
# Rust for The Curious C++ Developers

Victor Ciura

# Part 1

## A (provocative) premise

# It compiles. Ship it?

> *"Your program segfaults? It's a skill issue. Wax on, wax off, grasshopper."*

- The implicit C++ promise: **a good enough engineer + enough review + enough sanitizers = correct software.**
- "It compiles" tells you the program is **well-formed**, not that it is **well-behaved**.
- For 40 years we have treated the gap between *compiles* and *correct* as the programmer's personal responsibility.
- **What if the compiler could close more of that gap for you?**

# Rust is not (just) about memory safety

## The real product is correctness (by construction)

- Memory safety is the *famous* feature. Correctness is the *important* one.
- Rust's recurring design question: **"What mistakes do programmers keep making - and how can the language prevent them?"**
- Move whole bug *classes* from runtime to compile time:
  - null dereferences, ignored errors, unhandled variants, use-after-free, data races
- The sentiment "**once it compiles, it works**" is not *literally* true - but it's *meaningful*, because the compiler has already eliminated a long list of ways to screw up.

# Core Principle: make invalid states unrepresentable

- The cheapest bug is the one you **cannot write down**.
- Rust pushes that idea into everyday types:
  - no implicit `null`
  - errors are *values* you must handle
  - a closed set of cases the compiler forces you to cover
  - Typestate pattern
  - ownership and aliasing checked, not assumed
- **Design so that the wrong program won't type-check.**

# Common myths

| Myth | Reality |
| --- | --- |
| "Rust is *just* a memory-safety language" | Memory safety is a *side effect* of a correctness-first design |
| "Rust wants to replace C++" | Rust & C++ will have to coexist in projects for many years (good **interop** is the key) |
| "You'll spend your life fighting the borrow checker" | Fighting the borrow checker usually means your **ownership model is unclear** - it's a design smell, not a tax |
| "Rust is hard to learn" |  |

# The Thesis

- Rust combines **high-level functional programming** with **low-level control**.
- It spans the **widest domain range** of any mainstream language: firmware & drivers, kernels, tools, microservices, WASM, web apps.
- It treats **ergonomics as a first-class design value** - without giving up performance.
- It is **never surprising**: behavior is explicit in the code, and diagnostics guide you to the fix.

# Zero-cost vs. Ergonomics

### C++

- Start from **primitives** and **zero-cost abstractions**.
- Hold features back until they are **100% generic** and universal.
- Pay in boilerplate, ceremony, and slow library-to-language loops.

### Rust

- Start from the **user POV**: parse this CLI, send this message, define this API route.
- Ship an ergonomic answer for the **95% common case**.
- Prove ideas in libraries, then promote them into the language.

Neither is "wrong" - but the defaults shape who the language serves.

# Part 2

## My favorite features

# Everything is an expression

**"Expressions yield values. Statements do not."**

- In Rust, `if`, `match`, `loop`, and `{ blocks }` are **expressions** - they evaluate to a **value**.
- The last expression in a block *is* the block's value (tail).
- This nudges you toward **value-based programming**: build a value and bind it, instead of declaring a variable and mutating it into shape.

```
let roll = if feeling_lucky { 6 } else { 4 };   // if is an expression

let label = match roll {                          // so is match
    6 => "lucky",
    _ => "ordinary",
};
```

# Move is the default

**Rust - assignment moves ownership**

```rust
let s  = String::from("Rust");
let s1 = s;            // ownership MOVES to s1
println!("{s1}");      // ok
// println!("{s}");    // compile error:
//   borrow of moved value: `s`
```

**C++ - assignment copies**

```cpp
std::string s  = "Rust";
std::string s1 = s;          // deep COPY (silent cost)
std::cout << s;              // still valid

std::string s2 = std::move(s); // now s is
std::cout << s;  // "valid but unspecified" - compiles, wrong
```

- Rust: one owner at a time. Use-after-move is a **compile error**, not a runtime surprise.
- C++: copy is the silent default *cost*; `std::move` leaves a usable-but-empty *husk* (potential runtime issue).

# Functional: composable, lazy iterators

**Rust - lazy, zero-cost iterator chain**

```rust
let total: u32 = (1..=100)
    .filter(|n| n % 2 == 0)
    .map(|n| n * n)
    .sum();
// no intermediate vectors; compiles to a tight loop
```

**C++20 - ranges (more ceremony, slower to compile)**

```cpp
auto evens = std::views::iota(1, 101)
    | std::views::filter([](int n){ return n % 2 == 0; })
    | std::views::transform([](int n){ return n * n; });
int total = std::accumulate(
    evens.begin(), evens.end(), 0);
```

- Iterators are **lazy** (nothing runs until consumed) and **fusing** (no temporaries).
- High-level *functional* pipeline → low-level machine code.

# The mighty `enum`

**Rust's most versatile feature**

# Reminder: Product types, sum types

### Product type - AND

A `struct` holds *this* **and** *that*.

```
struct Point {
    x: f64,
    y: f64,
}   // always an x AND a y
```

### Sum type - OR

An `enum` is *this* **or** *that* - a closed set of alternatives.

```
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
}   // a Shape is a Circle OR a Rect
```

# Rust `enum` vs `std::variant`

## Same idea, very different ergonomics

### Rust - first-class language feature (extensively used)

```rust
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
}

fn area(s: &Shape) -> f64 {
    match s {
        Shape::Circle(r)     => PI * r * r,
        Shape::Rect { w, h } => w * h,
    }
}
```

### C++ - std::variant + visitor boilerplate

```cpp
struct Circle { double r; };
struct Rect   { double w, h; };
using Shape = std::variant<Circle, Rect>;

// you must hand-write this helper:
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

double area(const Shape& s) {
    return std::visit(overloaded{
        [](const Circle& c){ return PI * c.r * c.r; },
        [](const Rect&   r){ return r.w * r.h; },
    }, s);
}
```

# Why this matters

## A basic modeling tool, used everywhere

- Because the syntax is compact, enums become an **everyday** tool - almost as common as structs.
- Model events, states, protocol messages, AST nodes, results - directly in the type system.

```
enum WebEvent {
    PageLoad,                   // no payload
    KeyPress(char),             // a character
    Paste(String),              // an owned string
    Click { x: i64, y: i64 },   // named fields
}
```

- Each alternative carries *exactly* its own data - no empty structs, no *"valid only if..."* fields.

# Pattern matching & impossible states

## The payoff of sum types

# match + enums

```
fn describe(event: &WebEvent) -> String {
    match event {
        WebEvent::PageLoad           => "page loaded".into(),
        WebEvent::KeyPress(c)        => format!("key: {c}"),
        WebEvent::Paste(text)        => format!("pasted: {text}"),
        WebEvent::Click { x, y }     => format!("click @ {x},{y}"),
        // delete an arm  ->  error[E0004]: non-exhaustive patterns
    }
}
```

- `match` **destructures** the payload and **binds** it in one move.
- It's **exhaustive** by default: forget a case and it won't compile.
- Add a new variant later → the compiler lists *every* `match` you must update.

# `Option<T>`: the end of the billion-dollar mistake

**Rust - absence is in the type**

```rust
fn first_word(s: &str) -> Option<&str> {
    s.split_whitespace().next()
}

match first_word(line) {
    Some(w) => println!("first: {w}"),
    None    => println!("(empty)"),
}   // you CANNOT use w without proving it exists
```

**C++ - std::optional, but unchecked**

```cpp
std::optional<std::string>
first_word(const std::string& s);

auto w = first_word(line);
std::cout << *w;   // compiles, nothing forced the check
```

- There is no `null` in safe Rust. Maybe-absent values are `Option<T>` - `Some(x)` or `None`.
- The compiler won't let you touch the value until you've handled `None`.

# Make impossible states unrepresentable

**C++ - flags + maybe-valid fields**

```cpp
struct Request {
    std::string url;
    bool done;
    bool ok;
    std::string body;   // valid iff done && ok
    std::string error;  // valid iff done && !ok
};
// done=false, ok=true, body set?  compiles.
// many nonsense combinations are representable
```

**Rust - the type only allows real states**

```rust
enum Request {
    Pending { url: String },
    Success { url: String, body: String },
    Failed  { url: String, error: String },
}
// "succeeded but also has an error" cannot
// be constructed.
```

**If the bad state can't be built, the bug can't be written.**

# Errors as values

**No exceptions. No ignored return codes.**

# `Result<T, E>`

```
enum Result<T, E> {     // it's just an enum
    Ok(T),
    Err(E),
}

fn parse_port(s: &str) -> Result<u16, std::num::ParseIntError> {
    let n = s.parse::<u16>()?;     // on Err: return it now
    Ok(n)
}

match parse_port(input) {
    Ok(port) => start_server(port),
    Err(e)   => eprintln!("bad port: {e}"),
}
```

- Fallible functions **return** `Result`. Always.
- `#[must_use]`: ignore a `Result` and the compiler **warns** you.

# The `?` operator sugar

### Rust - explicit, one character

```rust
fn process(path: &str)
    -> Result<String, std::io::Error>
{
    let content =
        std::fs::read_to_string(path)?; // propagate up
    Ok(content.to_uppercase())
}
```

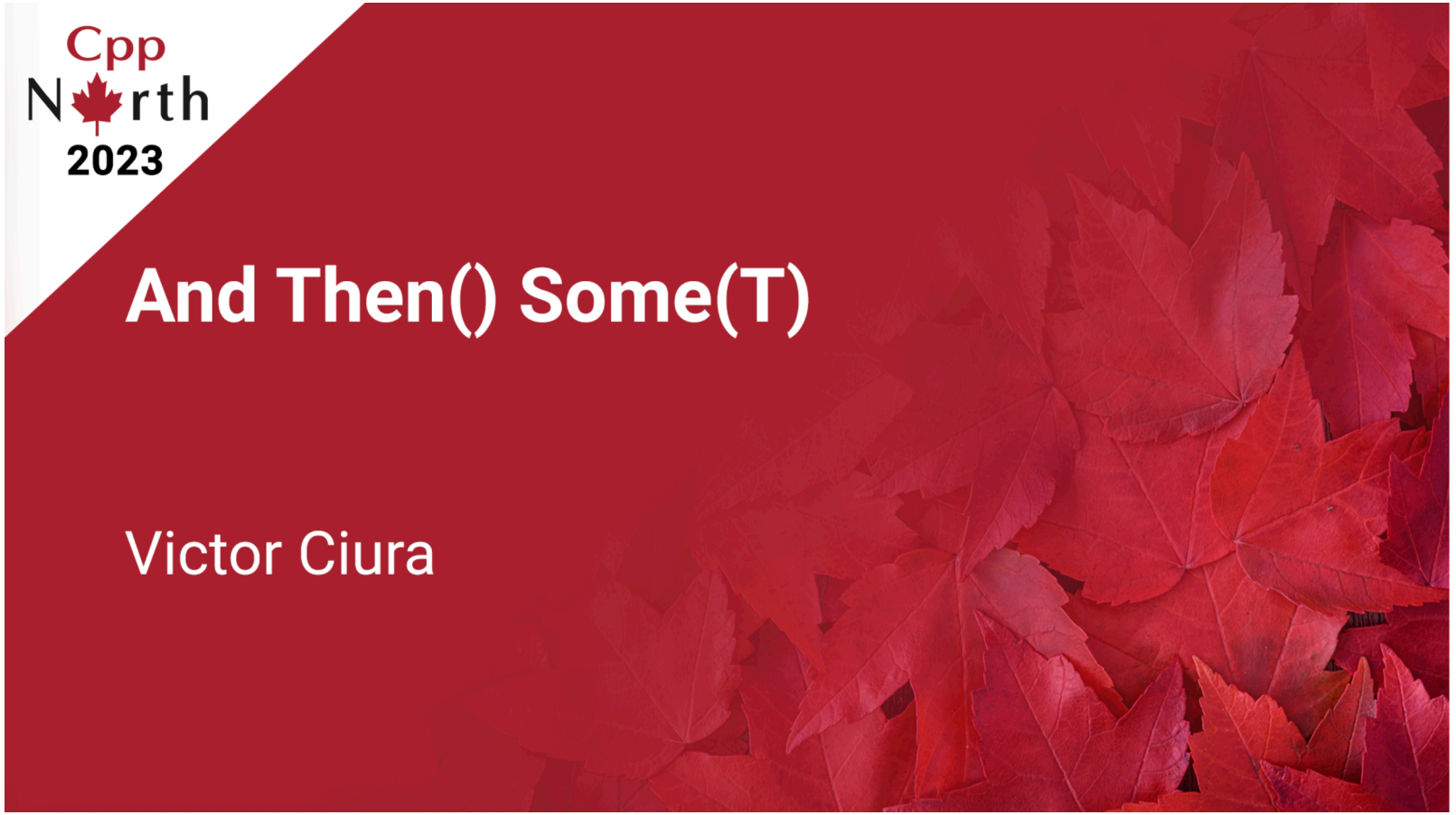
`?` means: *if* `Err`*, return it; else unwrap the* `Ok`. The error path is **visible**.

### C++ - exceptions, invisible at the call site

```cpp
std::string read_config(const std::string& p) {
    std::ifstream f(p);
    if (!f.is_open())
        throw std::runtime_error("Cannot open: " + p);
    std::string c; std::getline(f, c);
    return c;            // signature warns no one
}

auto cfg = read_config("nope.txt"); // try/catch here?
```

# Programming in the



# Don't look inside the 📦

### A "box" wraps a value in a context

```
unique_ptr<T>   ->  *p     p.get()
shared_ptr<T>   ->  *p     p.get()
vector<T>       ->  v[0]   *v.begin()
optional<T>     ->  *o     o.value()
expected<T,E>   ->  *e     e.value()
```

### The temptation: rip the box open and peek 👀

```
optional<string> s = parse(input); // may fail
string cap;
if (s)                          // peek
    cap = capitalize(*s);       // unwrap by hand
```

```
let s: Option<String> = parse(input);
let cap = match s {             // peek
    Some(v) => Some(capitalize(v)),
    None    => None,
};
```

### Functional move: act on the value *without* breaking the box.

# Lifting: change the contents, keep the 📦

A function `A -> B`, *lifted*, becomes `Box<A> -> Box<B>` - no peeking required.

**Rust - `map`, on `Option`**

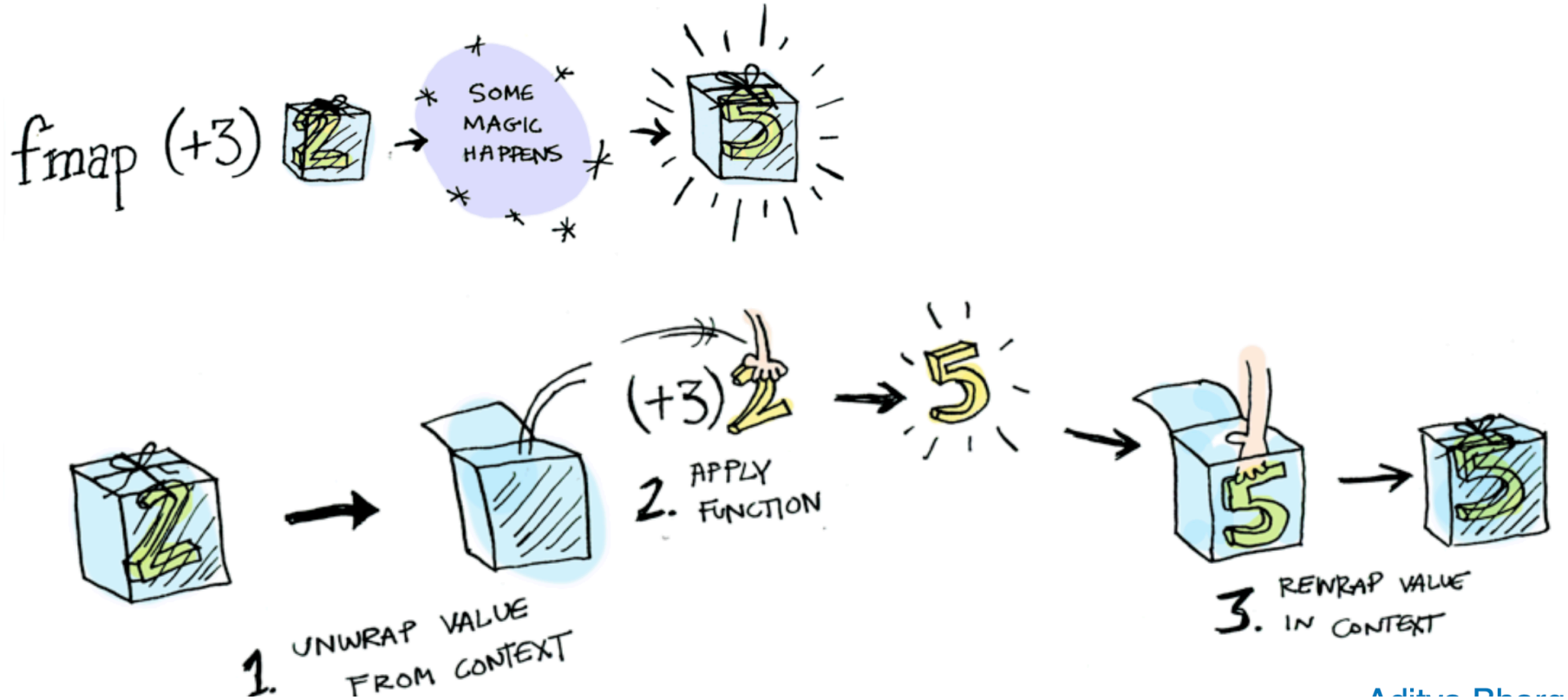
```rust
// capitalize: String -> String
let cap: Option<String> =
    parse(input).map(capitalize);
```

**C++23 - `transform` (P0798)**

```cpp
// capitalize: string -> string
std::optional<std::string> cap =
    parse(input).transform(capitalize);
```

- The generic "lift any function over the box" is the classic `fmap` / Functor `map`.
- Same shape in both languages now - Rust shipped it in 1.0, C++ standardized it in C++23.

# The functional move, visualized




The magic happens *inside* the box - `map` / `transform` do all steps for you.

# Chaining fallible steps

**Rust**

```rust
fn debug_location(pc: Option<u64>)
    -> Option<String>
{
    pc.and_then(load_symbol) // -> Option<Symbol>
      .map(symbol_name)      // -> Option<String>
}
```

**C++23 - same operation, same names**

```cpp
optional<string> debug_location() {
  return current_pc
      .and_then(load_symbol)  // optional<Symbol>
      .transform(symbol_name);// optional<string>
}
```

- A step that itself returns a box would make `map` / `transform` yield `Option<Option<T>>`.
- `and_then` (monadic *bind*, `>>=`) **flattens** as it chains - no `o.value().value()`.

# The same pipeline, two languages

### Rust - `Option` combinators

```rust
let msg = parse_int(sv)
    .filter(|&v| v > 0)
    .map(|v| v.clamp(0, MAX_LOGS))
    .map(|v| format!("Collecting {v} logs."))
    .unwrap_or_else(|| "Log error".into());
```

### C++23 - monadic `std::optional`

```cpp
auto msg = string_view_to_int(sv)
  .and_then([](int v) -> optional<int> {
     int n = std::clamp(v, 0, max_logs);
     return n > 0 ? optional{n} : nullopt; })
  .transform([](int v) {
     return std::format("Collecting {} logs.", v); })
  .or_else([] { return optional<string>{"Log error"}; })
  .value();
```

# Heritage: the same idea, three languages

| Concept | Haskell (1990) | Rust (2015) | C++ (2017-2023) |
|---|---|---|---|
| Maybe-a-value | `Maybe a` | `Option<T>` | `std::optional<T>` |
| Value-or-error | `Either e a` | `Result<T, E>` | `std::expected<T, E>` |
| Map (Functor) | `fmap` | `.map()` | `.transform()` |
| Bind (Monad) | `>>=` | `.and_then()` | `.and_then()` |
| Fallback | `mplus` | `.or_else()` | `.or_else()` |
| Sum type + match | `data` + `case` | `enum` + `match` | `variant` + `visit` |

**"When in doubt, draw inspiration from Haskell or Rust." - and C++ did.**

# Fearless concurrency

**Memory safety AND concurrency safety - in the type system**

# It starts with aliasing: shared XOR mutable

```
fn main() {
    let mut x = 42;
    let r1 = &x;          // immutable borrow
    let r2 = &mut x;      // error[E0502]: cannot borrow `x` as mutable
                          // because it is also borrowed as immutable
    println!("{r1} {r2}");
}
```

- The borrow checker enforces one rule: **many readers, XOR one writer** - never both at once
- No aliased mutable access means **no data races by construction**
- The same rule scales to **concurrency**

# `Send` and `Sync`: data races become type errors

### The compiler rejects a data race

```
use std::rc::Rc;
use std::thread;

let data = Rc::new(42);
thread::spawn(move || {
    println!("{}", data);  // error: `Rc<i32>`
});                        // cannot be sent between
                           // threads safely (!Send)
```

### Two marker traits

- **`Send`** - safe to **move** to another thread.
- **`Sync`** - safe to **share** ( `&T` ) across threads.
- Auto-derived for most types. **Not** for:
  - `Rc<T>` (non-atomic refcount) → use `Arc<T>`
  - `RefCell<T>` → use `Mutex` / `RwLock`
  - raw pointers
- `thread::spawn` *requires* `Send`. The wrong type simply won't compile.

# Talking between threads: channels

```
use std::sync::mpsc;
use std::thread;

let (tx, rx) = mpsc::channel();
for id in 0..4 {
    let tx = tx.clone();                    // one sender per producer
    thread::spawn(move || tx.send(id * id).unwrap());
}
drop(tx);                                   // close the last sender
let total: i32 = rx.iter().sum();           // receiver is just an iterator
```

- Prefer **message passing** over shared mutable state + locks.
- Senders **move** into thread closures - ownership makes "who can touch this" unambiguous.
- A safer, higher-level default than hand-rolled `std::thread` + `mutex` protocols.

# Recap: five features, one idea

**Make invalid states unrepresentable in the type system (fail to compile)**

# Part 3

## Tooling, workflows & engineering systems

## Shipping Rust at industrial scale

# Cargo: one tool for everything

### Rust - one tool, conventions built in

```
cargo new app        # scaffold a project
cargo add serde      # add a dependency
cargo build          # compile
cargo run            # build + run
cargo test           # unit + integration + doctests
cargo doc --open     # generate & view docs
cargo clippy         # 600+ lints
cargo fmt            # canonical formatting
```

### C++ - assemble your own

- Build: Make / CMake / Bazel / Meson...
- Deps: vcpkg / Conan / system / submodules...
- Test: GoogleTest / Catch2 / ctest...
- Lint: clang-tidy · Format: clang-format
- Docs: Doxygen
- Each with its own config, glue, and CI wiring.

`cargo new` → working "Hello, world" in seconds. Dependencies are cached, not installed globally.

# Dependencies & supply chain

- A **crate** is a package; **crates.io** is the registry; `Cargo.toml` declares deps, `Cargo.lock` pins them (reproducible builds).
- Rust's answer is **tooling, not faith**:
  - `cargo audit` - flags dependencies with known vulnerabilities
  - `cargo deny` - policy gate: licenses, duplicates, banned/yanked crates
  - `cargo vet` / `cargo crev` - share human audits of dependencies across orgs
  - `cargo aprz` - appraise the quality of Rust dependencies

# Crates at enterprise scale: trust, but verify

### A thriving registry...

- crates.io is huge and fast-moving - the productivity is real.
- ...but every dependency is **code from the internet**, often from a single maintainer.

### ...needs an enterprise posture

- `cargo vet` - record that deps were **audited** by someone you trust.
- A **crate review system**: which crates to use, preferred picks, a scoring rubric.
- Rigorous **SBOM**; private **1P registries** (internal feeds, internal `docs.rs`).

**Lower the risk of leaning on a hobby crate with one absent owner.**

# Documentation and tests that can't go stale

### Doctests - examples that are compiled AND run

```
/// Doubles a number.
///
/// ```
/// assert_eq!(mylib::double(21), 42);
/// ```
pub fn double(n: i32) -> i32 {
    n * 2
}
```

### Unit tests - next to the code they test

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn doubles() {
        assert_eq!(double(2), 4);
    }
}
```

- `cargo test` runs your unit tests, integration tests, **and** every code example in your docs.
- Docs that drift out of date **break the build**. Tests live beside the implementation, not in a far-off folder.

# Helpful Diagnostics

A paper cut for an experienced user can be a blocker for a new user.

The compiler should help, not punish.

### A real rustc diagnostic

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:4:15
  |
2 |   let s1 = s;
  |            - value moved here
4 |   println!("{s}");
  |            ^^^ value borrowed
  |                 after move
  |
help: consider cloning the value
  |
2 |   let s1 = s.clone();
  |             +++++++++
```

### Diagnostics on purpose

- Points at the **location**.
- Explains the **why**.
- Suggests a **fix** (often paste-ready).
- Links to a longer explanation ( `rustc --explain E0382` ).

# Verify, measure, harden

- **Correctness of `unsafe`**: **Miri** interprets your code and detects UB; ASan/TSan/UBSan also available.
- **Performance**: `cargo bench` + **criterion** for statistically sound benchmarks; built-in flamegraph tooling.
- **Coverage**: `cargo llvm-cov` shows what your tests miss.
- **Ship small & fast**: release profiles tune `opt-level`, **LTO**, `codegen-units`, `panic=abort`, symbol stripping.
- **CI/CD**: `build → clippy → test → audit → coverage` is a handful of lines; reproducible via the lockfile.

# Evolving without breaking: Editions

- A compiler **version** is global; an **edition** (2015 / 2018 / 2021 / 2024) is **per-crate**.
- Crates on **different editions interoperate** in the same build.
- Editions can introduce new keywords/syntax (e.g. `async` / `await`) **without breaking old code**.
- The result: Rust evolves continuously - **no Python 2/3 schism**.

# From the field: what got better 🙂

- **Memory-safety vulnerabilities** down; **data-race** concurrency bugs down.
- Rich ecosystem + streamlined dependency management.
- Explicit memory management makes engineers **more conscious of pitfalls**.
- Less friction to write tests => **more tests** actually get written.
- **Coding agents / Copilot flatten the learning curve.**

# Honest rough edges

### Still developing...

- IDE experience: **async debugging is painful**.
- Integrating `cargo` with a larger build system.
- **Dynamic/static linking** is challenging; safe **FFI** is hard, ABI matters.
- Language **interop** is not seamless (yet).
- A few features teams rely on aren't **stabilized** yet.

### "Async freedom" - a trap!

- Low-level systems (VMMs, kernels, DB engines) need **custom executors** for thread control and IO perf.
- Everyone else: **just use `tokio` - then measure.**

**But we still ship Rust - the wins outweigh the gaps.**

# Rust ↔ C++ interop

They must **play nice together for a long time**.

The wish list for first-class interop:

- No excessive `unsafe`, no hand-written C shims.
- No marshaling overhead.
- Broad type support **with safety** - not just C-ABI primitives.
- Works with real DLLs, the CRT, the C++ ABI.
- cargo ⇄ MSBuild / CMake / Bazel / Buck2
- Lowering everything through a C FFI is the easy, lossy answer.
- **High-fidelity** language semantics & types **mapping**.
- The goal: **ergonomics with safety**, automated, at scale.


NDC { TechTown }
Rust/C++ Interop:
Carcinization or
Intelligent Design?
Victor Ciura
Principal Engineer, Microsoft

# Takeaways

## What can you bring back to C++?

1. Push whole bug classes to *compile time*. **Correct by construction**.
2. The recurring pattern: **make invalid states unrepresentable.**
3. **Ergonomics is a feature -** and it compounds, for beginners and experts alike.
4. **Functional** *and* low-level control.
5. Be **curious**: you don't have to switch to learn something that improves your C++.

# Rust for The Curious C++ Developers

*Let's talk...*

Victor Ciura