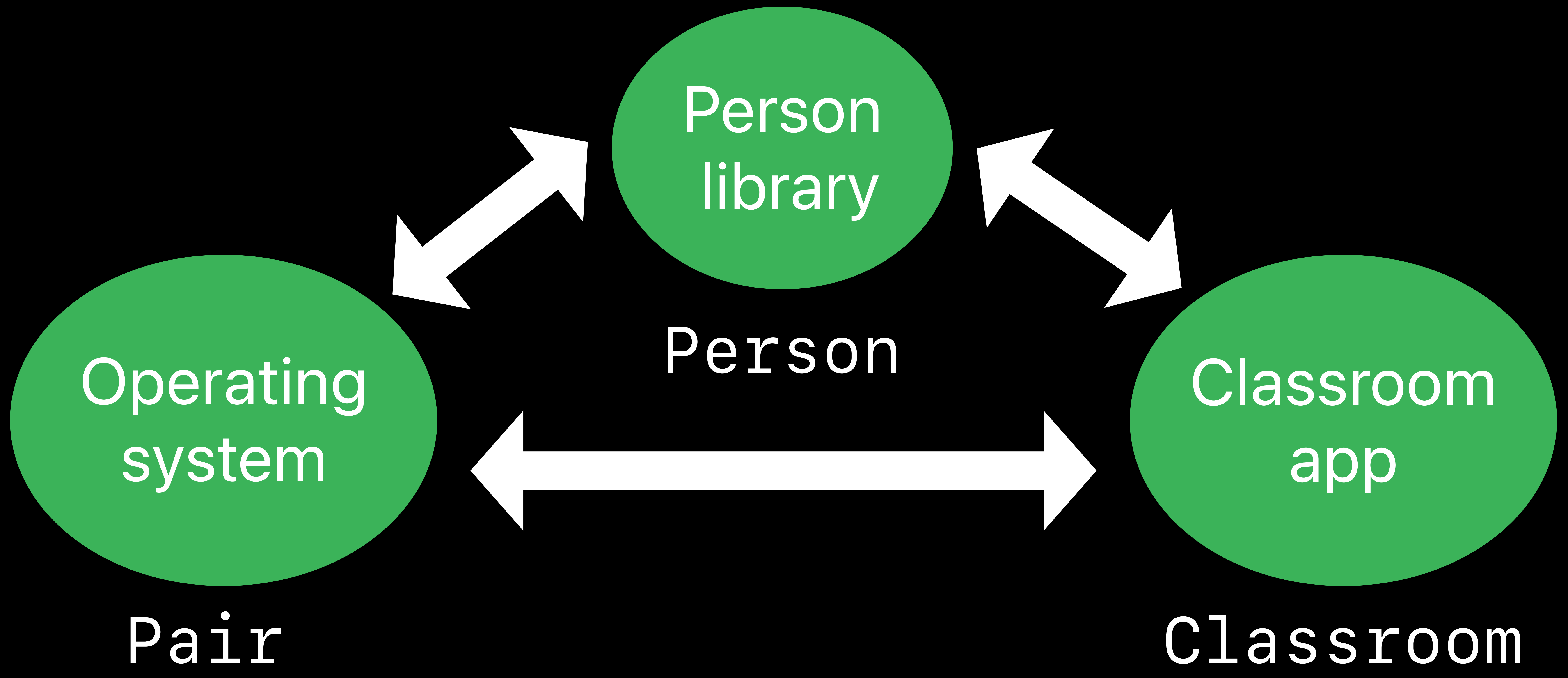
When can we know the layout of `Pair<Classroom, Person>`?

All generic implementations need to employ indirection

Resilience Domains

A resilience domain contains code that will be compiled together

A program can be composed of many different resilience domains



Optional footnote

Optimization and Resilience Domains

Across resilience domains, maintain stable ABI

Within a resilience domain, all implementation details are fair game

Optimizations need to be aware of resilience domain boundaries

Trading Future Evolution For Client Performance

Inline code is exposed to the client

```
extension Pair {
  @inline public func swapped() -> Pair<Second, First> {
    return .init(first: second, second: first)
  }
}
```

Enables caller optimization, generic specialization

Prevents any changes to the function’s semantics

Trading Future Evolution For Client Performance

Fixed-layout types promise never to change layout

```
@fixedLayout
public struct Pair<First, Second> {
  public var first: First
  public var second: Second
}
```

Enables layout of types in client code

Gives up ability to add/remove/reorder stored fields

Optional footnote

Challenges & Downsides

Large runtime component

- Runtime type layout
- Generics are particularly hard

Every language feature is harder

Older runtimes don’t support new language features

Optional footnote

What If There Is Only One Resilience Domain?

What If There Is Only One 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

This is how Swift scales *down*

Optional footnote

ABI Stability with Library Evolution

ABI stability enables systems to scale up

Library evolution provides flexibility to continually improve

Resilience domains control where the costs of ABI stability are paid

www.swift.org

Optional footnote