C++ to Rust

Follow this series for more bite-sized comparisons between C++ and Rust!

HOW TO DO OOP IN RUST?

NO CLASSES & NO CLASSICAL INHERITANCE

Rust does not have classes or classical inheritance, but it does support object-oriented programming (OOP) through composition, traits, and dynamic dispatch. It encourages an OOP style that avoids some of the pitfalls of inheritance-heavy design.

C++ to Rust

Follow this series for more bite-sized comparisons between C++ and Rust!

🔑 Key Object-Oriented Programming Features in Rust

- **Encapsulation**
  - ✅ **Supported via struct, impl, and module-level privacy (pub, private by default)**
- **Polymorphism**
  - ✅ **Achieved through traits**
  - 🧰 **Use generics for static dispatch**
  - 🧰 **Use dyn Trait for dynamic dispatch**
- **Inheritance**
  - ❌ **Not supported in the classical sense**
  - ✅ **Replaced by composition and trait inheritance**
- **Dynamic Dispatch**
  - ✅ **Supported using trait objects (Box<dyn Trait>, &dyn Trait)**
  - 🧠 **Allows runtime polymorphism without virtual tables**
- **Interfaces**
  - ✅ **Traits serve as Rust's equivalent to interfaces**
  - **Traits can include default method implementations**

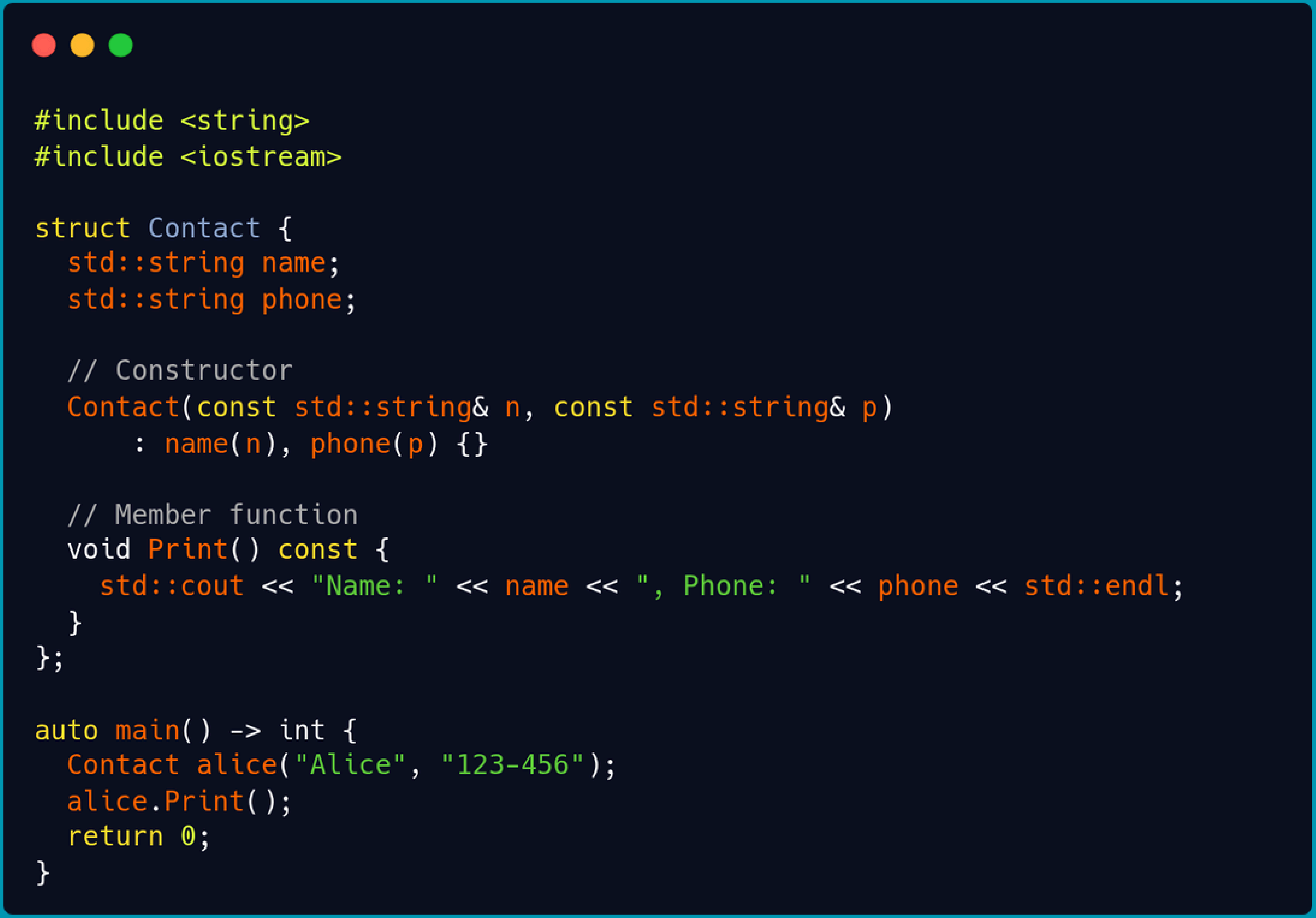
🔍 Let's give a look into Encapsulation:

→

C++ to Rust

Follow this series for more bite-sized comparisons between C++ and Rust!

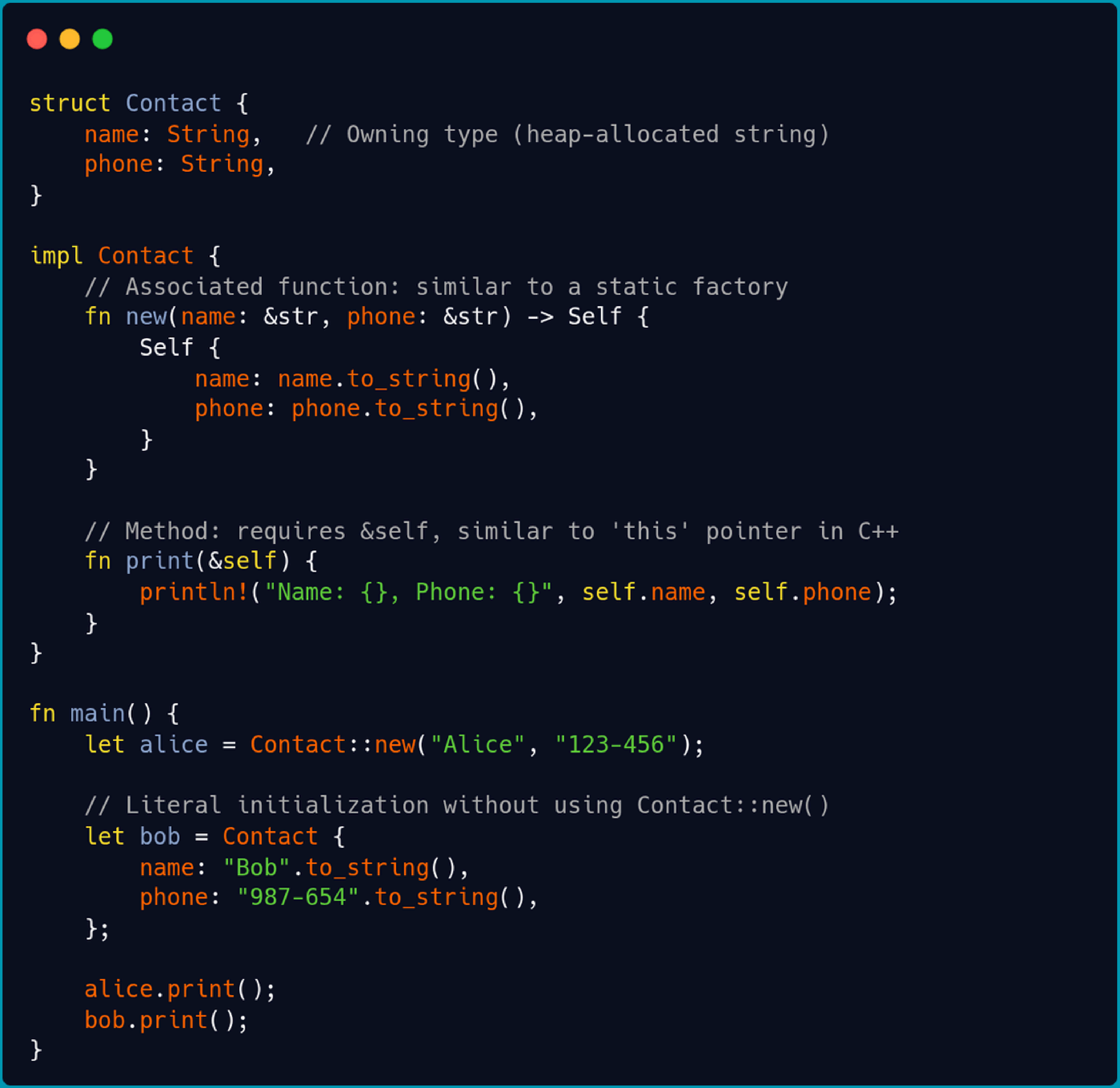
✅ C++ — struct:



```cpp
#include <string>
#include <iostream>

struct Contact {
  std::string name;
  std::string phone;

  // Constructor
  Contact(const std::string& n, const std::string& p)
      : name(n), phone(p) {}

  // Member function
  void Print() const {
    std::cout << "Name: " << name << ", Phone: " << phone << std::endl;
  }
};

auto main() -> int {
  Contact alice("Alice", "123-456");
  alice.Print();
  return 0;
}
```

✅ struct can have fields, constructors, and methods
✅ Member functions can be const
✅ this is used implicitly
✅ Fields are public by default

✅ Rust — struct:



```rust
struct Contact {
    name: String,    // Owning type (heap-allocated string)
    phone: String,
}

impl Contact {
    // Associated function: similar to a static factory
    fn new(name: &str, phone: &str) -> Self {
        Self {
            name: name.to_string(),
            phone: phone.to_string(),
        }
    }

    // Method: requires &self, similar to 'this' pointer in C++
    fn print(&self) {
        println!("Name: {}, Phone: {}", self.name, self.phone);
    }
}

fn main() {
    let alice = Contact::new("Alice", "123-456");

    // Literal initialization without using Contact::new()
    let bob = Contact {
        name: "Bob".to_string(),
        phone: "987-654".to_string(),
    };

    alice.print();
    bob.print();
}
```

✅ No built-in constructors, and new() is just a convention
— not built into the language
✅ Literal initialization is flexible, similar to aggregate init in C++

C++ to Rust

Follow this series for more bite-sized comparisons between C++ and Rust!

✅ **What to notice:**

⚖️ What's the C++ Counterpart?

- ✅ Rust struct, no 'class' keyword in Rust
  - → Equivalent to C++ struct or class
- ✅ Private fields by default in Rust
  - → Like private section in C++
- ✅ Associated function (fn new)
  - → Similar to a static member function in C++
- ✅ impl block in Rust
  - → Comparable to grouping member functions inside a C++ class
- ✅ Self in Rust
  - → Refers to the struct type, like using the class name or this in C++
- ✅ &self in method signature
  - → Similar to const Type* 'this' in a C++ const member function

🔍You may define a new() associated function in Rust when:

- ○ You want to pre-process inputs (e.g. parse, convert types)
- ○ You want to hide initialization logic from the user
- ○ You need to apply defaults or perform validation