
a
b
c
d
e
f
g
h
i
j
k

Carcinization

NOT crabs ➡️

Carcinization

Carcinisation is a form of convergent evolution in which non-crab crustaceans evolve a crab-like body plan.

NOT crabs ➡️

Carcinization

Carcinisation is a form of convergent evolution in which non-crab crustaceans evolve a crab-like body plan.

NOT crabs ➡️

Carcinization

Carcinisation is a form of convergent evolution in which non-crab crustaceans evolve a crab-like body plan.

The term was introduced into evolutionary biology by L.A. Borradaile, who described it as:

NOT crabs ➡️

Carcinization

Carcinisation is a form of convergent evolution in which non-crab crustaceans evolve a crab-like body plan.

The term was introduced into evolutionary biology by L.A. Borradaile, who described it as:

"the many attempts of Nature to evolve a crab"

NOT crabs ➡️

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...

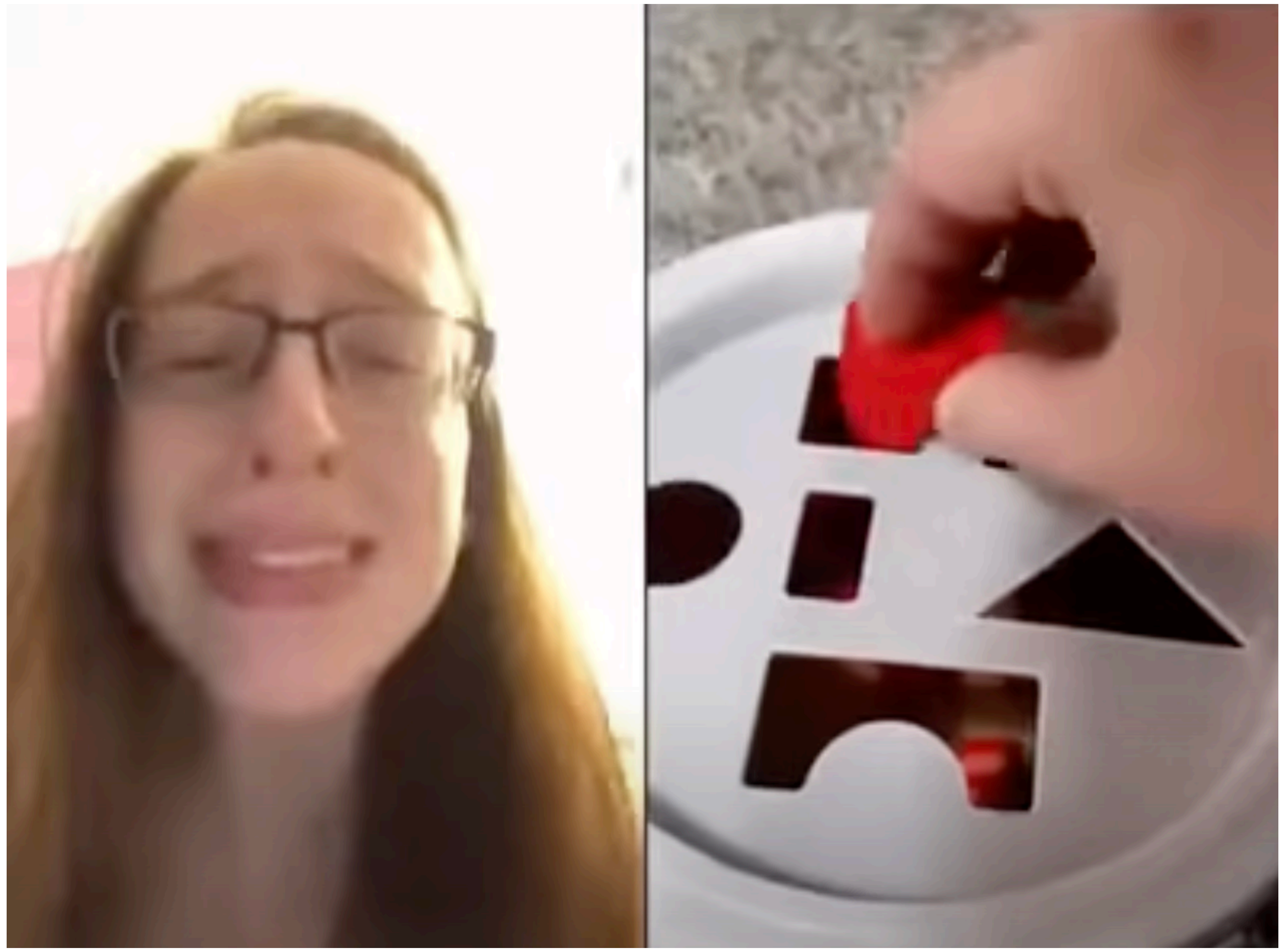
but so does C++ (that's on top of gazillion lines already out there)

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!

"But, Rust/C++ interop" 🥹



glue code

C FFI

unsafe

coge generators

(fat) compilers

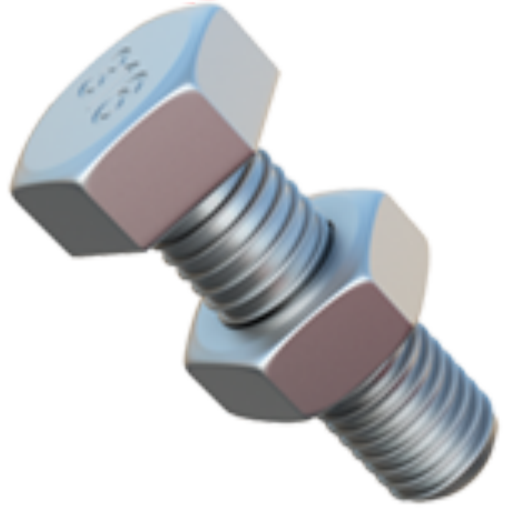
ergonomics

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

- 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

A story in 3 parts

- Attack of the Codegen
- The ABI Menace
- "Beam me up, Scotty!"

(sorry, wrong franchise)

Rust/C++ Interop:
Carcinization or Intelligent Design?

EuroRust

Paris, October 2025

@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

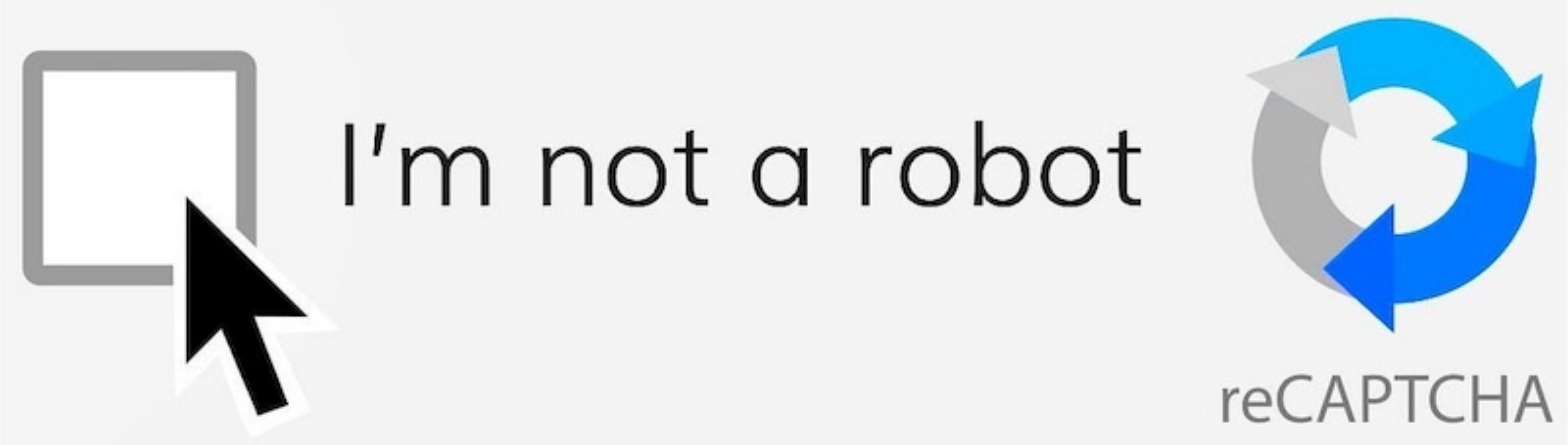
Disclaimer

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

What's out there...

⚠️ Disclaimer

No LLMs were hurt in the making of this presentation



100% artisanal code

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

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

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

void action(Widget * w);
```

➡️

```rust
#[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

```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 (allocations)

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)

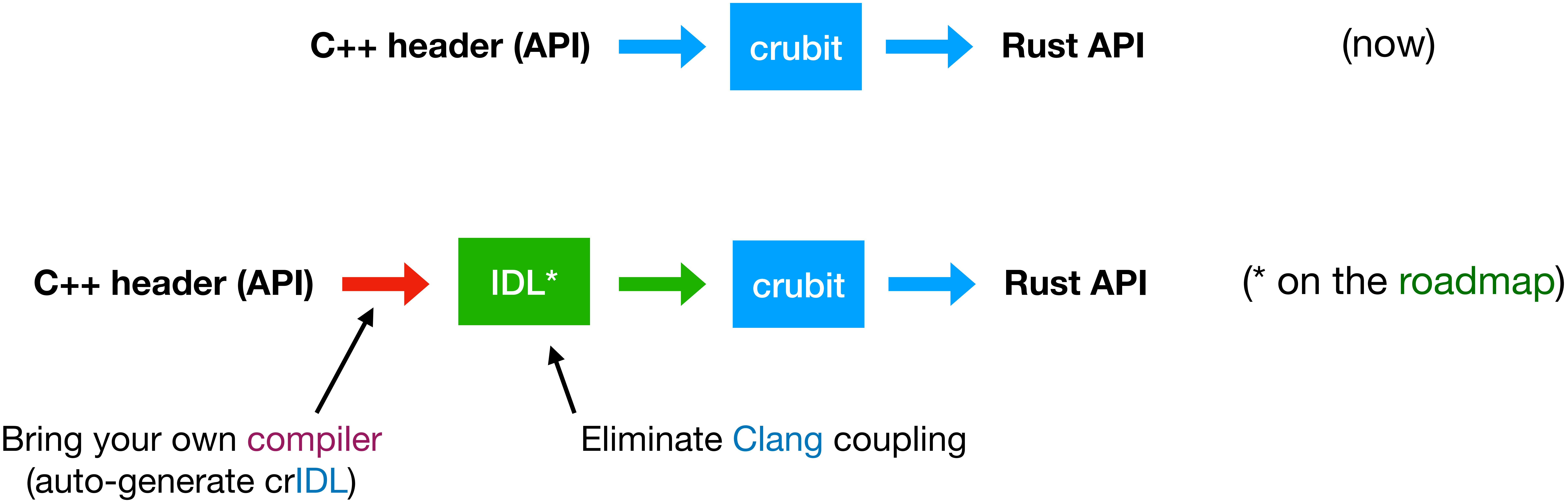
crubit

- 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

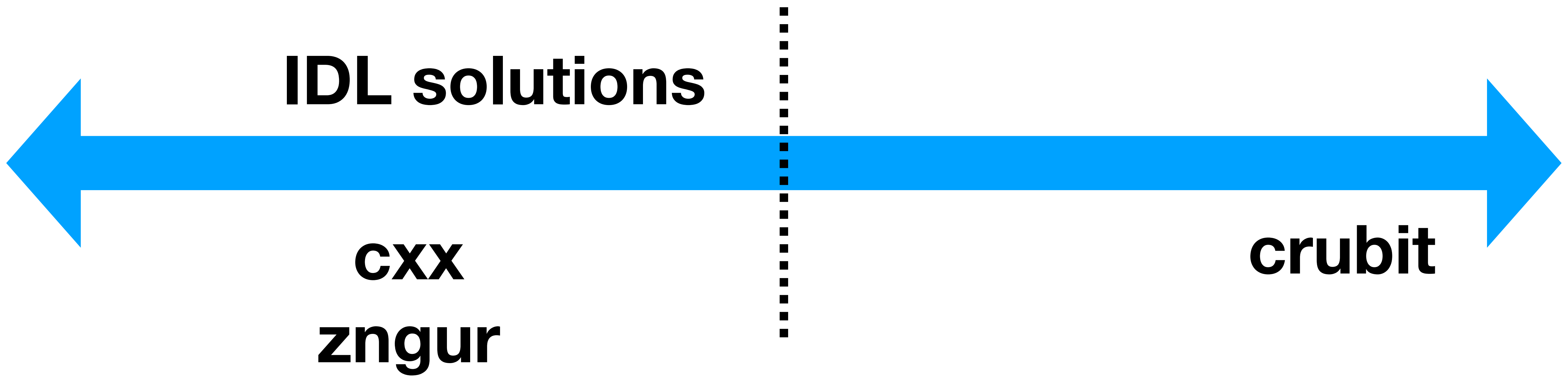


C++ header (API)
crubit
Rust API
(now)
C++ header (API)
IDL*
crubit
Rust API
(* on the roadmap)
Bring your own compiler
(auto-generate crIDL)
Eliminate Clang coupling

Tradeoffs...

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

Toolchain agnostic
Specific API bindings
ABI stability
Version resilience
Dyn linking



Tight compiler integration
Ergonomics
Bindings for all APIs
Unstable ABI
Compile all from source

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

Level: HARD!!!

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

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 <u>complicate</u> 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

teaser

The ABI Menace

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

C++ Tail Padding & Rust ABI

C++ Tail Padding & Rust ABI

C++ Tail Padding & Rust ABI

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

C++ Tail Padding & Rust ABI

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

C++ 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++ 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

C++ 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

C++ 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

C++ 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

C++ 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)

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 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

Trivial Relocatability C++26

Safely relocate objects in memory

wg21.link/P2786

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

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

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

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

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:

#def

- 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

Just C++ being comical, again...

I like to move it, move it...

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

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

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

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

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

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

restart_lifetime()

A Lifetime-Management Primitive
for Trivially Relocatable Types

📖 wg21.link/P3858

`restart_lifetime()`

- a call to `trivial_relocate()` performs a logically atomic operation whereby an object's representation is copied, its lifetime is ended at the original location, and its lifetime is restarted at the target location - without invoking any constructors or destructors

A Lifetime-Management Primitive
for Trivially Relocatable Types

📖 wg21.link/P3858

`restart_lifetime()`

- a call to `trivial_relocate()` performs a logically atomic operation whereby an object's representation is copied, its lifetime is ended at the original location, and its lifetime is restarted at the target location - without invoking any constructors or destructors
- new `restart_lifetime()` function in the `start_lifetime_as()` series of std functions

A Lifetime-Management Primitive
for Trivially Relocatable Types

📖 wg21.link/P3858

`restart_lifetime()`

- a call to `trivial_relocate()` performs a logically atomic operation whereby an object's representation is copied, its lifetime is ended at the original location, and its lifetime is restarted at the target location - without invoking any constructors or destructors
- new `restart_lifetime()` function in the `start_lifetime_as()` series of std functions
- it allows us to separate the "memory copying" aspect of relocation from restarting the object's lifetime at the new memory address

A Lifetime-Management Primitive
for Trivially Relocatable Types

📖 wg21.link/P3858

restart_lifetime()

- a call to `trivial_relocate()` performs a logically atomic operation whereby an object's representation is copied, its lifetime is ended at the original location, and its lifetime is restarted at the target location - without invoking any constructors or destructors
- new `restart_lifetime()` function in the `start_lifetime_as()` series of std functions
- it allows us to separate the "memory copying" aspect of relocation from restarting the object's lifetime at the new memory address

```
template<class T>
requires /* ... */
T* trivially_relocate(T* first, T* last, T* result)
{
  std::memcpy( result,
               first,
               (last-first)*sizeof(T));
  for(size_t i = 0; i < (last-first); ++i)
    std::restart_lifetime(result[i]);
}
```

A Lifetime-Management Primitive
for Trivially Relocatable Types

📖 wg21.link/P3858

Let's talk compilers!

Compilers & Interop

Many of the tricks here require deep compiler involvement

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

Compilers & Interop

Many of the tricks here require deep compiler involvement

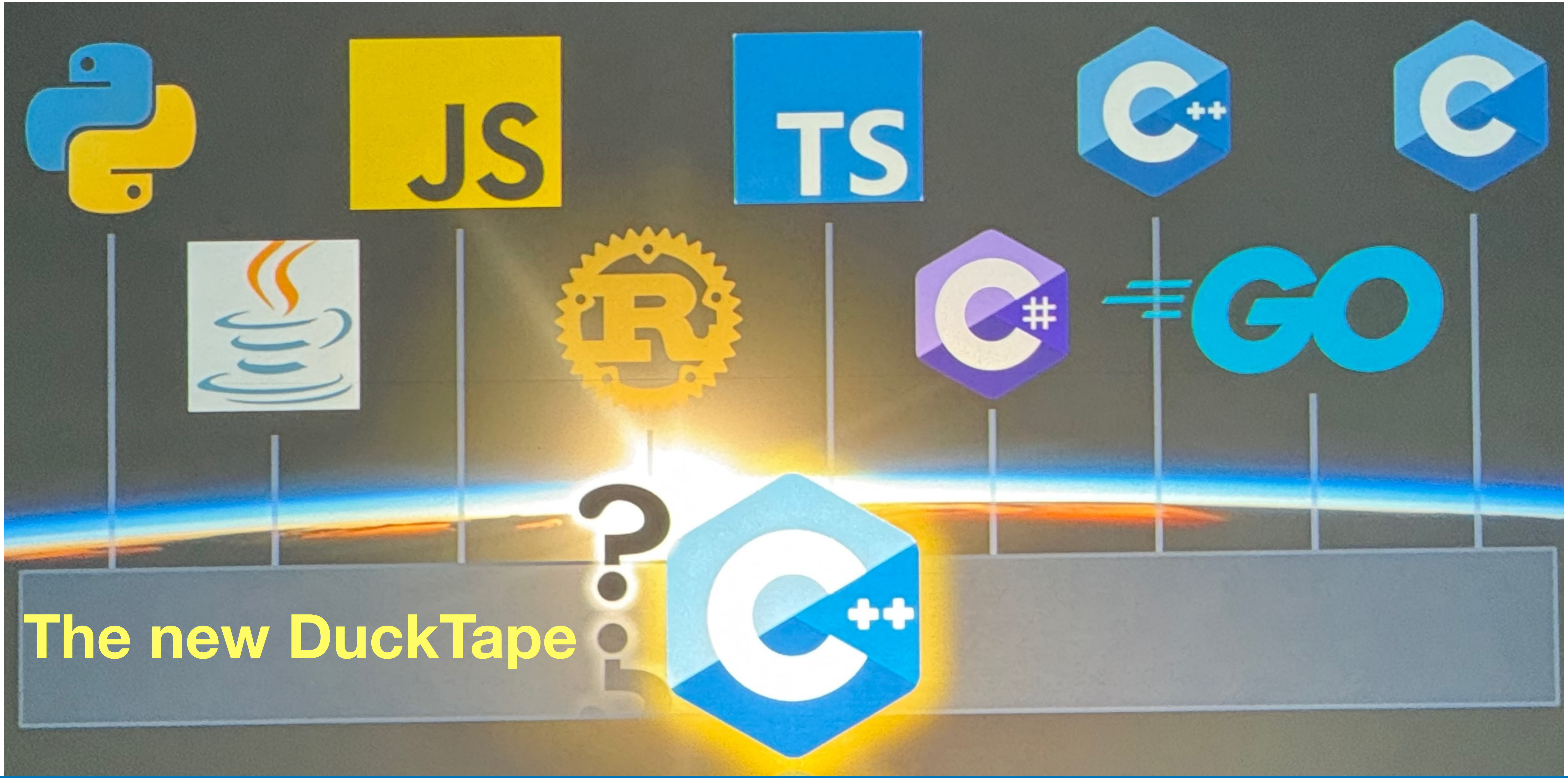
High-fidelity language semantics & mapping of std vocabulary types

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

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

Compilers & Interop

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



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

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

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++) -- eg. Rust25H1 Project Goals

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++) -- eg. Rust25H1 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

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++) -- eg. Rust25H1 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

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++) -- eg. Rust25H1 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

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

We are crubit

youtube.com/watch?v=Z5M4NIWoMJQ

Open Discussion

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

What are the interop requirements/challenges of your project?

Rust/C++ Interop:
Carcinization or Intelligent Design?

EuroRust

Paris, October 2025

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

Victor Ciura
~~Principal Engineer~~
Rambling Idiot
Rust Tooling @ Microsoft