Pure Virtual C++

Microsoft

Mind The Gap: C++/Rust Interop

@ciura_victor

@ciura_victor@hachyderm.io

@ciuravictor.bsky.social

Victor Ciura

Rust Tooling @ Microsoft

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 a thing (whether we like it or not)

C++ ↔️ Rust

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

glue code

C FFI

unsafe

coge generators

(fat) compilers

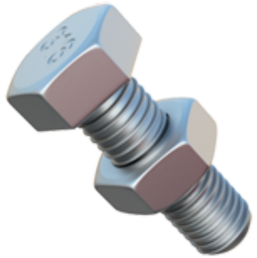
ergonomics

linkers

perf

ABI compat

Compiler



Interop Library

Debugger

Linker

ABI guarantees

Packaging

Build systems & CI

What you're going to get out of this talk

- I aim 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

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

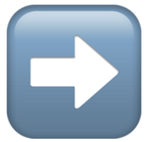
bindgen

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

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

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

```c
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]~~

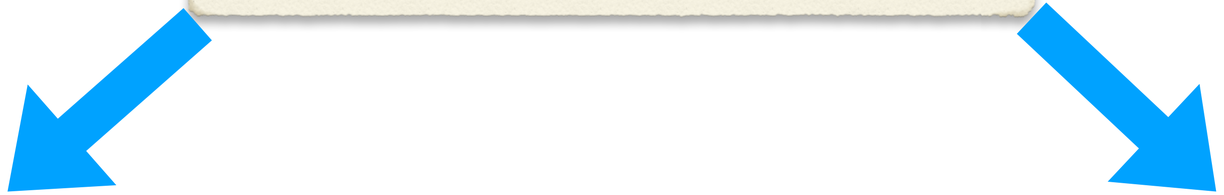
Slice representation is not guaranteed

- Lots of complicated, unsafe code on the Rust side
  - unsafe{} required to convert to/from C representation
- Requires scaffolding to make decent C++ interfaces

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]

✅

- Intentionally restrictive and opinionated
- 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)
- Dealing with callbacks, allocators, etc. is painful

zngur

```
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;
}
```

```
#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 🏎️

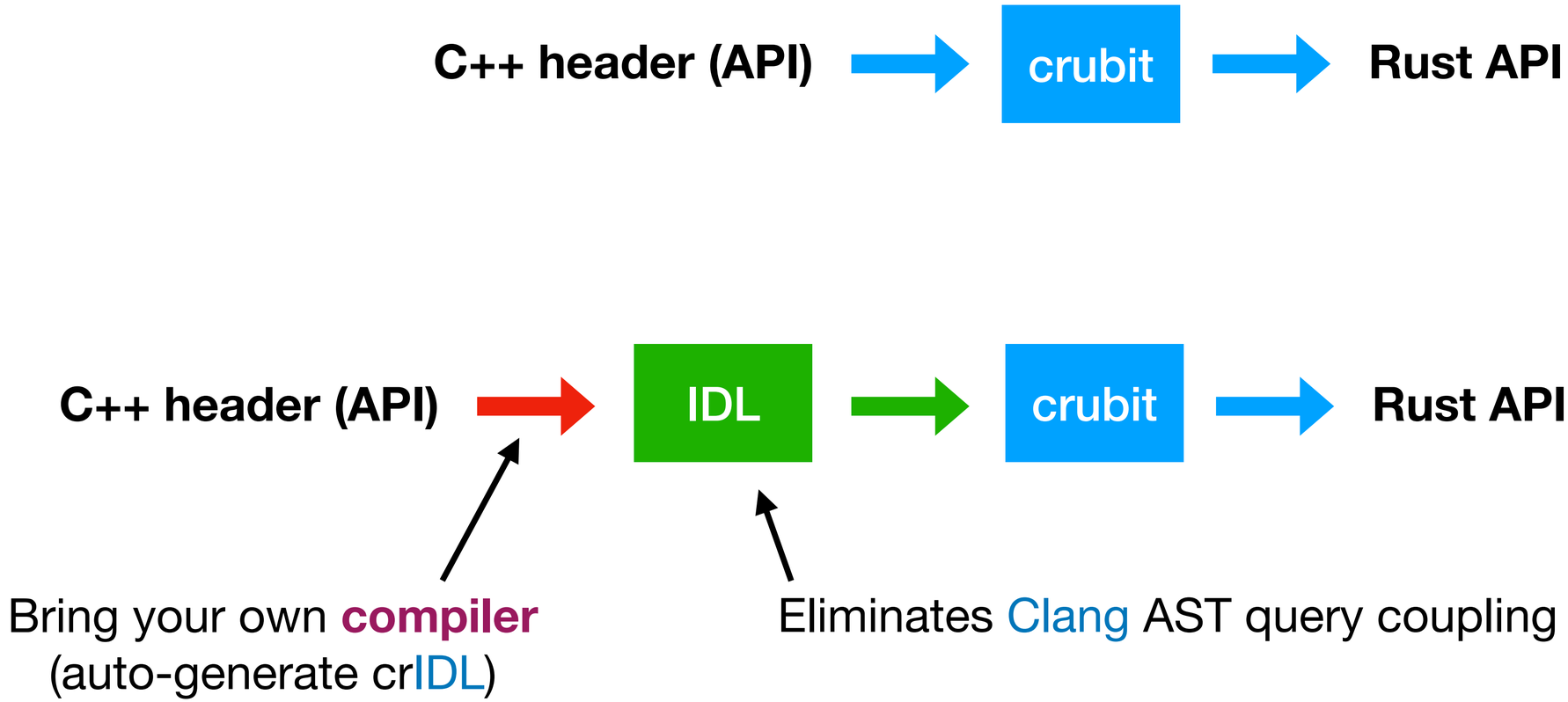
- Reduced need for pinning

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

crubit

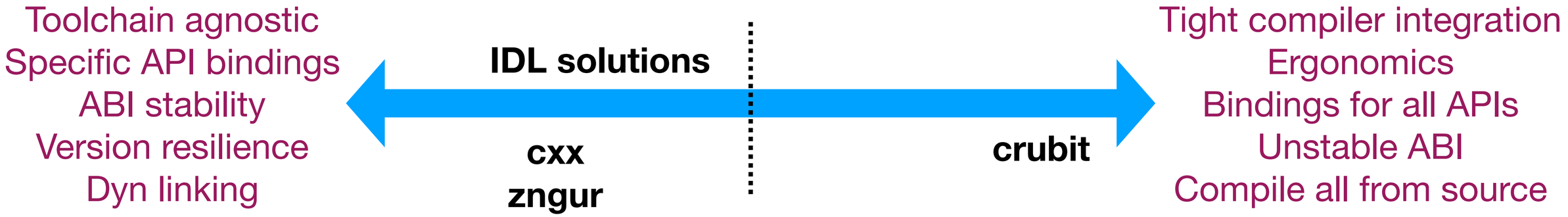
- Bold new project with the goal of high-fidelity lang interop between Rust and C++
- Needs native compiler integration (on both sides)
- Works directly on source files (no IDL needed)
- Covers the whole API surface (IDL-based solutions can be targeted)
- C++ compiler diversity: Clang (preview), ~~GCC, MSVC~~ (not today)
  - Optional "auto" IDL, opening up the C++ AST query interface to new compilers
- Pass by value: AllTheThings™ (that's where deep compiler integration comes in)
  - No layout manual annotations needed

crubit


C++ header (API)
crubit
Rust API
C++ header (API)
IDL
crubit
Rust API
Bring your own compiler
(auto-generate crIDL)
Eliminates Clang AST query 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 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

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*
- ...

Function Overloading

Ability for Rust to call overloaded C++ functions 😢 (required for full-fidelity API mapping)

- 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 C++ interop

(see **Carbon** vs. **Swift**)

- Can we solve this outside the core language?
  - Do we really need to hinder Rust powerful type inference with overloading?
- Some claim: we already have a clunky (hack) way to *simulate overloading* in Rust, so we might as well go all the way and do it right
  - 🏗️ New experiment (*rust-nightly*): traits + tuples + splat + #[overloaded] macro

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

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 C++ objects are genuinely non-relocatable:

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

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.
  - no code needs to run when moving the inline portion of these values to a new location
  - bitwise-relocatable types

- Many C++ libraries already optimize for such types
- Trivial relocation tries to standardize this optimization

I like to move it, move it...

wg21.link/P2786

Trivial Relocatability C++~~26~~

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

Pulled-out from '26 draft at the last minute

Polymorphic types on **ARM64e** needed to run code after relocation!

I like to move it, move it...

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

- Rust likes to move by default
  - does memcpy() on the bytes of T, regardless of type
  - render the moved-from object inaccessible
- C++ likes to copy by default
- C++ is by default needing move functions (ctor, =)
  - eg. std::string cannot be *memcopy*-ed due to SSO (self referential * in some implementations)
  - leaves the moved-from value accessible to be destroyed at the end of the scope
- 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++2~~6~~9 - Ability to declare C++ types bitwise-relocatable (annotate types)

Get standard library to be relocatable

=> allow most C++ types on the Rust stack (efficiency) 

Let's talk compilers!

Compilers & Interop

Many of the tricks here require deep compiler involvement

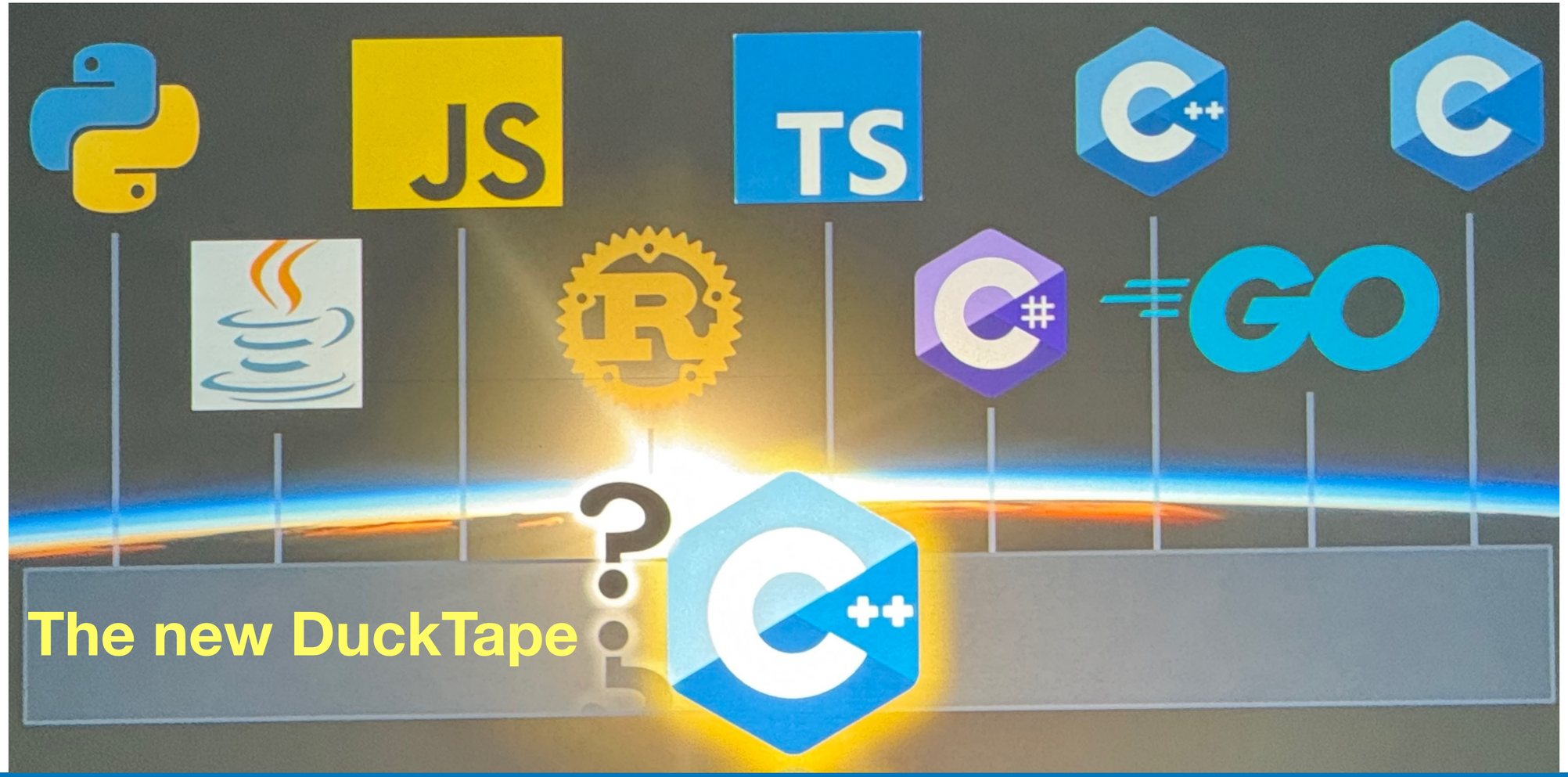
High-fidelity language semantics & mapping of std vocabulary types

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

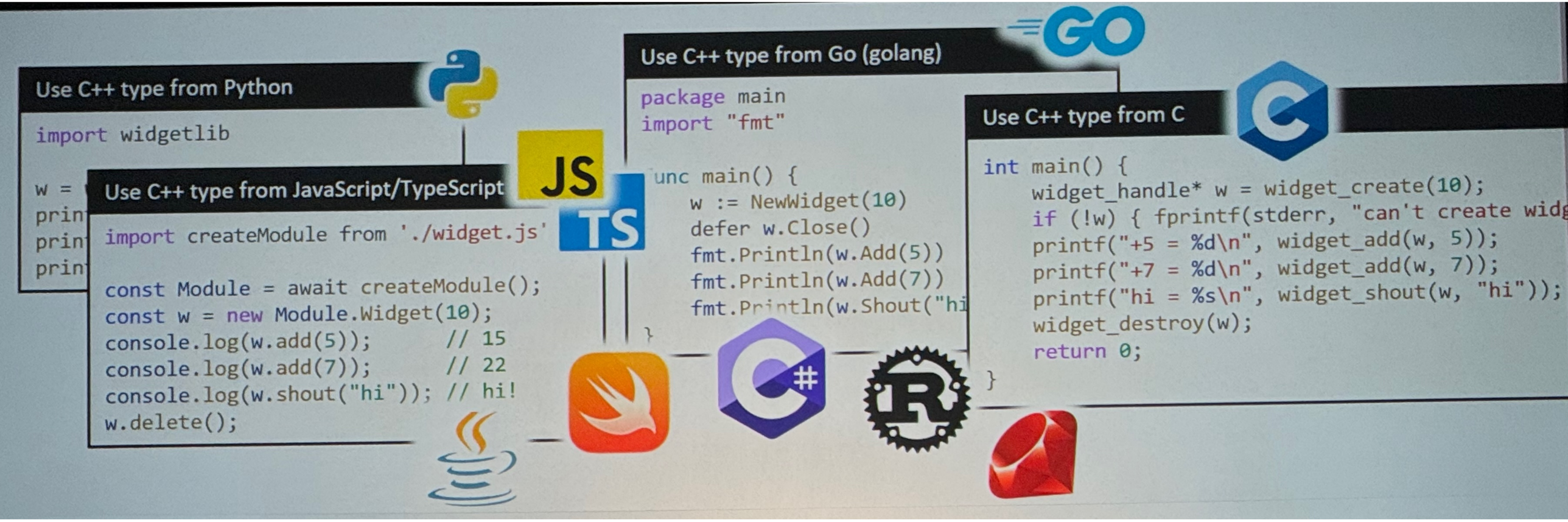
Binary-level fidelity, ABI, codegen, linking, dylib, etc.

Compilers & Interop

C++26 Reflection could be a game changer for lang interop!



C++26 Reflection could be a game changer for lang interop!



Herb Sutter: "Reflection: C++'s Decade-Defining Rocket Engine" (CppCon 2025)
youtube.com/watch?v=7z9NNrRDHQU

Address the 🐘 ABI in the room

google.com/search?q=victor+ciura+ABI

Who's driving this interop thing?

Active Effort (Interop)

2025-2026: Effervescent talks in the Rust Project & C++ community about this topic

/see: Rust Project Goals

Short term:

- 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
- Experimental compiler extensions

Rust Foundation joined INCITS in order to participate in the C++ ISO standards process

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++ folks 🙂

🗓️ Meetings:

2025

- 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 📝

2026

- May 21-23 Interop study group @ Rust-All-Hands - Notes 📝
- **July 8** Telecom sync - Notes 📝

Must watch 

RustConf
2025 | Seattle + Online

Fine-grained Rust / C++ interop

Taylor Cramer and Tyler Mandry

The original annual Rust programming language conference.

hosted by: Rust Foundation

youtube.com/watch?v=Z5M4NIWoMJQ

youtube.com/watch?v=k_sp5wvoEVM

Open Discussion

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

What are the interop requirements/challenges of your project?

Pure Virtual C++

Microsoft

Mind The Gap: C++/Rust Interop

@ciura_victor

@ciura_victor@hachyderm.io

@ciuravictor.bsky.social

Victor Ciura

Rust Tooling @ Microsoft