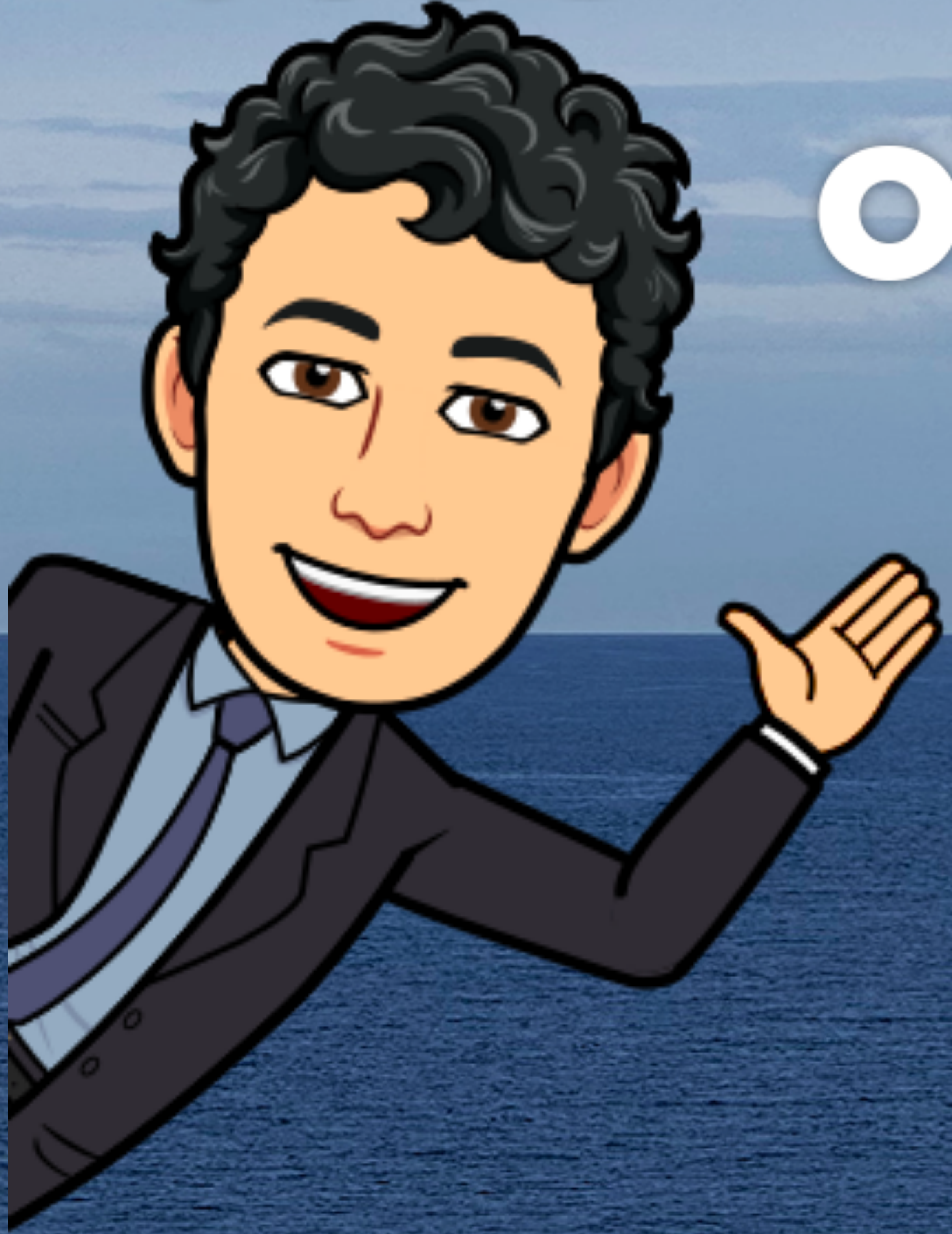
How To Get the Benefits of Rust Traits for Runtime Polymorphism in C++

Eduardo Madrid

2025

How To Get the Benefits of Rust Traits for Runtime Polymorphism in C++



Eduardo Madrid

2025

Who Are We?
Where are we coming from?
Where are we going to?

Me

- Author of Open Source libraries
    - SWAR
    - Type Erasure
- Prior experience as a Team/Tech lead at Snap, and Automated/High Frequency Trading.
- What I publish and share with the community is about maximizing the leverage of effort to results.

Runtime Polymorphism Series

- CPPCon 2020: “Not Leaving Performance On The Jump Table”: Performance
- C++ London Users Group: “Type Erasing the Pains of Runtime Polymorphism”: 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 design patterns
- More in 2025
- Substantial overlap in these topics: experience tells me that the overlap is helping a lot.

What Is Type Erasure?

A way to achieve Runtime Polymorphism on your own, in ways that are different to what the programming language gives you as features

Status

- We come from C++ Now 2025: “Runtime Polymorphism with Freedom and Performance” (Theory, design patterns, performance)
- Hopefully, we would go to CPPCon 2025: “Rust Traits in Style for C++”:
    - Assumes we already have support for Rust Traits and go much further
- Where are we?
    - Showing how to go from Type Erasure to Rust Traits, in practice

Conflict Of Interest

- Embarrassing to promote my own framework as the foundation for these many things
- My framework is just proof that things are possible
- Initially this presentation was not going to happen, but special circumstances allowed it!

The Issue Is Quite Simple:

Identify the essentials to not have to deal with irrelevant details

Abstraction!

Runtime Polymorphism, in essence

- When we are writing the code “compile time”, we don’t know the actual types that the application will be using (hence runtime), just their essential properties.
- We work toward the essence in our code.
- The details are resolved at runtime, this is called “dynamic dispatch”
- John Lakos’ in private conversation: The same type but *configured* to have different behavior

The Subtyping Relation #1

- Via example: a telephone
- WRT the concept of telephone, the concrete things that allow placing a call are "substitutable" for telephone; if all the required characteristics of a telephone, including receiving calls, are also provided, then the concrete thing is a subtype of the supertype, in this case, a telephone.

The Subtyping Relation #2

- Suggests the mapping
  - supertype => base class,
  - subtype => derived type.
- Inheritance is a syntactical and structural relation, what the Liskov Substitution Principle means is that it ought to respect the semantics of substitutability.
- Subtyping is not about inheritance, example: Rust Traits.

The Subtyping Relation #3

- C++ has fantastic mechanisms for *compile time polymorphism*, foreshadowing: we recruit these fantastic mechanisms to convert compile time polymorphism to runtime polymorphism, in one particular way: as Rust Traits.

Subtyping and most programing languages

- Subtyping as subclassing is just about the only abstraction mechanism of most programming languages, including Java and C#, that's why this topic could not be more important.
- Also, that's why, for them, substitutability, polymorphism, runtime polymorphism, subclassing and the Liskov's Substitution Principle are essentially the same thing
- Presents us with the challenge of imagining different ways to achieve this.
- Bonus: subtyping as subclassing is eminently a runtime mechanism.

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

C++'s uniqueness

- The weirdest thing is that C++ allows you to devise mechanisms that end up rivaling the very features of the language! And in user code!
  - This happens in practice:
    - Any good type erasure framework ought to be objectively superior to inheritance + virtual (subclassing) in a variety of ways,
    - Rust Traits ought to be within our reach.

Important Example: Serialization

- Serialization is a subtyping relation need for which typically subclassing does not work well. Let's concentrate on the serialization part.

- The deserialization is, essentially, a Factory design pattern, not really runtime polymorphism, so, we will omit deserialization from the discussion.

```cpp
struct TypeRegistry {
    template<typename T>
    std::uint64_t id() const;
};

extern TypeRegistry g_registry;

struct ISerialize {
    virtual std::ostream &
    serialize(std::ostream &) const = 0;
    virtual std::size_t
    length() const = 0;
};

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();
    }
};
```

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

```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:

- Primitive 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!

```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.

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.
    1. “Puts the cart ahead of the horse”, before we know what are the subtyping relations needed, they have to be supported already, or lots of work to retrofit them.
    2. Also, the “ISerialize” is not unique, rather specific to each application.
2. Take it or leave it: A feature of the language you can’t finesse.
3. And then, there is another world of hurt. See Nicolai Josuttis at this conference.

The Classic:

Watch Sean Parent's
"Inheritance Is the Base Class of Evil"

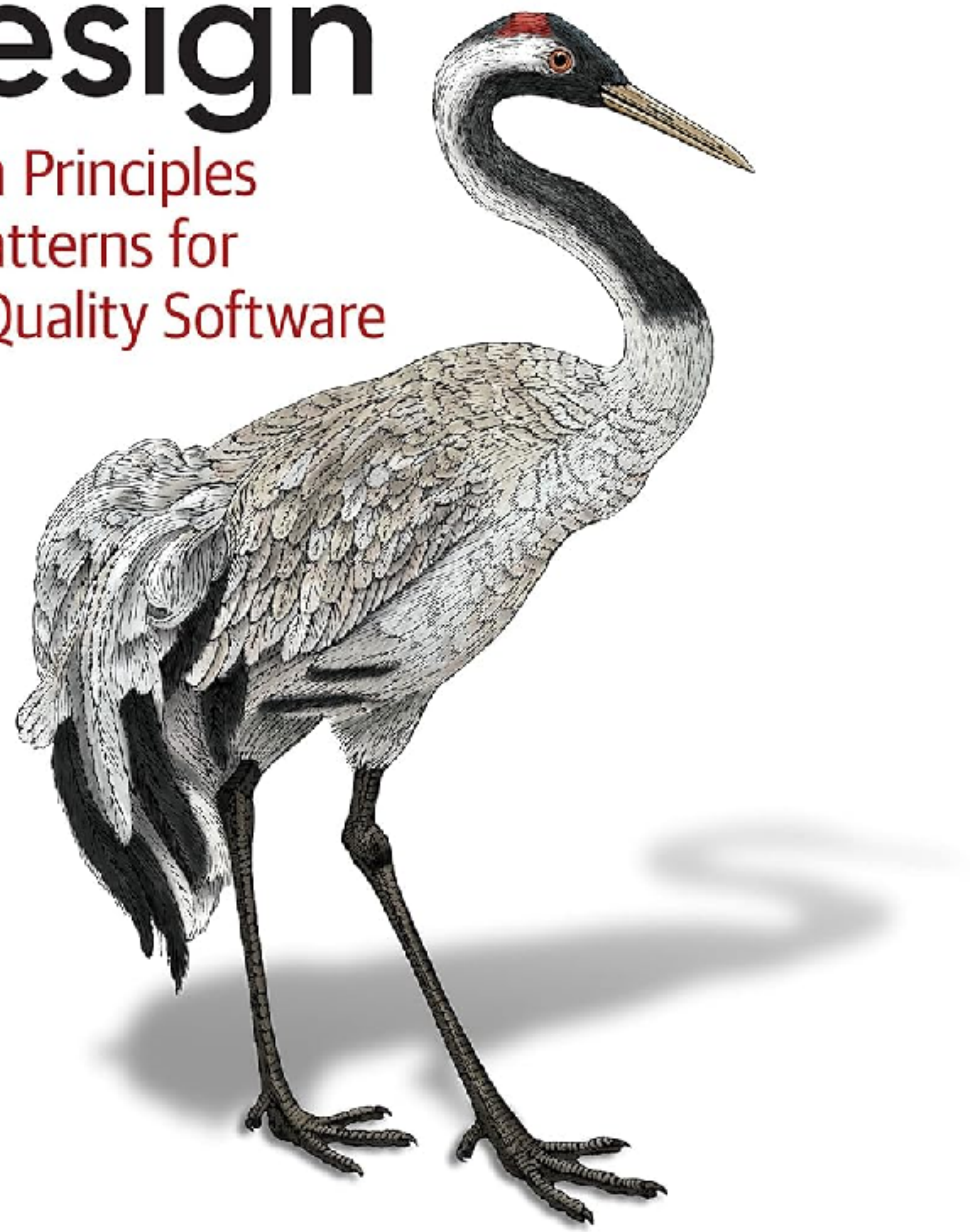
External Polymorphism #1

- We can empower any type to have runtime polymorphism 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
    - Klaus is a member of the C++ On Sea community

O'REILLY

C++ Software Design

Design Principles and Patterns for High-Quality Software



Klaus Iglberger

External Polymorphism #2

- Translate Compile Time Polymorphism to Runtime
  - Basically, a “virtual table”, that seems all you need, except very advanced new possibilities.
- Otherwise: It is a Decorator or Adapter Design Pattern.
- You can use many other Design Patterns, including Strategy

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`
  - If the proposal for CPPCon is accepted, it will be completed then
    - Auditable!
  - Let us know if you’d like to collaborate
  - Will be the ongoing basis for Rust Traits support.

Supporting Rust Traits

1. We need to hold objects of arbitrary types:
   1. Taken care of by zoo type erasure
2. We need to bind the functions/implementations to the type-erasure containers
   1. We need to build the virtual table
   2. Bind the trait functions to the type-erasure container user interface
3. Make it feel more “Rusty”

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

```
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... {
```

```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 {};
};
```

Zoo Virtual Table Building

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

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... how the user uses this

```
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

- Looks like awful code
- …but very repetitive, boilerplate
- I know that we can de-emphasize the customizations we don’t strictly need and encapsulate this very succinctly
  - IDENTIFIERS are crucial here… but we don’t have reflection yet!
    - Use reflection proposals
    - Use preprocessing

Back In Planet Earth

- We have all we need!
- The virtual table to have dynamic dispatch
- How to implement the functions that will be referred to by the virtual table
- The binding between real objects and the virtual table
- How to add them to the public interface
- How to also have destruction and moving taken care of.

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)
    }
}
```

What’s lacking?

- Looks like awful code
- …but very repetitive, boilerplate
- I know that we can de-emphasize the customizations we don’t strictly need and encapsulate this very succinctly
  - IDENTIFIERS are crucial here… but we don’t have reflection yet!
    - Use reflection proposals
    - Use preprocessing

Conclusion

- 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 use value semantics for things that are runtime polymorphic (not “boxed”, not “unsized”)
    - 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.

Conclusion

- C++ is that powerful

END!