

Masterclass

Rust for C/C++ Developers

{FRQ DEV 2025}
{celebrate.community.code.}

Raphaël Coëffic - CTO @ **FRAFOS**
Member of the Frequentis Group

Rust for C++ Developers

- Part 1: Why Use Rust?
- Part 2: Essential Rust Concepts for C++ Developers
- Part 3: Getting Started

Rust for C++ Developers

Part 1

Why Use Rust?

- Memory Safety
- Fearless Concurrency
- Productivity
- Good Use Cases for Rust
- Migration Strategy
- Key Takeaways for C/C++ Developers

Why Use Rust?

Memory Safety

The Problem with C/C++:

- Buffer overflows, use-after-free, double-free errors
- 70% of security vulnerabilities in major software are memory safety issues
- Manual memory management leads to subtle bugs that are hard to catch

Why Use Rust?

Memory Safety

C++ (Potential Bug):

```
int main() {  
    std::vector<int> data = {1, 2, 3};  
  
    int* reference = &data[0];  
    data.clear(); // Dangling pointer!  
  
    std::cout << *reference << std::endl; // Undefined behaviour  
    return 0;  
}
```

Why Use Rust?

Memory Safety

Rust's Solution:

- Ownership system prevents memory safety issues at compile time
- Zero-cost abstractions: no runtime overhead
- No garbage collector needed
- RAI (Resource Acquisition Is Initialisation) taken to the next level

Why Use Rust?

Memory Safety

Rust (Compile-time Safety):

```
// This won't compile - Rust prevents use-after-free
fn main() {
    let data = vec![1, 2, 3];

    let reference = &data[0];
    drop(data); // Error: cannot drop while borrowed

    println!("{}", reference);
}
```

Why Use Rust?

Fearless Concurrency

C/C++ Concurrency Challenges:

- Data races are undefined behaviour
- Manual synchronisation prone to deadlocks
- Difficult to reason about thread safety

Why Use Rust?

Fearless Concurrency

C++ (Potential Data Race):

```
std::vector<int> data = {1, 2, 3, 4, 5}; // Global data

void worker(int id) {
    // Potential data race - multiple threads accessing shared data
    for (int i = 0; i < data.size(); ++i) {
        data[i] += id; // Race condition!
    }
}

void main() {
    std::thread t1(worker, 1), t2(worker, 2);
    t1.join(); t2.join();
}
```

Why Use Rust?

Fearless Concurrency

Rust's Approach:

- Data races prevented at compile time:
 - Offer tools to tackle safe transfer of ownership and shared access between threads.
- Fearless concurrency - if it compiles, it's thread-safe

Why Use Rust?

Fearless Concurrency

Rust (Compile-time Protection):

```
fn main() {  
    let data = vec![1, 2, 3, 4, 5];  
    let mut handles = vec![];  
  
    for i in 0..2 {  
        let handle = thread::spawn(move || {  
            for j in 0..data.len() {  
                data[j] += i;  
            }  
        });  
        handles.push(handle);  
    }  
  
    for handle in handles { handle.join().unwrap(); }  
}
```

Why Use Rust?

Fearless Concurrency

Rust (Working Example):

```
fn main() {  
    // wrap the Mutex to be able to share ownership across multiple threads  
    let data = Arc::new(Mutex::new(vec![1, 2, 3, 4, 5]));  
    let mut handles = vec![];  
  
    for i in 0..2 {  
        let data = Arc::clone(&data); // clone the reference  
        let handle = thread::spawn(move || {  
            let mut data = data.lock().unwrap(); // lock for a mutable reference  
            for j in 0..data.len() {  
                data[j] += i;  
            }  
        });  
        handles.push(handle);  
    }  
  
    for handle in handles { handle.join().unwrap(); }  
}
```

Why Use Rust?

Productivity

Dispelling the Myths

- “Rust is too hard to learn”
- “Fighting the *Borrow Checker* slows development”

Productivity Advantages

- Modern tooling
- Rich ecosystem
- Refactoring confidence
- Cross-platform by default

Why Use Rust?

Productivity: dispelling the myths

Myth: "Rust is Too Hard to Learn"

Reality check:

- Learning curve exists, but it's front-loaded: Who learned C++, will be able to learn Rust
- Once you understand [ownership](#), development accelerates.
- Compiler errors are educational, not obstructive
- C++ template errors vs Rust's helpful diagnostics

Why Use Rust?

Myth: "Rust is Too Hard to Learn"

C++ (Cryptic Template Error):

```
#include <map>
#include <string>
```

```
int main() {
    std::map<int, std::string> m;
    m[1] = "hello";
    const auto& cm = m;
    cm[2] = "world"; // Error, but what kind of error?
    return 0;
}
```

```
cryptic_template_error.cpp: In function 'int main()':
cryptic_template_error.cpp:8:9: error: passing 'const std::map<int, std::__cxx11::basic_string
<char> >' as 'this' argument discards qualifiers [-fpermissive]
    8 |         cm[2] = "world"; // Error, but what kind of error?
      |         ^
In file included from /opt/homebrew/Cellar/gcc/14.2.0_1/include/c++/14/map:63,
               from cryptic_template_error.cpp:1:
/opt/homebrew/Cellar/gcc/14.2.0_1/include/c++/14/bits/stl_map.h:524:7: note: in call to 'std
::map<_Key, _Tp, _Compare, _Alloc>::mapped_type& std::map<_Key, _Tp, _Compare, _Alloc>::operat
or[](key_type&&) [with _Key = int; _Tp = std::__cxx11::basic_string<char>; _Compare = std::les
s<int>; _Alloc = std::allocator<std::pair<const int, std::__cxx11::basic_string<char> > >; map
ped_type = std::__cxx11::basic_string<char>; key_type = int]'
   524 |         operator[](key_type&& __k)
      |         ~~~~~~
```

Why Use Rust?

Myth: "Rust is Too Hard to Learn"

Rust (Clear, Educational Error):

```
fn main() {  
    let x = 5;  
    x = 10; // Error with clear explanation  
}
```

```
error[E0384]: cannot assign twice to immutable variable `x`  
--> clear_error.rs:3:5  
2 |     let x = 5;  
   |         - first assignment to `x`  
3 |     x = 10; // Error with clear explanation  
   |     ^^^^^^ cannot assign twice to immutable variable  
  
help: consider making this binding mutable  
2 |     let mut x = 5;  
   |         +++  
  
error: aborting due to 1 previous error; 1 warning emitted
```


Why Use Rust?

Productivity: dispelling the myths

Myth: "Fighting the *Borrow Checker* slows development"

Reality check:

- Borrow checker catches bugs early that would be runtime issues in C/C++ (especially concurrency and use-after-free issues)
- Debugging time shifts from runtime to compile time
- Less time spent in debuggers, more time writing features
- Less fuzzing needed as well (no buffer overflows)

Why Use Rust?

Productivity: dispelling the myths

Bottom Line:

- At the end, it's all about ownership!
- Ownership is a wide topic, which needs to be dealt with, nothing specific to C++ to Rust.

Why Use Rust?

Productivity: Modern Tooling

Cargo Commands:

- Build: `cargo build`
- Run: `cargo run`
- Consistent formatting: `cargo fmt`
- Linting and best practices: `cargo clippy`
- Integrated testing: `cargo test`
- Code and API documentation: `cargo doc`
- Benchmarks: `cargo bench`

Why Use Rust?

Rich Ecosystem

Cargo Package Manager:

- Similar to [npm](#), [pip](#)
- Built-in dependency management, testing, documentation
- Over 100,000 packages (crates) available

Quality Libraries:

- [serde](#) for serialisation (better than most C++ JSON libraries)
- [tokio](#) for async runtime
- [clap](#) for command-line parsing
- [rayon](#) for data parallelism

Why Use Rust?

Productivity

Refactoring Confidence:

- If major refactoring compiles, it usually works
- Type system catches breaking changes
- No silent memory corruption to debug later

Cross-platform by Default:

- Write once, compile everywhere
- No platform-specific memory management gotchas

Why Use Rust?

Good Use Cases for Rust

- **Systems Programming:**
 - Device drivers, embedded systems, boot loaders, firmware
- **Network Services:**
 - Web servers, microservices, load balancers, Proxy servers
- **Performance-Critical Applications:**
 - Scientific computing, data science
 - Cryptocurrency and blockchain, High-frequency trading systems

Why Use Rust?

Real-World Adoption

Major Companies Using Rust:

- **Mozilla:** Firefox's CSS engine (Stylo)
- **Dropbox:** File storage backend
- **Microsoft:** Windows components, Azure services
- **Facebook:** Source control backend (Mononoke)
- **Cloudflare:** Network services
- **AWS:** lightweight virtualisation for serverless (Firecracker)
- **Discord:** Game chat backend

Why Use Rust?

Real-World Adoption

Why They Chose Rust:

- Memory safety reduces security vulnerabilities
- Without compromises on performance (equal to C/C++)
- Easier to maintain than equivalent C/C++ code
- Better concurrency story

Why Use Rust?

When NOT to Use Rust

Consider Alternatives When:

- Prototyping (unless you're already fluent in Rust)
- Small scripts or one-off utilities (I use mostly Python)
- Projects with massive existing C/C++ codebases
- Teams unwilling to invest in learning curve

Why Use Rust?

Migration Strategy

Gradual Adoption:

- Start with new self-contained components in Rust
- Use FFI to interface with existing C/C++ code
- Rewrite performance-critical modules first (other languages)
- Build developer expertise incrementally

Why Use Rust?

Key Takeaways for C/C++ Developers

- **Rust isn't just "Safer C++":** it's a different paradigm that prevents entire classes of bugs
- **The learning investment pays off quickly:** initial productivity dip followed by significant gains
- **Start small:** use Rust for new projects or performance-critical components
- **Leverage the ecosystem:** don't rebuild everything from scratch
- **Memory safety and performance:** you can have both without compromise

Questions to Consider:

- What are your biggest pain points with C/C++ development?
- How much time does your team spend debugging memory or concurrency related issues?
- How does debugging and testing affect your project timelines? (project budgets?)

Rust for C++ Developers

Part 2

Essential Rust Concepts for C++ Developers:

- Composition over Inheritance
- Move Semantics: Default vs Optional
- Ownership and Borrowing
- Error Handling: Result vs Exceptions
- Pattern Matching
- Generics
- Traits
- Macros

Composition over Inheritance

- No concept of classes with inheritance
- Rust uses structs to define custom data types, similar to classes but without inheritance
- Enums allow for defining a type that can be one of several different variants
- Traits in Rust are similar to interfaces or abstract base classes in other languages (more later)
- Rust encourages using composition (combining simpler types) to build complex types, rather than relying on inheritance

Move Semantics

Default vs Optional

C++ Copy by Default (Expensive):

```
#include <vector>
#include <string>
#include <iostream>

class Resource {
public:
    std::vector<int> data;
    std::string name;

    Resource(size_t size, const std::string& n)
        : data(size, 42), name(n) {}

    // Copy constructor - expensive!
    Resource(const Resource& other)
        : data(other.data), name(other.name) {}

    // Move constructor - efficient, but optional
    Resource(Resource&& other) noexcept
        : data(std::move(other.data)), name(std::move(other.name)) {}
};
```

Move Semantics

Default vs Optional

C++ Copy by Default (Expensive):

```
void cpp_move_semantics() {  
    Resource r1(1000000, "BigResource");  
  
    // Accidental copy - compiles but slow!  
    Resource r2 = r1; // Copy constructor called  
  
    // Explicit move - fast but easy to forget  
    Resource r3 = std::move(r1); // Move constructor called  
  
    // r1 is now in "valid but unspecified state" - dangerous!  
    // std::cout << r1.name; // Might work, might not  
  
    std::vector<Resource> vec;  
    Resource temp(500000, "TempResource");  
  
    // Without std::move, this copies!  
    vec.push_back(temp); // COPY - programmer forgot std::move  
  
    // With std::move, this moves  
    vec.push_back(std::move(temp)); // MOVE - but temp is now unusable  
}
```

Move Semantics

Default vs Optional

Rust Move by Default (Safe and Efficient):

```
#[derive(Debug)]
struct Resource {
    data: Vec<i32>,
    name: String,
}

impl Resource {
    fn new(size: usize, name: &str) -> Self {
        println!("Created {} with {} elements", name, size);
        Resource {
            data: vec![42; size],
            name: name.to_string(),
        }
    }
}
```


Move Semantics

Default vs Optional

Rust Move by Default (Safe and Efficient):

```
fn rust_move_semantics() {  
    let r1 = Resource::new(1000000, "BigResource");  
  
    // Move by default - always efficient!  
    let r2 = r1; // r1 is moved, not copied  
  
    // println!("{:?}", r1); // Compile error: value used after move  
    println!("{:?}", r2.name); // Only r2 is valid  
  
    let mut vec = Vec::new();  
    let temp = Resource::new(500000, "TempResource");  
  
    // Move by default - no std::move needed!  
    vec.push(temp); // temp is moved into vec  
  
    // println!("{:?}", temp); // Compile error: value used after move  
}
```

Move Semantics

Default vs Optional

Rust Explicit Copy:

```
// If you actually want to copy, you must be explicit
#[derive(Debug, Clone)] // Must implement Clone trait
struct CopyableResource {
    small_data: i32,
}

fn rust_copy() {
    let copyable = CopyableResource { small_data: 42 };
    let copied = copyable.clone(); // Explicit copy
    println!("Original: {:?}, Copy: {:?}", copyable, copied); // Both valid
}
```

Move Semantics

Key Differences

Aspect	C++	Rust
Default	Copy (expensive)	Move (efficient)
Move syntax	<code>std::move()</code> required	Automatic
Safety	Use-after-move compiles	Use-after-move = compiler error
Performance	Easy to accidentally copy	Always optimal by default
Explicit Copying	Default behaviour	Requires <code>Clone</code> trait

Move Semantics

Use-after-move

C++ Use-After-Move Bug:

```
void cpp_use_after_move_bug() {  
    std::string s1 = "Hello";  
    std::string s2 = std::move(s1);  
  
    // This compiles but is undefined behaviour!  
    std::cout << s1 << std::endl; // What gets printed?  
    s1.append(" World");           // Might crash, might work  
  
    // std::move doesn't actually move - it just casts to rvalue reference  
    // The actual move happens in the constructor/assignment  
}
```

Move Semantics

Use-after-move

Rust Compile-Time Protection:

```
fn rust_use_after_move_protection() {  
    let s1 = String::from("Hello");  
    let s2 = s1; // s1 moved to s2  
  
    // This won't compile - use after move detected!  
    // println!("{}", s1);  
  
    println!("{}", s2); // Only s2 is valid  
}
```

Move Semantics

Use-after-move

Rust Compile-Time Protection:

```
error[E0382]: borrow of moved value: `s1`
```

```
--> clear_error.rs:6:20
```

```
2 |     let s1 = String::from("Hello");  
   |           -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait  
3 |     let s2 = s1; // s1 moved to s2  
   |           -- value moved here  
..  
6 |     println!("{}", s1);  
   |                   ^^ value borrowed here after move
```

```
= note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of  
the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
```

```
help: consider cloning the value if the performance cost is acceptable
```

```
3 |     let s2 = s1.clone(); // s1 moved to s2  
   |           +++++++
```

```
error: aborting due to 1 previous error
```

Ownership Rules

1. Each value has exactly **one** owner
2. When the owner goes out of scope, the value is **dropped**
3. You can have **multiple immutable** references **OR** **one mutable** reference

Ownership Rules

C++ References

// C++ - References can dangle

```
int& dangerous_cpp() {  
    int local = 42;  
    return local; // Dangling reference!  
}
```

```
void cpp_aliasing() {  
    int x = 5;  
    int& ref1 = x;  
    int& ref2 = x;  
    ref1 = 10; // Both ref1 and ref2 see the change  
    ref2 = 20; // Potential for confusion  
}
```


Ownership Rules

Rust References

// Rust - This won't compile

```
fn safe_rust() -> &i32 {  
    let local = 42;  
    &local // Error: borrowed value does not live long enough  
}
```

```
fn rust_borrowing() {  
    let mut x = 5;  
    let ref1 = &x;    // Immutable borrow  
    let ref2 = &x;    // Multiple immutable borrows OK  
    // let ref3 = &mut x; // Error: cannot borrow as mutable  
    println!("{}", ref1, ref2);  
  
    let ref_mut = &mut x; // Now we can have mutable borrow  
    *ref_mut = 10;  
    // println!("{}", ref1); // Error: immutable borrow used here  
}
```

Error Handling: Result vs Exceptions

C++ Exceptions

```
std::string read_file(const std::string& filename) {  
    std::ifstream file(filename);  
    if (!file.is_open()) { throw std::runtime_error("Could not open file: " + filename); }  
  
    std::string content, line;  
    while (std::getline(file, line)) { content += line + "\n"; }  
    if (content.empty()) { throw std::runtime_error("File is empty"); }  
  
    return content;  
}  
  
void process() {  
    try {  
        auto content = read_file("data.txt");  
        std::cout << "File content: " << content << std::endl;  
    } catch (const std::exception& e) {  
        std::cerr << "Error: " << e.what() << std::endl; // What specific error occurred? Hard to tell.  
    }  
}
```

Error Handling: Result vs Exceptions

Rust Result Type

```
use std::{fs, io};

#[derive(Debug)]
enum ProcessError {
    IoError(io::Error),
    EmptyFile,
}

impl From<io::Error> for ProcessError {
    fn from(error: io::Error) -> Self {
        ProcessError::IoError(error)
    }
}

fn read_file(filename: &str) -> Result<String, ProcessError> {
    let content = fs::read_to_string(filename)?; // ? operator propagates errors

    if content.is_empty() {
        return Err(ProcessError::EmptyFile);
    }

    Ok(content)
}

fn process() {
    match read_file("data.txt") {
        Ok(content) => println!("File content: {}", content),
        Err(ProcessError::IoError(e)) => eprintln!("IO Error: {}", e),
        Err(ProcessError::EmptyFile) => eprintln!("Error: File is empty"),
    }
}
```

Pattern Matching

Traditional C++ Approach

```
enum class Shape { Circle, Rectangle, Triangle };

struct ShapeData {
    Shape type;
    double width, height, radius;
};

double calculate_area_cpp(const ShapeData& shape) {
    if (shape.type == Shape::Circle) {
        return 3.14159 * shape.radius * shape.radius;
    } else if (shape.type == Shape::Rectangle) {
        return shape.width * shape.height;
    } else if (shape.type == Shape::Triangle) {
        return 0.5 * shape.width * shape.height;
    }
    return 0.0; // Easy to forget cases
}
```

Pattern Matching

C++17 with std::variant

```
#include <variant>
```

```
using Shape = std::variant<  
    struct Circle { double radius; },  
    struct Rectangle { double width, height; },  
    struct Triangle { double base, height; }  
>;
```

```
double calculate_area_modern_cpp(const Shape& shape) {  
    return std::visit([](const auto& s) -> double {  
        if constexpr (std::is_same_v<std::decay_t<decltype(s)>, Circle>) {  
            return 3.14159 * s.radius * s.radius;  
        } else if constexpr (std::is_same_v<std::decay_t<decltype(s)>, Rectangle>) {  
            return s.width * s.height;  
        } else if constexpr (std::is_same_v<std::decay_t<decltype(s)>, Triangle>) {  
            return 0.5 * s.base * s.height;  
        }  
    }, shape);  
}
```

Pattern Matching

Rust Pattern Matching

```
enum Shape {  
    Circle { radius: f64 },  
    Rectangle { width: f64, height: f64 },  
    Triangle { base: f64, height: f64 },  
}  
  
fn calculate_area(shape: &Shape) -> f64 {  
    match shape {  
        Shape::Circle { radius } => 3.14159 * radius * radius,  
        Shape::Rectangle { width, height } => width * height,  
        Shape::Triangle { base, height } => 0.5 * base * height,  
        // Compiler ensures all cases are handled  
    }  
}
```

Generics

Simple Type Parameters

```
#include <vector>
```

```
template<typename T>  
class Container {  
private:  
    std::vector<T> data;  
public:  
    void push(const T& item) { data.push_back(item); }  
    T& get(size_t index) { return data[index]; }  
    size_t size() const { return data.size(); }  
};
```

```
template<typename T>  
T max_value(const T& a, const T& b) {  
    return (a > b) ? a : b;  
}
```

Generics

Simple Type Parameters

```
#include <string>

void cpp_generics_example() {
    Container<int> int_container;
    Container<std::string> string_container;

    int_container.push(42);
    string_container.push("hello");

    // Type deduction works here
    auto result1 = max_value(10, 20);
    auto result2 = max_value(3.14, 2.71);

    // But explicit types sometimes needed
    auto result3 = max_value<double>(10, 3.14); // Mixed types
}
```


Generics

Simple Type Parameters

```
struct Container<T> {  
    data: Vec<T>,  
}  
  
impl<T> Container<T> {  
    fn new() -> Self {  
        Container { data: Vec::new() }  
    }  
  
    fn push(&mut self, item: T) {  
        self.data.push(item);  
    }  
  
    fn get(&self, index: usize) -> Option<&T> {  
        self.data.get(index)  
    }  
  
    fn size(&self) -> usize {  
        self.data.len()  
    }  
}  
  
// Generic function - type inference works well  
fn max_value<T: PartialOrd>(a: T, b: T) -> T {  
    if a > b { a } else { b }  
}
```

Generics

Simple Type Parameters

```
fn rust_generics_example() {  
    let mut int_container: Container<i32> = Container::new();  
    let mut string_container: Container<String> = Container::new();  
  
    int_container.push(42);  
    string_container.push("hello".to_string());  
  
    // Type inference usually works  
    let result1 = max_value(10, 20);  
    let result2 = max_value(3.14, 2.71);  
  
    // Explicit types when needed  
    let result3 = max_value(10i32, 20i32);  
}
```

Generics

std::optional vs Generic Enums

```
#include <optional>
#include <iostream>

// std::optional - built-in, but limited
void cpp_optional_example() {
    std::optional<int> maybe_value = 42;
    std::optional<int> no_value;

    if (maybe_value.has_value()) {
        std::cout << "Value: " << maybe_value.value() << std::endl;
    }

    if (!no_value.has_value()) {
        std::cout << "No value" << std::endl;
    }
}
```

Generics

std::optional vs Generic Enums

```
// Option<T> - built into the language
fn rust_option_example() {
    let maybe_value: Option<i32> = Some(42);
    let no_value: Option<i32> = None;

    match maybe_value {
        Some(value) => println!("Value: {}", value),
        None => println!("No value"),
    }

    // Or use if let for single case
    if let Some(value) = maybe_value {
        println!("Value: {}", value);
    }
}
```

Traits

Interface Definition and Implementation

```
trait Drawable {  
    fn draw(&self);  
}  
  
trait Resizable {  
    fn resize(&mut self, factor: f64);  
}  
  
struct Circle {  
    radius: f64,  
}  
  
impl Drawable for Circle {  
    fn draw(&self) {  
        println!("Drawing circle with radius {}", self.radius);  
    }  
}  
  
impl Resizable for Circle {  
    fn resize(&mut self, factor: f64) {  
        self.radius *= factor;  
    }  
}
```

Traits

Trait bounds

// Generic function with trait bounds

```
fn render<T: Drawable>(shape: &T) {  
    shape.draw();  
}
```

// Multiple trait bounds

```
fn resize_and_render<T: Drawable + Resizable>(shape: &mut T, factor: f64) {  
    shape.resize(factor);  
    shape.draw();  
}
```

// Alternative syntax

```
fn resize_and_render_alt<T>(shape: &mut T, factor: f64)  
where  
    T: Drawable + Resizable  
{  
    shape.resize(factor);  
    shape.draw();  
}
```

Macros

C++ Preprocessor vs Rust Macros:

- Text replacement vs AST manipulation:
 - Rust macros work on syntax trees, not raw text
- Type safety:
 - Rust macros are checked at compile time
- Lots of additional features:
 - Pattern Matching: Rust macros can match different syntax patterns
 - Procedural Macros: Code generation (like `#[derive]`)
 - Repetition Patterns: Clean syntax for variadic-like functionality

Macros

Declarative Macros

```
macro_rules! vec {  
  ( $( $x:expr ),* ) => {  
    {  
      let mut temp_vec = Vec::new();  
      $(  
        temp_vec.push($x);  
      )*  
      temp_vec  
    }  
  };  
}
```

```
fn main() {  
  let v = vec![1, 2, 3];  
  println!("{:?}", v);  
}
```


Macros

Procedural Macros

// Using serde - automatic serialisation

```
use serde::{Serialize, Deserialize};
```

```
#[derive(Debug, Serialize, Deserialize)] // Procedural macros
```

```
struct Person {  
    name: String,  
    age: u32,  
    email: String,  
}
```

```
#[derive(Debug, Serialize, Deserialize)]
```

```
struct Company {  
    name: String,  
    employees: Vec<Person>,  
    founded: u32,  
}
```

Macros

Procedural Macros

```
fn procedural_macro_example() {  
    let person = Person {  
        name: "Alice".to_string(),  
        age: 30,  
        email: "alice@example.com".to_string(),  
    };  
  
    // Serialisation code generated automatically  
    let json = serde_json::to_string(&person).unwrap();  
    println!("JSON: {}", json);  
  
    // Deserialisation also works  
    let parsed: Person = serde_json::from_str(&json).unwrap();  
    println!("Parsed: {:?}", parsed);  
}
```

Macros

Procedural Macros

```
#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Hash, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn derive_example() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 1, y: 2 };
    let p3 = Point { x: 2, y: 1 };

    // All of these work automatically:
    println!("{:?}", p1);           // Debug
    println!("{}", p1 == p2);      // PartialEq
    println!("{}", p1 < p3);      // PartialOrd

    let cloned = p1.clone();       // Clone

    use std::collections::HashMap;
    let mut map = HashMap::new();
    map.insert(p1, "point1");      // Hash
}
```

Macros

Advanced Use

```
use tokio::sync::oneshot;

#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::channel();
    let (tx2, rx2) = oneshot::channel();

    tokio::spawn(async {
        let _ = tx1.send("one");
    });

    tokio::spawn(async {
        let _ = tx2.send("two");
    });

    tokio::select! {
        val = rx1 => {
            println!("rx1 completed first with {:?}", val);
        }
        val = rx2 => {
            println!("rx2 completed first with {:?}", val);
        }
    }
}
```

Macros

Advanced Use

Macros can define their own DSL **verified at compile time**:

```
let countries = sqlx::query!(  
    "  
    SELECT country, COUNT(*) as count  
    FROM users  
    GROUP BY country  
    WHERE organization = ?  
    ",  
    organization  
)  
    .fetch_all(&pool) // -> Vec<{ country: String, count: i64 }>  
    .await?;  
  
// countries[0].country  
// countries[0].count
```

Rust for C++ Developers

Part 3

Getting Started: Your First Rust Project

- Project Structure
- Basic Commands
- Next Steps

Your First Rust Project

Project Structure

Let's just start and create a new project:

```
cargo new in-space  
cd in-space
```

Now let's add some dependencies:

```
cargo add anyhow  
cargo add reqwest --features json  
cargo add serde --features derive  
cargo add tokio --features full
```

Your First Rust Project

Project Structure

A git repository with the following files has been created:

- [Cargo.toml](#): This is Cargo's file to define dependencies and other project settings.
- [src/main.rs](#): First Rust source code file with a ready-made "Hello World".

In [Cargo.toml](#), Dependencies are listed like this:

```
[dependencies]
anyhow = "1.0"
reqwest = { version = "0.12", features = ["json"] }
serde = { version = "1.0", features = ["derive"] }
tokio = { version = "1", features = ["full"] }
```


Your First Rust Project

Basic Commands

Build (like make)

cargo build # Debug build

cargo build --release # Release build (like -O3)

Run (like ./my_project)

cargo run

cargo run --release

Check without building (fast feedback)

cargo check

Format code (like clang-format)

cargo fmt

Lint (like clang-tidy)

cargo clippy

Generate documentation

cargo doc --open

Your First Rust Project

Do Something

```
use anyhow::{Result, bail};
use serde::{Deserialize, Serialize};

#[derive(Debug, Deserialize, Serialize)]
struct Person {
    name: String,
    craft: String,
}

#[derive(Debug, Deserialize)]
struct SpaceResponse {
    number: u32,
    people: Vec<Person>,
}
```

Your First Rust Project

Do Something

```
impl SpaceResponse {
    async fn fetch() -> Result<Self> {
        // Make HTTP request to the Open Notify API
        let url = "http://api.open-notify.org/astros.json";
        let response = reqwest::get(url).await?;

        // Check if request was successful
        if !response.status().is_success() {
            bail!("Failed to fetch data: {}", response.status());
        }

        // Parse JSON response
        let space_data: SpaceResponse = response.json().await?;

        // Return data
        Ok(space_data)
    }
}
```

Your First Rust Project

Do Something

```
#[tokio::main]
async fn main() -> Result<()> {
    println!("🚀 Querying people currently in space...\n");
    let space_data = SpaceResponse::fetch().await?;

    // Display results
    println!("Total people in space: {}\n", space_data.number);

    if space_data.people.is_empty() {
        println!("No one is currently in space.");
    } else {
        println!("People currently in space:");
        for (i, person) in space_data.people.iter().enumerate() {
            println!("{}", i, (aboard {})", i + 1, person.name, person.craft);
        }
    }

    println!("\n✨ Data provided by Open Notify API");
    Ok(())
}
```

Your First Rust Project

Implement a new feature

Let's use a HashMap:

```
use std::collections::HashMap;
```

And group by spacecraft:

```
println!("Grouped by spacecraft:");
let mut craft_groups: HashMap<&String, Vec<&String>> = HashMap::new();

for person in &space_data.people {
    craft_groups
        .entry(&person.craft)
        .or_default()
        .push(&person.name);
}

for (craft, crew) in craft_groups {
    println!("\n{} ({} crew members):", craft, crew.len());
    for name in crew {
        println!("  • {}", name);
    }
}
```

Your First Project

Next Steps

Links:

- Start with the Rust Book: <https://doc.rust-lang.org/book/>
- Interactive exercises: <https://rustlings.cool/>
- Cheat Sheet: <https://cheats.rs>
- [C++ to Rust Phrasebook](#)
- C++ vs Rust Type Safety: [Matt Godbolt sold me on Rust \(by showing me C++\)](#)

Community:

- [r/rust subreddit](#)
- [Rust Users Forum](#)
- [Rust Discord server](#)