Rust Traits in Style for C++

How We Unlocked Their Big Benefits for Users and Much More!

EDUARDO MADRID

CppCon

The C++ Conference

2025

September 13 - 19

Rust Traits in Style for C++

How We Unlocked Their Big Benefits for Users and Much More!

EDUARDO MADRID

Cppcon
The C++ Conference

2025
September 13 - 19

Who Are We?
What will we do?

Me

- Lead of the Open Source zoo libraries
  - SWAR
  - Type Erasure
- Prior experience as a Team/Tech lead at Snap, and Automated/High Frequency Trading.
- My theme is doing things that are amazing and can only be done in C++
- Ah, I am Rust-skeptic

RP == Runtime Polymorphism

RP Series

- CPPCon 2020: “Not Leaving Performance On The Jump Table”: Performance
- C++ London Users Group: “Type Erasing the Pains of RP”: Conceptual introduction to the zoo framework for Type Erasure
- C++ Now 2021, with Phil Nash, “Polymorphism `A la carte”: Objective C, “Object Oriented Considered Harmful”
- C++ Now 2024, by Fedor Pikus, “Type Erasure Demystified”: A very different explanation of a few performance techniques used in the zoo framework
- C++ Online 2025: “External Polymorphism And Type Erasure”: Covering both as a “dance” of design patterns that is very fruitful
- C++ Now 2025: “RP with Freedom And Performance”: More advanced capabilities, unique to zoo.
- C++ Now 2025: Rust Traits In C++: mapping of Rust Traits to C++

Today

- We will do something hard to believe:

- Instead of the feature for RP in the language, `virtual` and inheritance,
- we will use user-code (a type erasure framework),
- that allows us to synthesize benefits from the Rust feature to do RP, Rust Traits,
- still in C++.
- If we survive the challenge of understanding all of necessary things really well,
- we get “best of all worlds”:

1. Better performance, object code size
2. More modeling power

How’s this possible?

- Because of C++:
  - Performance preserving/increasing abstraction
  - Because of the wisdom of not committing to “the language designers know best”
  - Because of **our** resistance for outstanding complexity
- I hope this shows you the value of these C++ conferences:
  - We gather to help each other do what seems impossible!

Let’s try to see if we can understand these things as if we were language designers

What’s our focus?

- Out of scope:
  - The compile-time powers of Rust Traits, monomorphization, etc.
  - RP in the style of Rust Enums, that resemble `std::variant`.
  - Comprehensive coverage of features
  - Rust to/from C++ binding
- Guidance: how can you express Rust features in C++

What Is the “style” of RP of Rust Traits?

RP == Runtime Polymorphism

What is RP, why is it good?

The Issue Is Quite Simple:

Identify the essentials to not have to deal with irrelevant details

Gather the essentials into something common that is reused in all particular cases

N implementations,
M uses:
N*M pieces of code

1 Interface
N implementations
M uses:
1 + N + M pieces of code

If we get it right, huge win!

Abstraction!

Also, conceptual leverage

Runtime?
The actual implementation is not known when compiling

Polymorphism?
That we can **substitute** a particular implementation for any other (poly) and things will work

That’s the “Liskov Substitution Principle”, named after the inventor of the concept, Barbara Liskov

In the Style Of Traits?
I will tell you a bit later

Part 1

Type: Interface

Subtype: Implementation

WARNING: in most languages, including C++, *subtyping* is conflated with *subclassing*

Subclassing (base class and `virtual` overrides): what we won’t be doing

Let us serialize

```
struct ISerialize {
    virtual std::ostream &
    serialize(std::ostream &) const = 0;

    virtual std::size_t
    length() const = 0;

    virtual ~ISerialize() {}
};
```

…an integer

But `int` does not inherit from `ISerialize`...

Then we wrap our integers into a wrapper *type* that inherits from `ISerialize`

We’re cool, we will write the integer wrapper as a template, while we are at it

```cpp
template<typename T>
struct SerializeWrapper: ISerialize {
    T value_;

    virtual std::ostream &serialize(std::ostream &to) const override {
        return
            to << g_registry.id<T>() << ':' << value_;
    }

    virtual std::size_t length() const override {
        std::ostringstream temporary;
        serialize(temporary);
        return temporary.str().length();
    }
};
```

Serialization as subclassing:

- Pre-existing types (`int`, `std::string`, etc.) cannot inherit from this interface. One must wrap them: that’s why we do `SerializeWrapper`.
- If you make your type in the “`ISerialize`” hierarchy, then, you have a problem if later:
  - another application wants its own serialization mechanism
  - You want your type to participate in *any other* subclassing.
- I like that Rust Traits don’t have these problems!

The tragedy is that subtyping-as-subclassing is the most popular and the worst way for doing subtyping

Rust knows better!

```rust
// Definition of the subtyping relation: you can invoke
// serialize(object, output) on anything that implements this trait.
pub trait Serialize {
    fn serialize(&self, to: &mut dyn io::Write) -> io::Result<()>;
}

// ...

impl Serialize for i32 {
    fn serialize(&self, to: &mut dyn io::Write) -> io::Result<()> {
        write!(to, "{}", self)
    }
}
```

```rust
use std::io;

// Definition of the subtyping relation: you can invoke
// serialize(object, output) on anything that implements this trait.
pub trait Serialize {
    fn serialize(&self, to: &mut dyn io::Write) -> io::Result<()>;
}

// A user-defined type.
pub struct Point {
    x: f64,
    y: f64,
}

// Binding Point to the Serialize trait, allowing it to be used polymorphically.
impl Serialize for Point {
    fn serialize(&self, to: &mut dyn io::Write) -> io::Result<()> {
        write!(to, "({}, {})", self.x, self.y)
    }
}

// We can't do this in C++: primitive types like int can't implement interfaces,
// thus we must use wrappers.
// In Rust, we can just implement the trait directly for i32.
impl Serialize for i32 {
    fn serialize(&self, to: &mut dyn io::Write) -> io::Result<()> {
        write!(to, "{}", self)
    }
}
```

Rust Traits

- An opt-in mechanism for types to participate in subtyping relations
- Not related to inheritance (which is an structural property in C++), it does not need base classes.
- There are more things related to compilation time, but our focus is exclusively on runtime polymorphism, so, compile-time aspects like trait bounds and generics are out of scope.
- Like Python's “Duck Typing” interfaces, but you have to provide the binding between the objects and the interfaces.

Now, a C++ world of hurt:

```
struct UserType {
    // ...
    ISerialize instanceMemberVariable_;
    // ...
};
```

Slicing!

Inheritance Intrusiveness

- Forces an structural property just for the sake of RP

- Requires *referential semantics* (the class has a pointer (smart or not) or a reference) which is not cool:
  1. Allocation
  2. Indirection
  3. Lifetime Management
  4. Incentive to share state
  5. Disables local reasoning

  …
  Error: numbered list overflow

- Error: too much temptation

Subclassing Pains

1. Intrusive: we need to wrap perfectly good types to put them in a hierarchy. Busy work. Typical example: wrapping integers in `ISerialize` wrappers.
   Using the example of serialization:

   1. Frequently the need to serialize a type is identified long after writing the type, then, we have to retrofit.

   2. Even then, different applications would want to serialize in a different way, this would force the wrappers to change to inherit from multiple interfaces: an avalanche of work.

2. Take it or leave it: A feature of the language you can't finesse.

3. Further problems:

Rethink RP in C++ by Nicolai Josuttis,
C++ On Sea 2025.
YouTube video premieres in 14 days

```rust
pub struct UserType {
    // `dyn Serialize` = a *trait object* (fat ptr: data + vtable).
    // It's unsized, so it must live behind a pointer. `Box` = owner.
    member: Box<dyn Serialize>,
}
```

```rust
impl UserType {
    // Accept any concrete `T` that implements the trait.
   pub fn new<T>(x: T) -> Self
    where
        // `+ 'static` is needed because `member` has no lifetime
        // parameter. It guarantees `T` doesn't borrow short-lived data.
        T: Serialize + 'static,
    {
        // `Box::new(x)` allocates, then coerces to `Box<dyn Serialize>`
        // by building a vtable for `T` (dynamic dispatch enabled).
        Self { member: Box::new(x) }
    }

    // Replace the stored value with another serializable thing.
    pub fn set<T>(&mut self, x: T)
    where
        T: Serialize + 'static,
    {
        self.member = Box::new(x);
    }
}
```

```rust
// Example use:
let mut buf = Vec::new();
let mut u = UserType::new(Point { x: 1.0, y: 2.0 });
u.serialize_to(&mut buf)?;                   // writes "(1, 2)"
u.set(7_i32);                                // swap to a different model
u.serialize_to(&mut buf)?;                   // writes "7"
```

“In the Style of…”

- There is a huge ideological gap between Rust and C++, things don’t just translate one to one:
    - Rust is very serious about mutability, and gates it by “borrowing mutable references”:
        - There may be only one `&mut T` exclusive-or many `&T`.
            - That’s enforced at compilation by the *borrow checker*.
- In C++ we do whatever the hell we want
- Rust has `unsafe` for “trust me, I know what I am doing” code
- In Rust moves are the default and implicit, copies are opt-in

Rust's error prevention

- No implicit copy on pass/assign for owning types.
- Clear split: Copy (for trivial types), Clone (for types that require explicit work)
- That, and the borrowing rules, make it so that the compiler prevents use-after-move and aliasing bugs by construction!

Rust’s error prevention

- No implicit copy on pass/assign for owning types.
- Clear split: Copy (for trivial types), Clone (for types that require explicit work)
- That, and the borrowing rules, make it so that the compiler prevents use-after-move and aliasing bugs by construction!

The One Major Limitation

- I’ve not seen a way to express the rules and borrow checker, so, we can’t benefit from Rust’s error prevention:
  - We can keep doing as we want,
  - or we can learn from the *style* of Rust and constrain ourselves as a discipline.

By the way, the struct “UserType” is move-only

Part 2

Type Erasure

- We can empower any type to have RP using the "External Polymorphism" design pattern.

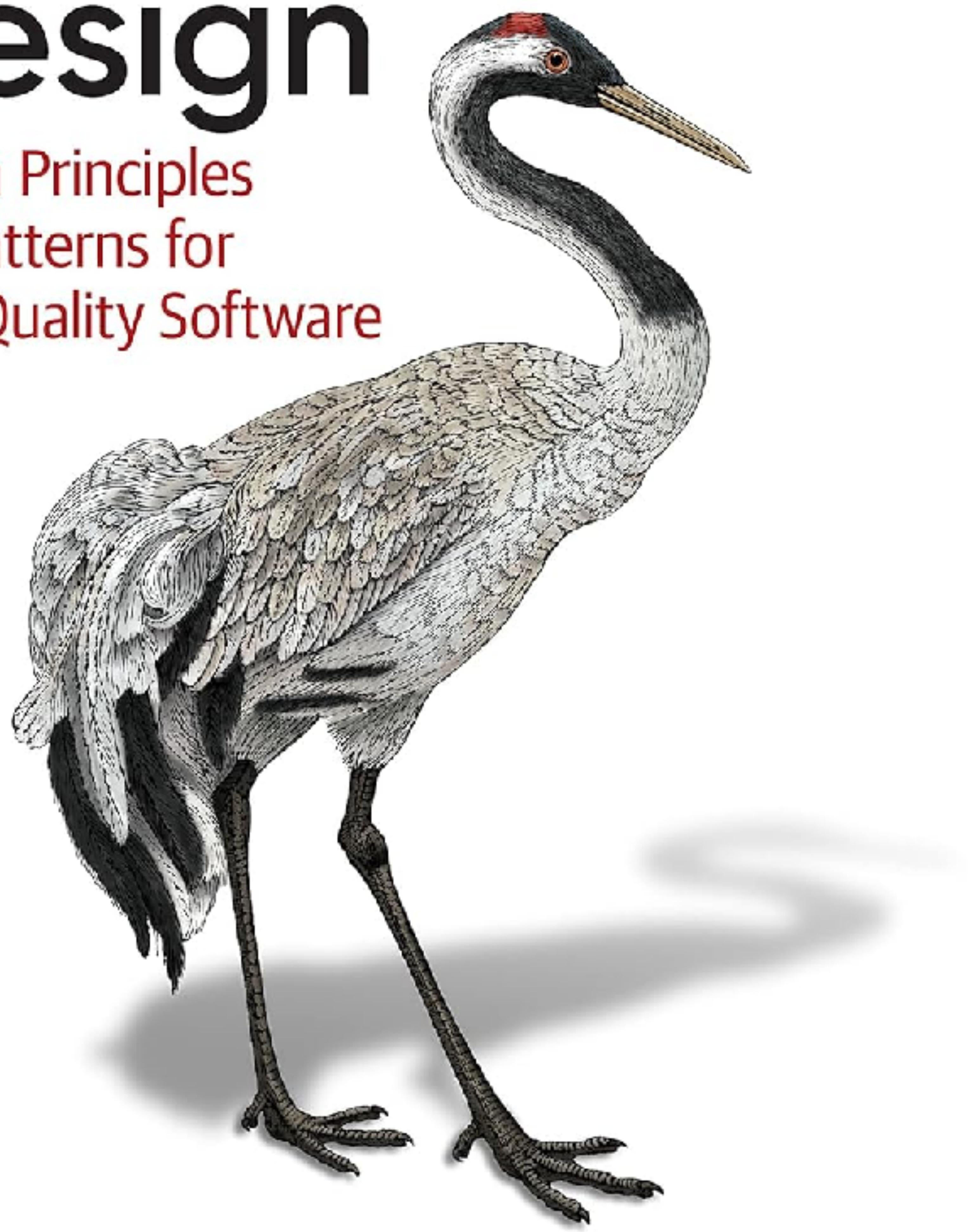
- If we do external polymorphism of ownership (destruction, moving, potentially copying), then we may assume total control of the object, and I call that Type Erasure, the apparent contradiction of "internal external polymorphism" (C++ Online and C++ Now 2025 for more)

- The best description of External Polymorphism I've ever seen is in "C++ Software Design" by Klaus Iglberger

O'REILLY®

C++ Software Design

Design Principles and Patterns for High-Quality Software



Klaus Iglberger

Expression of Traits in C++

- To synthesize RP in a different way, we will use C++’s excellent compile time polymorphism mechanisms, to generate what the compiler does for inheritance and `virtual`: Type Erasure.
- We need:
  - How to build a virtual table
  - “Fat Pointers” or “Type Erased Containers” that manage a value and link them to their virtual table
    - Including “dynamic dispatch”
  - A mechanism for the users to be able to extend type erasure containers with the equivalent of Rust Traits

Zoo Type Erasure

- Uncompromising about performance (latency, object code size, others)
    - It overwhelms because of subtlety and customization possibilities
- We (including Jamie Pond) are doing “Type Erasure, but Auditable”, basically, the same but streamlined, in the namespace `zoo::tea`
    - Let us know if you’d like to collaborate
    - Will be the ongoing basis for our RP work

How the user uses their “Trait”

```
using Policy = // This is the configuration of the container
    zoo::Policy< // This is the policy builder
        void *, // <- The local space
        zoo::Destroy, zoo::Move, zoo::Copy,// basic affordances
        SerializeAffordance // <- The user “trait”!
    >;
using Serializable = zoo::AnyContainer<Policy>;

auto useSerializeTrait(std::OSTREAM &to, const Serializable &a)
{
    return a.ser(to);
}

Serializable make(int i) { return i; }
```

```cpp
struct SerializeAffordance {
    struct VTableEntry {
        std::OSTREAM *(*serialize_impl)(std::OSTREAM &, const void *);
        std::size_t (*length_impl)(const void *);
    };

    template<typename ConcreteValueManager>
    constexpr static inline VTableEntry Operation = {
        [](std::OSTREAM &to, const void *cvm) {
            auto asConcreteValueManagerPtr =
                static_cast<const ConcreteValueManager *>(const_cast<void *>(cvm));
            using OriginalType = typename ConcreteValueManager::ManagedType;
            const OriginalType *value = asConcreteValueManagerPtr->value();
            return impl::howToSerializeT(to, *value);
        },
        [](const void *cvm) {
            std::OSTRINGSTREAM temporary;
            Operation<ConcreteValueManager>.serialize_impl(temporary, cvm);
            return temporary.str().length();
        }
    };

    template<typename>
    constexpr static inline VTableEntry Default = {
        [](std::OSTREAM &out, const void *defaultValueManager) {
            // possibly throw?
            return out;
        },
        [](const void *defaultValueManager) -> std::size_t {
            // some implementation
            return 0;
        }
    };

    template<typename UserInterfaceContainer> struct UserAffordance {
        std::OSTREAM &ser(std::OSTREAM &out) const {
            auto crtp =
                const_cast<UserInterfaceContainer *>(
                    static_cast<const UserInterfaceContainer *>(this)
                );
            auto baseValueManagerPtr = crtp->container();
            auto implementation =
                baseValueManagerPtr->
                    template vTable<SerializeAffordance>()->
                        serialize_impl;
            return *implementation(out, baseValueManagerPtr);

        }
    };

    template<typename> struct Mixin {};
};
```

Dynamic Dispatch

- We will use the virtual table mechanism:
  - An object of a type that is `Serializable` will have a virtual table with pointers to the implementations of the functions for `serialize` and `length`.
  - That's not the only things needed: but also destruction, moving and copying.
  - All of these runtime-polymorphic behaviors will be activated by invoking the function pointers in the virtual table
- Then, the objects must have all of their state *and* a pointer to the virtual table
  - This is taken care of by the type erasure framework

Making The Virtual Table

- At https://github.com/thecppzoo/zoo/blob/master/inc/zoo/Any/VTablePolicy.h#L245C1-L253C1

```
struct SerializeAffordance {
    struct VTableEntry {
        std::OSTREAM *(*serialize_impl)(std::OSTREAM &, const void *);
        std::size_t (*length_impl)(const void *);
    };
```

Zoo Virtual Table Building

```
struct VTableEntry {
    std::OSTREAM *(*serialize_impl)(std::OSTREAM &, const void *);
    std::size_t (*length_impl)(const void *);
};
```

```
template<typename HoldingModel, typename... AffordanceSpecifications>
struct GenericPolicy {
    struct VTable: AffordanceSpecifications::VTableEntry... {
        template<typename Affordance>
        const typename Affordance::VTableEntry *upcast() const noexcept {
            return this;
        }
    };
```

```
struct VTable: AffordanceSpecifications::VTableEntry... {
```

Implementing Virtual Tables

```
template<typename ConcreteValueManager>
constexpr static inline VTableEntry Operation = {
    [](std::OSTREAM &to, const void *cvm) {
        auto asConcreteValueManagerPtr =
            static_cast<const ConcreteValueManager *>(const_cast<void *>(cvm));
        using OriginalType = typename ConcreteValueManager::ManagedType;
        const OriginalType *value = asConcreteValueManagerPtr->value();
        return impl::howToSerializeT(to, *value);
    },
    [](const void *cvm) {
        std::OSTRINGSTREAM temporary;
        Operation<ConcreteValueManager>.serialize_impl(temporary, cvm);
        return temporary.str().length();
    }
};
```

We have runtime polymorphism!

Binding Dynamic Dispatch to the user interface

```
template<typename UserInterfaceContainer>
struct UserAffordance {
    std::OSTREAM &ser(std::OSTREAM &out) const {
        auto crtp =
            const_cast<UserInterfaceContainer *>(
                static_cast<const UserInterfaceContainer *>(this)
            );
        auto baseValueManagerPtr = crtp->container();
        auto implementation =
            baseValueManagerPtr->
                template vTable<SerializeAffordance>()->
                    serialize_impl;
        return *implementation(out, baseValueManagerPtr);

    }
};
```

And… usage

```
using Policy =
    zoo::Policy<
        void *, zoo::Destroy, zoo::Move, zoo::Copy,
        SerializeAffordance
>;
using Serializable = zoo::AnyContainer<Policy>;

auto useSerializeTrait(std::OSTREAM &to, const Serializable &a)
{
    return a.ser(to);
}

Serializable make(int i) { return i; }
```

Live in the Compiler Explorer

Back In Planet Earth

- We do the same things that Rust Traits do:
  - Subtyping without subclassing
  - Opt in
  - We are also using overloads/templates to implement the trait for all types that satisfy a property, for example “being insertable to a stream, or that `os << thingy` will work.
  - Rust allows you to do this:

```rust
use std::fmt::Display;
use std::io::{self, Write};

trait Serialize {
    fn serialize(&self, to: &mut dyn Write) -> io::Result<()>;
}

// Blanket implementation for all types that implement Display
impl<T: Display> Serialize for T {
    fn serialize(&self, to: &mut dyn Write) -> io::Result<()> {
        write!(to, "{}", self)
    }
}
```

Recap

- We can capture runtime polymorphism in the style of Rust Traits:
  - Opt-in subtyping without inheritance or subclassing
- We have advantages, for example, we can avoid the indirection of Box in many cases:
    - In Rust, if you want to keep a member variable runtime-polymorphic, it must always live behind a pointer, such as Box<dyn Trait>, &[T], &dyn Trait, Arc<dyn Trait>
    - Zoo Type Erasure not only gives you value semantics if you want, but a universe of possibilities much further, for example, you can choose to use a larger local buffer for better performance in some cases.
  - We remain in C++!

Conclusion

- C++ is that powerful

END!