Duck-Tape Chronicles

Rust/C++ Interop

VICTOR CIURA

2025
September 13 - 19

A full week of 8am-10pm sessions

A full week of 8am-10pm sessions

⚠️ Disclaimer

No LLMs were hurt in the making of this presentation

This presentation was prepared by a *human* agent.
No hallucinations. But errors and 🔥 hot-takes are allowed.

Why do you care?
Why are you here?

Why do you care?
Why are you here?

When Rust folks are looking into C/C++ interop, it's natural…
they NEED it in order to call into existing libs they don't yet have.

Why do you care?
Why are you here?

When Rust folks are looking into C/C++ interop, it's natural…
they NEED it in order to call into existing libs they don't yet have.

But when C++ folks look into Rust interop, it's more than curiosity...
you know some degree of desperation has occurred 🔥

Rust code everywhere is increasing at an accelerated rate...

Rust ↔️ C++

Rust code everywhere is increasing at an accelerated rate...

but so does C++ (that's on top of gazillion lines already out there)

Rust ↔️ C++

Rust code everywhere is increasing at an accelerated rate...

but so does C++ (that's on top of gazillion lines already out there)

Hybrid codebases are quickly becoming the norm (whether we like it or not)

Rust ↔️ C++

Rust code everywhere is increasing at an accelerated rate...

but so does C++ (that's on top of gazillion lines already out there)

Hybrid codebases are quickly becoming the norm (whether we like it or not)

Rust ↔️ C++

They need to play nice together... for a looong time!

Who thinks interop is about... C FFI

Who thinks interop is about...

C FFI

glue code

Who thinks interop is about...

C FFI

glue code

coge generators

Who thinks interop is about...

C FFI

glue code

coge generators

(fat) compilers

Who thinks interop is about...

C FFI

glue code

coge generators

(fat) compilers

linkers

Who thinks interop is about...

C FFI

glue code

coge generators

(fat) compilers

linkers

ABI compat

What you’re going to get out of this talk

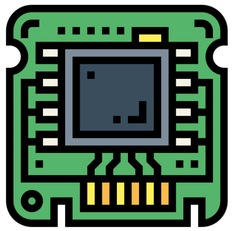
- This presentation aims to highlight:
  - some of the major interop challenges
  - existing solutions out there
  - tease out the avenues at the forefront of this pursuit

What you're going to get out of this talk

- This presentation aims to highlight:
  - some of the major interop challenges
  - existing solutions out there
  - tease out the avenues at the forefront of this pursuit

- General high-fidelity interoperability has yet to be achieved 🌶️
- Just "*making things work*" is not enough in the domain space of C++ and Rust
- Many of the explored solutions so far fail to deliver on all needed requirements

Rust extreme range of operation




Rust / C++ interoperability

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations

Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety

Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI

Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI
- Ergonomics - with safety

Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI
- Ergonomics - with safety
- Works with dynamic libraries (including the weirdness* of Windows DLLs, CRT)

Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI
- Ergonomics - with safety
- Works with dynamic libraries (including the weirdness* of Windows DLLs, CRT)
- Plays well with C++ ABI

Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI
- Ergonomics - with safety
- Works with dynamic libraries (including the weirdness* of Windows DLLs, CRT)
- Plays well with C++ ABI
- Easily automated

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI
- Ergonomics - with safety
- Works with dynamic libraries (including the weirdness* of Windows DLLs, CRT)
- Plays well with C++ ABI
- Easily automated
- Debuggable

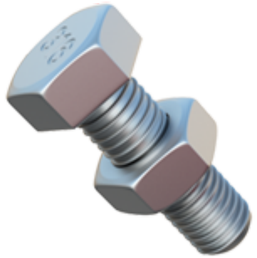
Choose... ~~none~~ some?

Rust / C++ interoperability

- No perf overhead (avoid marshaling costs, eg. copying strings)
- No boilerplate or re-declarations / No C++ annotations
- Broad types support - with safety
- Avoid lowering through C FFI
- Ergonomics - with safety
- Works with dynamic libraries (including the weirdness* of Windows DLLs, CRT)
- Plays well with C++ ABI
- Easily automated
- Debuggable
- Hybrid build systems (CMake, cargo, MSBuild, bazel, buck...)

Choose... ~~none~~ some?

Compiler



Interop Library

Debugger

Linker

ABI guarantees

Packaging

Build systems & CI

CppCon

September 2025

Duck-Tape Chronicles
Rust/C++ Interop

Episode $1^{3/4}$

@ciura_victor

@ciura_victor@hachyderm.io

@ciuravictor.bsky.social

Victor Ciura

Principal ~~Engineer~~

Rambling Idiot

Rust Tooling @ Microsoft

About me

Advanced Installer

Clang Power Tools

Oxidizer SDK

Visual C++

Rust Tooling
Microsoft

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

Disclaimer

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

What's out there...

C - The Original Duck Tape

🙂

- **C** is the lingua franca FFI systems language
- Every API consumable from most languages
- The only ABI-stable "universal interop glue"

☹️

- Poor abstraction
- No safety
- Naked structs (public fields)
- Raw pointers
- Manual lifetimes

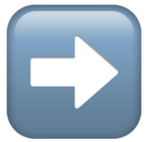
bindgen

Allows Rust to call into C APIs

C headers ➡️ Rust FFI bindings

```
typedef struct Widget {
...
} Widget;

void action(Widget * w);
```



```
#[repr(C)]
pub struct Widget {
...
}

extern "C" {
  pub fn action(w: *mut Widget);
}
```

Source generation (build step)

cbindgen

Allows C code to call Rust APIs

.rs ➡️ C headers

```
#[repr(C)]
pub struct Widget {
...
}

#[unsafe(no_mangle)]
pub extern "C" fn action(w: *mut Widget) {
...
}
```

```
typedef struct Widget {
...
} Widget;

void action(Widget * w);
```

Source generation (build step)

bindgen / cbindgen

- Works directly on source files (not IDL)
- Source generation (build step)
- Types: repr(C) ABI only
- Pass by value: for C types
- ~~Structs with private fields~~
- ~~C++ classes~~
- ~~std::unique_ptr, std::optional~~
- ~~Box<T>, Option<T>~~
- ~~Rust enums~~
- ~~&str, String~~
- ~~std::string~~
- ~~&[T]~~

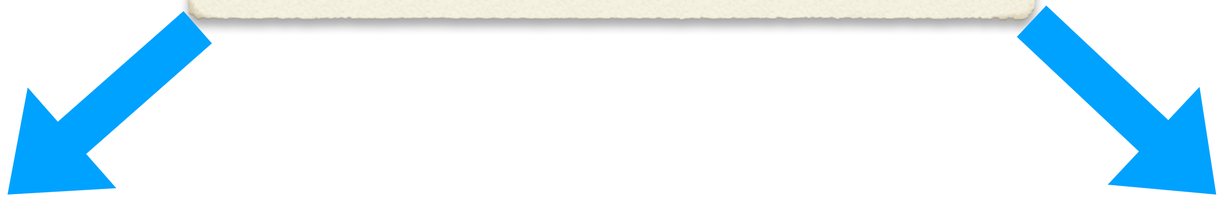
unsafe{} required to convert to/from C representation

CXX

Macro-based **IDL**

Needs to be separately maintained (manually)

```
#[cxx::bridge]
mod ffi {
  struct Widget {
    things: Vec<String>
  }
}
```



```
#[repr(C)]
struct Widget {
  things: Vec<String>
}
```

```
struct Widget {
  rust::Vec<rust::String> things;
};
```

cxx

- Types: standard types (mostly), slices, IDL structs
- C++ classes
- std::unique_ptr, std::optional
- Box<T>, Option<T>
- &str, String
- std::string
- std::vector
- Vec<T>
- &[T]

  - cxx does't know the memory layout of user types
  - ❌ Pass-by-value => need to Box<T> or unique_ptr<T>
  - relies heavily on pinning (reduced ergonomics)

zngur

```rust
struct Widget {
  id: u32,
  things: Vec<String>
}

impl Widget {
  fn new_empty(id: u32) -> Self {
    Self {
      id: id,
      things: vec![],
  }

  fn work() -> f32 {
   ...
  }
}
```

Custom **IDL** (.zng)

```
type crate::Widget {
  #layout(size = 32, align = 8)

  fn new_empty(u32) -> crate::Widget;
  fn work() -> f32;
}
```

```cpp
#include "generated.h"

void cpp_caller() {
  auto w = rust::crate::Widget::new_empty(42);
  w.work();
}
```

zngur

- Custom **IDL** (.zng)
  - Needs to be separately maintained (manually)

- Types: standard types (mostly), slices, IDL structs

✅ Pass-by-value: have to manually annotate types with: #[layout(size, align)]

- no need for indirection/boxing and heap allocation

- Reduced need for pinning

- Favors Rust-friendly APIs and developer experience,
  accepting *occasional* runtime cost to get there

crubit

crubit

- Bold new project with the goal of high-fidelity lang interop between Rust and C++

crubit

- Bold new project with the goal of high-fidelity lang interop between Rust and C++
- Needs native compiler integration (**Clang** + rustc)

crubit

- Bold new project with the goal of high-fidelity lang interop between Rust and C++
- Needs native compiler integration (**Clang** + rustc)
- Works directly on source files (no IDL needed)

crubit

- Bold new project with the goal of high-fidelity lang interop between Rust and C++
- Needs native compiler integration (**Clang** + rustc)
- Works directly on source files (no IDL needed)
- Covers the whole API surface (IDL-based solutions can be targeted)

crubit

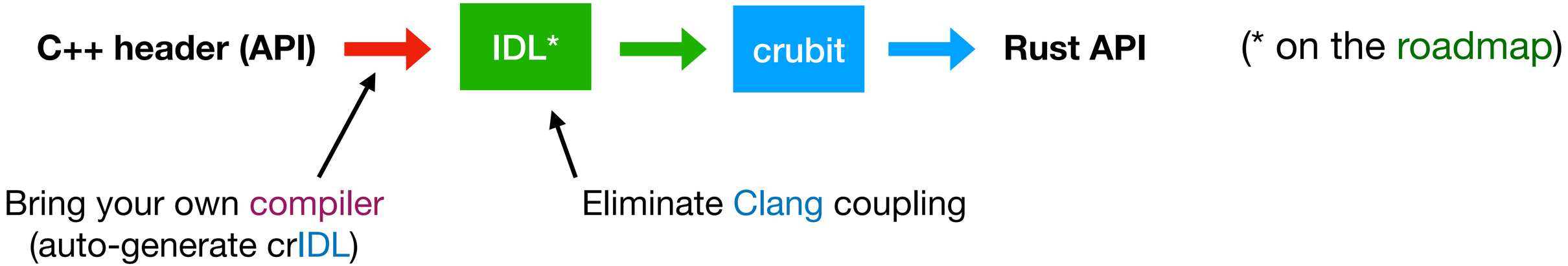
- Bold new project with the goal of high-fidelity lang interop between Rust and C++
- Needs native compiler integration (**Clang** + rustc)
- Works directly on source files (no IDL needed)
- Covers the whole API surface (IDL-based solutions can be targeted)
- C++ compiler diversity: ~~MSVC, GCC~~, Clang
  - Optional IDL (TBD - on the roadmap)

crubit

- Bold new project with the goal of high-fidelity lang interop between Rust and C++
- Needs native compiler integration (**Clang** + rustc)
- Works directly on source files (no IDL needed)
- Covers the whole API surface (IDL-based solutions can be targeted)
- C++ compiler diversity: ~~MSVC, GCC~~, Clang
  - Optional IDL (TBD - on the roadmap)
- Pass by value: AllTheThings™ (that's where deep compiler integration comes in)

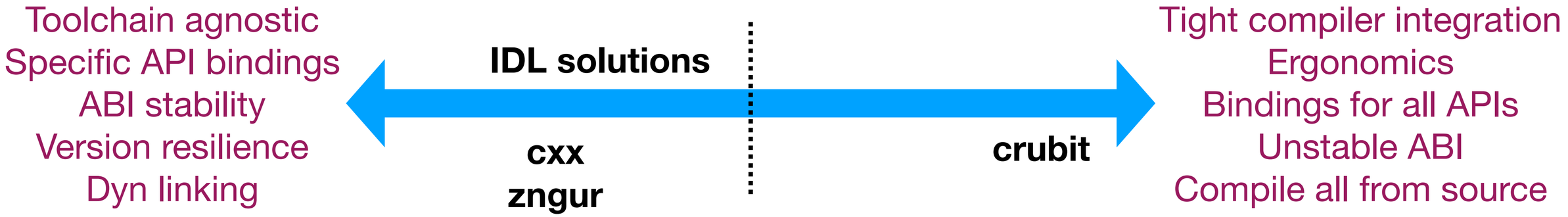
crubit


C++ header (API)
crubit
Rust API


C++ header (API)
IDL*
crubit
Rust API
(* on the roadmap)
Bring your own compiler
(auto-generate crIDL)
Eliminate Clang coupling

Tradeoffs...

Projects have very diverse interop needs, so no solution fits all (equally)



Language Semantics

Some C++ features not having direct Rust equivalents:

- Overloaded assignment operator
- Overloaded dereference operator
- Overloaded new and delete operators
- Function overloading
- Argument-dependent lookup
- Default function parameters
- Implicit conversions
- SFINAE
- In-place initialization
- Move constructors

Language Semantics

Profound semantic differences between language constructs

- Rust semantics is a subset of C++ semantics
- Generally, Rust is less expressive than C++

=>

- Using Rust code from C++ is easier
- Using C++ code from Rust much harder

Calling C++ from Rust

Level: HARD!!!

- C++ features not having direct Rust equivalents (eg. overloading)
- unsafe
- Lifetimes
- Aliasing (refs)
- Movable types that are non memcopy
- ...

Calling Rust from C++

Level: I CAN DO IT

- Rust semantics is a subset of C++ semantics
- Rust's strong type system
  - easy to grasp intended semantics of functions, types

- Querying rustc - ⚠️ Rust ABI is not stable: these need to be refreshed on each update
  - determine the exact size & alignment of every Rust type
  - struct fields
  - key trait implementations:
    - Drop ➡️ C++ dtor
    - Clone ➡️ C++ copy ctor

Function Overloading

Function Overloading

Ability for Rust to call overloaded C++ functions 😥 (this is a real need!)

Function Overloading

Ability for Rust to call overloaded C++ functions 😥 (this is a real need!)

Function Overloading

Ability for Rust to call overloaded C++ functions 😥 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level
  - 🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

(see Carbon)

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

  (see Carbon)

- We can probably solve this outside the core language

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

  (see Carbon)

- We can probably solve this outside the core language
- No need to hinder Rust powerful type inference with overloading

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

  (see Carbon)

- We can probably solve this outside the core language
- No need to hinder Rust powerful type inference with overloading

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

  (see Carbon)

- We can probably solve this outside the core language
- No need to hinder Rust powerful type inference with overloading

- At the ABI level overloading effectively doesn't exist

  - it's just differently mangled symbol names

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

  (see Carbon)

- We can probably solve this outside the core language
- No need to hinder Rust powerful type inference with overloading

- At the ABI level overloading effectively doesn't exist

  - it's just differently mangled symbol names
- No fundamental need for a Rust to allow function overloads in the core language

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (this is a real need!)

- Some folks say we really need to have a way to semantically identify a C++ overload from Rust - at language level

  🌶️🌶️ Let's resist temptation to complicate Rust for the sake of interop

  (see Carbon)

- We can probably solve this outside the core language
- No need to hinder Rust powerful type inference with overloading

- At the ABI level overloading effectively doesn't exist
  - it's just differently mangled symbol names
- No fundamental need for a Rust to allow function overloads in the core language
- Need a way to name-mangle such that separate functions map to the correct overloads

The ABI Menace

What is ABI, anyway?

ABI isn't a property of a programming language

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

ABI is something defined by the *platform*

Eg.
Compilers determine class layout: ❌ portable

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

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
  - make changes affecting v-table
  - (re)use existing padding

ABI Stability - Why?

ABI Stability - Why?

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

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

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
- Use libraries compiled with a different compiler version

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
- Use libraries compiled with a different compiler version
- You don’t have to recompile everything (full project visibility)

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
- Use libraries compiled with a different compiler version
- You don’t have to recompile everything (full project visibility)
- Binaries can be shipped and updated independently (patches)

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
- Use libraries compiled with a different compiler version
- 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)

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
- Use libraries compiled with a different compiler version
- 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 (dynamically loaded)

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
- Use libraries compiled with a different compiler version
- 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 (dynamically loaded)
- Language interop (hybrid projects)

The (early) 90s are calling...

- Old-school interop: COM, CORBA, XPCOM, ...

- COM
  - MIDL for interop
  - metadata
  - ABI resilience

Design 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

Struct Layout

C++ compilers could provide a class' data members with layout metadata

=> allow representation of Rust struct fields in C++

Retrieve layout via the C++ AST and the rustc query API

Layout

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

- *Widget* library should layout the type without indirection
- Expose metadata with layout information:
  - size/alignment of type
  - offsets of each of the public fields
  - overlapping sub-objects
  - padding tricks & vtables
- Attributes, annotations, or compiler synthesized

```
size_t Widget_size = 32;
size_t Widget_align = 8;
size_t Widget_field1_offset = 0;
size_t Widget_field2_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~~

- performance: indirects only when necessary

- 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

Dynamically-sized

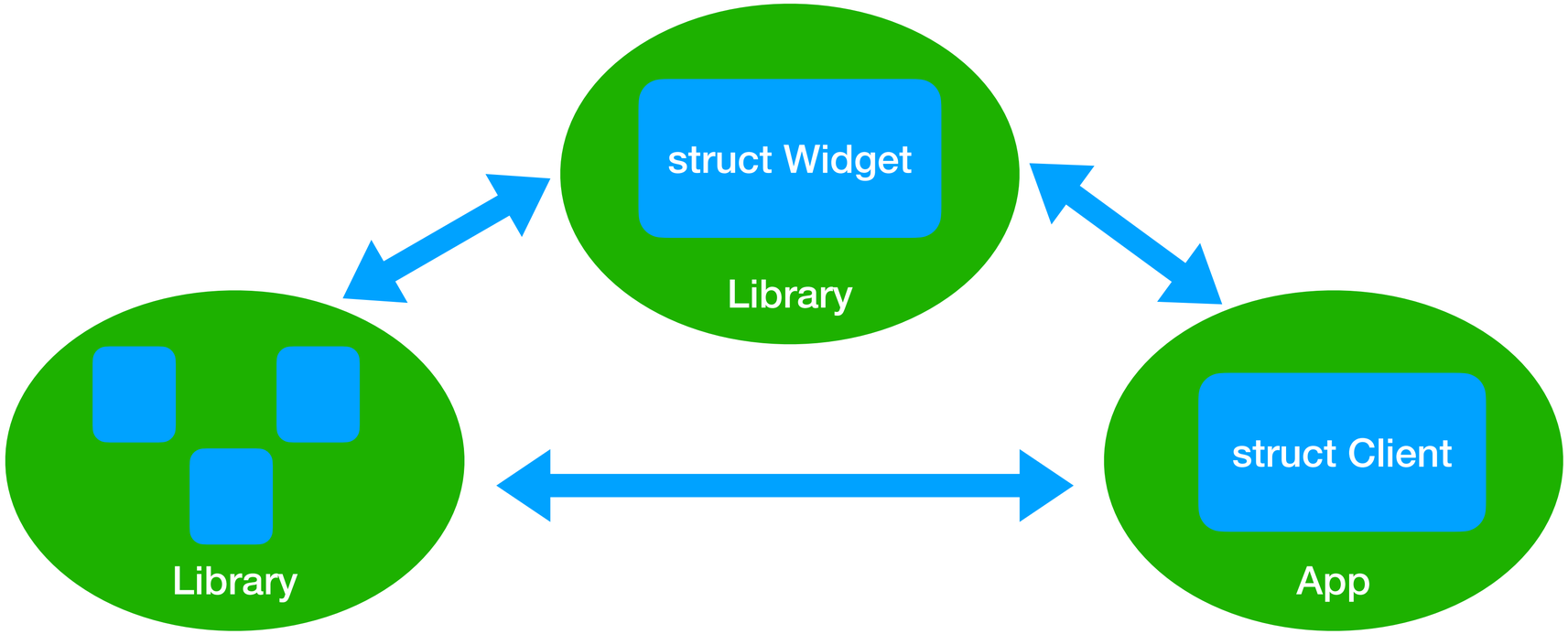
- Support for dynamically-sized things on the stack is key (eg. LLVM)
- Compilers can use of this for of ABI-stable value types:
  - you have local variable of some struct defined in an ABI-stable library
  - so you don’t know it’s size until load time
- Dynamic allocs can handle this nicely (with minimal perf impact)
- C++ desperately wants all objects to have compile-time-constant size
  - the notion of sizeof/alignof being runtime values clashes with the C++ model

Interop Domains

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

- introduce appropriate indirections across separately-evolved software modules
- while optimizing away that indirection within software modules that are always compiled together

Interop Domains



An interop domain contains code that will always be compiled together

Domains can control where the costs of interop are paid

Interop Domains

Optimization vs. Resilience

- 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
- A program can have just 1 resilience domain

Tail Padding & Rust ABI

Tail Padding & Rust ABI

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not
- Rust treats tail padding as part of the value
  - users expect to be able to memcpy() of size_of::<T>() bytes

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

- Rust treats tail padding as part of the value
  - users expect to be able to memcpy() of size_of::<T>() bytes
- C++ does not allow this
  - fields of a child class may be placed in tail padding of the base class

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

- Rust treats tail padding as part of the value
  - users expect to be able to memcpy() of size_of::<T>() bytes
- C++ does not allow this
  - fields of a child class may be placed in tail padding of the base class

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

- Rust treats tail padding as part of the value
  - users expect to be able to memcpy() of size_of::<T>() bytes
- C++ does not allow this
  - fields of a child class may be placed in tail padding of the base class

- A field with [[no_unique_address]] may have its tail padding reused for a neighbor field

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

- Rust treats tail padding as part of the value
  - users expect to be able to memcpy() of size_of::<T>() bytes
- C++ does not allow this
  - fields of a child class may be placed in tail padding of the base class

- A field with [[no_unique_address]] may have its tail padding reused for a neighbor field

Tail Padding & Rust ABI

- C++ is allowed to reuse tail padding of structs, but Rust does not

- Rust treats tail padding as part of the value
  - users expect to be able to memcpy() of size_of::<T>() bytes
- C++ does not allow this
  - fields of a child class may be placed in tail padding of the base class

- A field with [[no_unique_address]] may have its tail padding reused for a neighbor field

- Prevents Rust from turning a C++ child reference into a base class reference
  - doing so would allow overwriting the tail padding (and thereby the child fields)

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)]`

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/

I like to move it, move it...

Object Relocation

One particularly sensitive topic about handling C++ **values**
is that they are all *conservatively* considered **non-relocatable**

Object Relocation

In contrast, a **relocatable value** would preserve its invariant,
even if its bits were moved arbitrarily in memory

For example, an `int32` is relocatable because moving its 4 bytes
would preserve its actual value, so the address of that value does not matter to its integrity

I like to move it, move it...

Object Relocation

C++'s assumption of non-relocatable **values** hurts everybody

for the benefit of a few questionable designs

Object Relocation

Only a *minority* of objects are genuinely non-relocatable:

Eg.
- objects that use internal pointers
- objects that need to update observers that store pointers to them

Trivially Relocatable

Trivially Relocatable

- Relocating an object to a distinct physical location is a destructive move
  - create new object having original value at destination
  - destroy the source object

Trivially Relocatable

- Relocating an object to a distinct physical location is a destructive move
  - create new object having original value at destination
  - destroy the source object

Trivially Relocatable

- Relocating an object to a distinct physical location is a destructive move
  - create new object having original value at destination
  - destroy the source object

- For a lot of types (eg. std container types): copying the bytes and discarding the source
  - anything with no self-references, eg. std::vector, std::unique_ptr, etc.

Trivially Relocatable

- Relocating an object to a distinct physical location is a destructive move
  - create new object having original value at destination
  - destroy the source object

- For a lot of types (eg. std container types): copying the bytes and discarding the source
  - anything with no self-references, eg. std::vector, std::unique_ptr, etc.

Trivially Relocatable

- Relocating an object to a distinct physical location is a destructive move
  - create new object having original value at destination
  - destroy the source object

- For a lot of types (eg. std container types): copying the bytes and discarding the source
  - anything with no self-references, eg. std::vector, std::unique_ptr, etc.

- Many libraries already optimize for such types

Trivially Relocatable

- Relocating an object to a distinct physical location is a destructive move
  - create new object having original value at destination
  - destroy the source object

- For a lot of types (eg. std container types): copying the bytes and discarding the source
  - anything with no self-references, eg. std::vector, std::unique_ptr, etc.

- Many libraries already optimize for such types
- Trivial relocation standardizes this important optimization

I like to move it, move it...

Trivial Relocatability C++26

Safely relocate objects in memory

wg21.link/P2786

I like to move it, move it...

Trivial Relocatability C++26

Safely relocate objects in memory

Many types in C++ cannot be trivially moved or destroyed, but do support trivially moving an object from one location to another by copying its bits — an operation known as trivial relocation

wg21.link/P2786

I like to move it, move it...

Trivial Relocatability C++26

Safely relocate objects in memory

Many types in C++ cannot be trivially moved or destroyed, but do support trivially moving an object from one location to another by copying its bits — an operation known as trivial relocation

wg21.link/P2786

I like to move it, move it...

Trivial Relocatability C++26

Safely relocate objects in memory

Many types in C++ cannot be trivially moved or destroyed, but do support trivially moving an object from one location to another by copying its bits — an operation known as trivial relocation

Some types even support bitwise swapping, which requires replacing the objects passed to the swap function, without violating any object invariants

wg21.link/P2786

I like to move it, move it...

Trivial Relocatability C++26

Safely relocate objects in memory

Many types in C++ cannot be trivially moved or destroyed, but do support trivially moving an object from one location to another by copying its bits — an operation known as trivial relocation

Some types even support bitwise swapping, which requires replacing the objects passed to the swap function, without violating any object invariants

wg21.link/P2786

I like to move it, move it...

Trivial Relocatability C++26

Safely relocate objects in memory

Many types in C++ cannot be trivially moved or destroyed, but do support trivially moving an object from one location to another by copying its bits — an operation known as trivial relocation

Some types even support bitwise swapping, which requires replacing the objects passed to the swap function, without violating any object invariants

Optimizing containers to take advantage of this property of a type is already in widespread use throughout the industry, but is undefined behavior as far as the language is concerned

wg21.link/P2786

Trivially Relocatable

A class is trivially relocatable if:

- it has no virtual base classes
- all of its sub-objects are trivially relocatable
- it has no *deleted* destructor
- AND:
  - its move constructor, move-assignment operator, and destructor are *defaulted*
  - OR
  - it's tagged with the `trivially_relocatable_if_eligible` keyword

I like to move it, move it...

C++ and Rust have opposite ways of handling move:

- Rust likes to move by default
- C++ likes to copy by default
- Rust does memcpy() on the bytes of T, regardless of type
- C++ is by default needing move functions (ctor, =)
  - eg. std::string cannot be memcopy-ed due to SSO (self referential *)
- Rust Pin solves the issue with self-referential types
  - not ergonomic (pollutes the context)

I like to move it, move it...

❌ place a C++ object on a Rust stack since it cannot be safely memcopy-moved (relocated)

C++26: Make C++ types trivially relocatable (annotate types)

Get standard library to be relocatable
=> allow most C++ types on the Rust stack (efficiency)

📖 Improving Rust/C++ Interop with Trivial Relocatability:
camio.github.io/trivially_relocate_rust/trivially_relocate_rust.pdf

I like to move it, move it...

Support for destructive moves in C++ would match the behavior of Rust drop mechanics

I like to move it, move it...

Support for destructive moves in C++ would match the behavior of Rust drop mechanics

- Rust move: which is a blind memcpy
  - render the moved-from object inaccessible
- C++ move: where a move is really like a mutating Clone operation
  - leave the moved-from value accessible to be destroyed at the end of the scope

I like to move it, move it...

Support for destructive moves in C++ would match the behavior of Rust drop mechanics

- Rust move: which is a blind memcpy
  - render the moved-from object inaccessible
- C++ move: where a move is really like a mutating Clone operation
  - leave the moved-from value accessible to be destroyed at the end of the scope

📦 moveit

- safe in-place construction of Rust and C++ objects
- mirrors Rust's drop semantics in its destructive moves
- moved-from values can no longer be used afterwards

Let's talk compilers!

Compilers & Interop

Many of the tricks here require deep compiler involvement:

- on C++ side (pick your poison 🙂)
- on Rust side (easy: 1 instance?)

Compilers & Interop

Many of the tricks here require deep compiler involvement:

- on C++ side (pick your poison 🙂)
- on Rust side (easy: 1 instance?)

High-fidelity language semantics & mapping of vocabulary types:

- front-ends (C++, rustc)
- toolchain independent IR
- support libs?

Compilers & Interop

Many of the tricks here require deep compiler involvement:

- on C++ side (pick your poison 🙂)
- on Rust side (easy: 1 instance?)

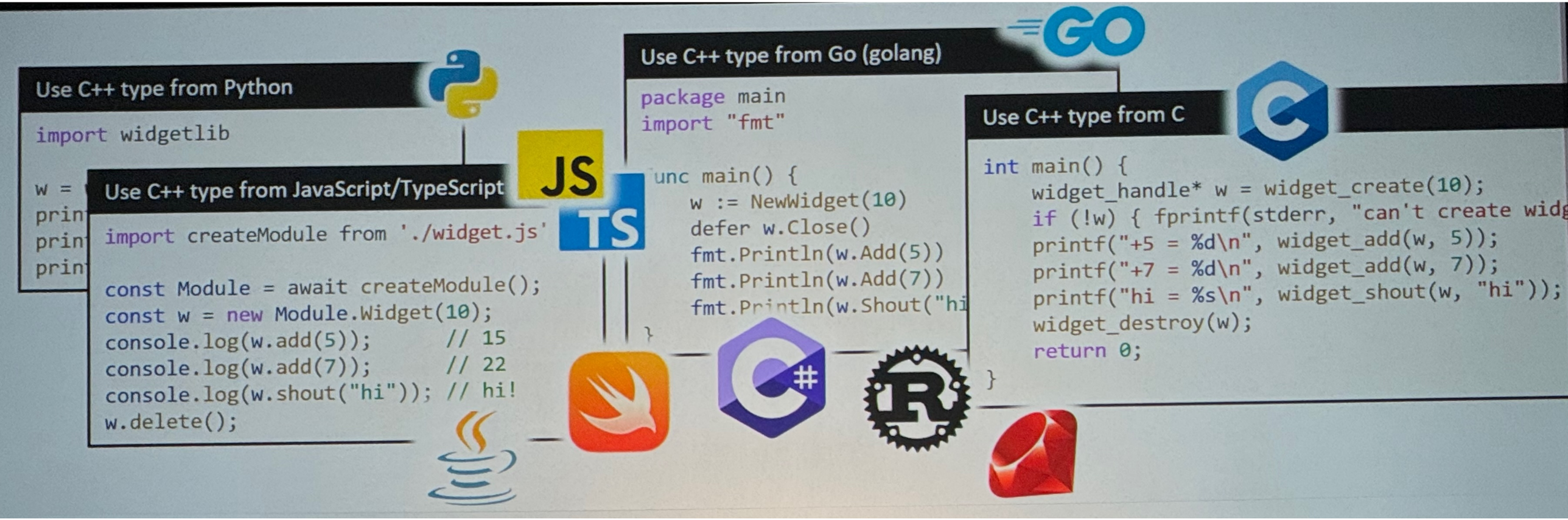
High-fidelity language semantics & mapping of vocabulary types:

- front-ends (C++, rustc)
- toolchain independent IR
- support libs?

Binary-level fidelity, ABI, linking, dylib, etc.

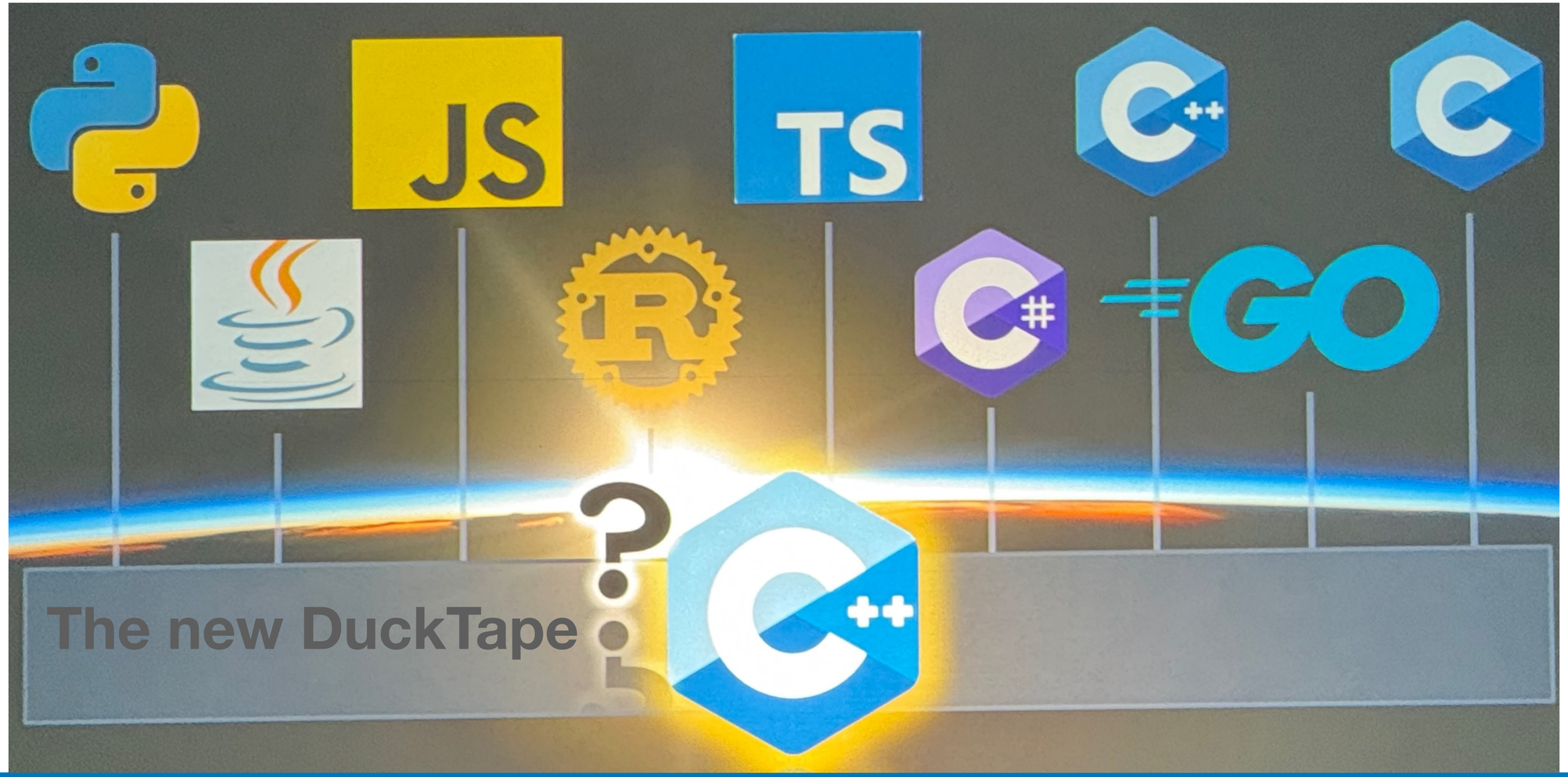
- platform integration
- post-build tooling
- codegen / back-end

C++26 Reflection will be a game changer!



Herb Sutter: "Reflection: C++'s Decade-Defining Rocket Engine" (CppCon 2025)

C++26 Reflection will be a game changer!



Who's driving this thing?

Active Effort

This year, there have been effervescent talks in the Rust Project & community about this topic (in the broader interop context, not just C++)

Active Effort

This year, there have been effervescent talks in the Rust Project & community about this topic (in the broader interop context, not just C++)

Short term: Rust25H1 Project Goals

- Contribute engineering time to some of the key interop crates
- Gain perspective on what sort of challenges need solutions external to those crates

Active Effort

This year, there have been effervescent talks in the Rust Project & community about this topic (in the broader interop context, not just C++)

Short term: Rust25H1 Project Goals

- Contribute engineering time to some of the key interop crates
- Gain perspective on what sort of challenges need solutions external to those crates

Medium term:

- Evaluate approaches for "seamless" interop between C++ and Rust
- Document the problem space of current interop challenges (identify the gaps)
- Facilitate top-down discussions about priorities and tradeoffs

Active Effort

This year, there have been effervescent talks in the Rust Project & community about this topic (in the broader interop context, not just C++)

Short term: Rust25H1 Project Goals

- Contribute engineering time to some of the key interop crates
- Gain perspective on what sort of challenges need solutions external to those crates

Medium term:

- Evaluate approaches for "seamless" interop between C++ and Rust
- Document the problem space of current interop challenges (identify the gaps)
- Facilitate top-down discussions about priorities and tradeoffs

Rust Foundation joined INCITS in order to participate in the C++ ISO standards process (Jon Bauman, David Sankel, et.al.)

Rust/C++ Interop Study Group

Interested? join the Rust Project Zulip server

- rust-lang.zulipchat.com
- #t-lang/interop channel

You'll find there some familiar Rust and C++ names 🙂

Rust/C++ Interop Study Group

Interested? join the Rust Project Zulip server

- rust-lang.zulipchat.com
- #t-lang/interop channel

You'll find there some familiar Rust and C++ names 🙂

🗒️ Meetings:

- Feb 26 First lang-team design meeting on the topic - Notes 📝
- Apr 23 Short-sync on interop interest in industry
- May 15-17 Interop study group @ Rust-All-Hands - Notes 📝
- Sep 2 Interop study group @ RustConf - Notes 📝

Must watch 

youtube.com/watch?v=k_sp5wvoEVM

RustConf
2025 | Seattle + Online

Fine-grained Rust / C++ interop

Taylor Cramer and Tyler Mandry

We are crubit

The original annual Rust programming language conference.

Learn more at rustconf.com

hosted by: Rust Foundation

Open Discussion

What does Rust/C++ interop mean for you?

What are the interop requirements/challenges of your project?

CppCon

September 2025

Duck-Tape Chronicles
Rust/C++ Interop

Episode 1[3/4]

Episode 2

Victor Ciura

Principal ~~Engineer~~
Rambling Idiot

Rust Tooling @ Microsoft

@ciura_victor

@ciura_victor@hachyderm.io

@ciuravictor.bsky.social