C++ now

2025

Runtime Polymorphism with Freedom and Performance

External Polymorphism and Type Erasure

Eduardo Madrid

C++ now

2025

Runtime Polymorphism with Freedom and Performance

External Polymorphism and Type Erasure

Eduardo Madrid

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.

Outline

- Revisit Abstraction, Subtyping, Polymorphism, Subclassing, …
- Solve the needs for Runtime Polymorphism using first the External Polymorphism Design Pattern, and then how it turns into Type Erasure
- Type Erasure opens multiple opportunities to do amazing things.
- We will show some of them

The Issue Is Quite Simple:

Identify the essentials to not have to deal with irrelevant details

Abstraction!

The Subtyping Relation #1

- Via example: If we just want to place a call, we don’t care whether the telephone is a smartphone, a landline phone, or perhaps even a phone app in a computer, what we care is that it is able to “dial” a number and have the audio capabilities to talk, those are the essentials.

- 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

- This suggests a simple mapping in programming languages that have inheritance: a base class ought to be a supertype, and derived classes subtypes.

- Inheritance is a syntactical relation, what the Liskov Substitution Principle means is that the syntactical relation ought to respect the semantics of substitutability.

- Subtyping is not about inheritance

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
- But we should not fall into that trap.
- Bonus: subtyping as subclassing is eminently a runtime mechanism.

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

Generic Programming Subtyping

- Take the example of traversal of a range: It is distilled to these essentials:
  - Iterator dereferencing (`*iter`, `iter->member`), increments (`++`), a sentry (`end`), iterator equality (`==`, `!=`) and the LSP.
  - With that, anything is iterable, traversable.
  - Again, we use only those operations and the compiler does potentially huge amounts of work for us
- Paradoxically, this is a preeminently compilation-time capability

C++ and Abstraction

1. It never committed to a particular way of doing things, how it moved from Object Orientation to Generic Programming.
    1. The primary subtyping mechanism used to be subclassing
    2. Generic Programming: Taking things to their most abstract form so they are most general
2. It has very expressive abstraction mechanisms that don’t lose performance, including support for most of Functional Programming.
3. Its fundamental library is essentially user code “blessed”

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,
    - People can use the mechanisms of coroutines to emulate the essential benefits of exceptions
- That’s the importance of freedom: not being constrained by a narrow understanding of what is good or important: no one size-fits-all imposition leads to a diversity of successes.

Polymorphism?
Substitutability, subtypes

Good “Runtime Polymorphism” ought to reflect the advanced powers of C++’s compile time polymorphism; or how to use Generic Programming to make the mechanisms needed for RP.

Good “Runtime Polymorphism” ought to reflect the advanced powers of C++’s compile time polymorphism

Just How Bad is Subclassing?

Pains:

Watch Sean Parent's

"Inheritance Is the Base Class of Evil"

Applause?

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.

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

Referential Semantics Pains: Their own hell

- One extra indirection
- Allocations: memory fragmentation, synchronization, may fail
- Shallow/deep copy?
- Disables local reasoning
- Lifetime Analysis
- Incentivizes sharing, and this complicates lifetime management
- Performance hostile in many other ways: no “data” affordances, RTTI

Subclassing Pains

1. Difficulty to model many subtyping relations:
   Kevlin Henney's "Valued Conversions" [https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4610004b383e5c4f2dffbea0019c85847e18fff4]:
   How would you like to pay for that?

    1. Money?

    2. Bartering?

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

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

Subclassing Pains: Lack of modeling powers

1. Have you heard the concept of "convergent engineering" or "convergent evolution"?
    1. For example, insects, birds, and bats all evolved flight, but they are rather *unrelated* species of animals.
    2. Mammals and swimming. Most primates and homo sapiens swimming.
    3. In Biology this is called, depending, para-phyletic or poly-phyletic groups.
2. If we import Biology's term to Subtyping relations, there clearly are monophyletic relations that are supported just fine by subclassing. But subclassing fails rather catastrophically at paraphyletic and polyphyletic relations!
3. Canonical polyphyletic subtype relation: SERIALIZATION!

Support for Para- and Poly- phyletic subtyping?

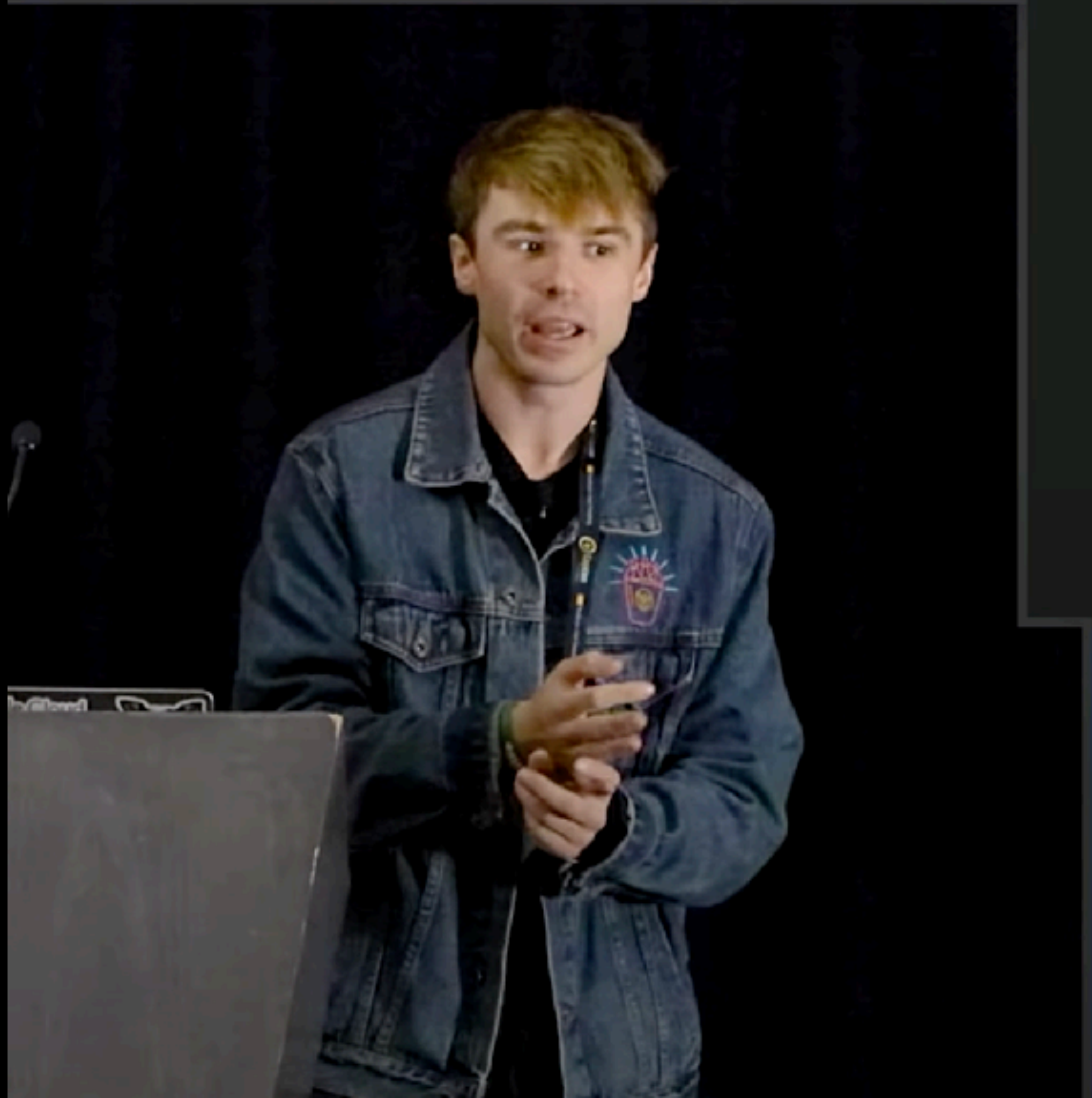
Let’s take a theoretical breather by looking at some code:

Cppcon
The C++ Conference

Cppcon.org

Progressive Egyptian Multiplication

```
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```



Composing Ancient Mathematical Knowledge Into Powerful Bit-fiddling Techniques

Jamie Pond

Last Night’s Jamie Pond's PR: it has a new “affordance”: [https://github.com/thecppzoo/zoo/pull/110/files]

Now we have all to complete the title:

Performance => Freedom

- Runtime Polymorphism inherently requires trampolining.
- Each indirect jump (assembler) is a compiler optimization barrier.
- We need the freedom to organize what happens at each stage of the trampolining, so the compiler can optimize that;
- and be excruciatingly attentive to details concerning the jumps
- We need freedom to organize things exactly right.

Performance => Freedom

- More than half —half!— of the work in zoo type erasure was to prevent the normal mechanisms of the language from acting:
    - But successful: [https://godbolt.org/z/Y7Mr86]
- Fedor Pikus presentation last year “Type Erasure Demystified”[https://www.youtube.com/watch?v=p-qaf6OS_f4] that explains some of the significant performance improvements that apparently I first identified and articulated in Open Source code.

External Polymorphism

- 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

External Polymorphism

- Example in the Compiler Explorer/my old markdown [https://github.com/thecppzoo/zoo/blob/master/presentations/C%2B%2B-online-2025-external-polymorphism-and-type-erasure.md]

External Polymorphism Demo

- An adapter:
  - It refers to the object somehow (a pointer is good)
  - It gives the runtime polymorphism:
    - Via subclassing! (Nothing bad, the original object types are left undisturbed) see Sean Parent’s presentation example.
    - Even better: via the virtual table mechanism.

External Polymorphism:
It’s essence is to give runtime polymorphism to types that don’t have it

Not concerned with runtime polymorphism of the **ownership.**

Type Erasure

- If you own the objects you’re given External Polymorphism:
- External Polymorphism with destruction, moving, and perhaps copying
- Not a hard boundary

`std::any`

- A container that needs `any_cast` to transform it into something usable

`std::function`

- Canonical example
- Not intrusive!
- You can have local variables of type `std::function` (including function parameters) as well as members (not forced for them to be pointers or references)
- If it allocates, it is a fallback mechanism, this lessens the problems.

However, it is a very bad design

Not the fault of the inventors, but the fault of our community to notice the problems and correct them more opportunely

`std::function`

- Performs type-erasure, like `std::any`, and a bunch of other things (does not follow the “single responsibility”) principle nor others:
- Without configurability:
  - No way to indicate the size, alignment of the local buffer
  - It is copyable, forcing the targets to need to be copiable: Hostile to move semantics, therefore it is hostile to the strong exception guarantee
- Throws an exception when misused: See John Lakos on `std::vector::at`

`std::function`

- Supports only one anonymous call interface
    - Not a data member: Indirect function call penalty
- Annoyances like using RTTI and its inefficiencies
- Not a function but a **trampoline**!
    - Example const-call: the trampoline might be "const" and the target non-const, in the same way a pointer might be const and point to non-const
    - "Paternalistic" forwarding of arguments (discussion with colleagues)
- I need to stop!

O'Dwyer's std::function design space

- At https://quuxplusone.github.io/blog/2019/03/27/design-space-for-std-function/
- Ownership
- Local Buffer Configuration
  - Disable heap allocation
- Fundamental Affordance set:
  - Move-Only? Not even movable?
- Is the user-specified affordance `const`, `noexcept`?

Legitimate dimensions

- Most of the combinations of choices make sense and have good use cases
- The community drains discussing options that do not have a clear best
- Missing the point “how do we make these choices available to the user”, or perhaps, the implicit assumption is the belief that it is impossible to make mechanisms that let the users choose.

Consequences: People redesign, poorly, and implement even worse

Because this is highly technical and misunderstood

And it turns out the design space is much more vast!

Rather than relying on the Standard Library to supply ever more species of type-erasure fishes, we should learn how to type-erasure fish ourselves!

I thought it could not be done much better

To understand the roadblocks, I proceeded from first principles

My discovery:

1. Subtle wrong assumptions
2. Unclear thinking

Zoo Type Erasure

- Provably optimal performance solution, codegen.
- I’ve shown the modeling powers of zoo’s type erasure are beyond any other framework
- Hence: a solution exists!
- I’d love if you try. Especially if you reject my framework and do it differently! I will try to help you as much as you want. Why? I know there is a lot to be learned, “tip of the iceberg” kind of thing.

Feel free to contact me!

Type Erasure is “internal-external polymorphism”

Internal-External Polymorphism

- I tweaked the nomenclature to arrive to a deliberate contradiction:
  - External Polymorphism is an “stand-offish” way to give polymorphism to things that don’t have it.
  - Type Erasure does that and also takes complete control of the target:
    - VALUE SEMANTICS EMERGE!
      - Even if underneath there might be a reference!
- Emergence of complexity: unpredictably interesting and useful behaviors that arise only in between contradictory design goals

Internal-External Polymorphism #1

- Example: a “Value Manager” that is neither a local buffer (also called “small buffer optimization, SBO”) nor a simple heap pointer but an *opt-in* value manager made by an *user* of the framework, so the pointer is a shared pointer.

Internal-External Polymorphism #2

- Let’s go one further, to begin to implement copy on write.
- Again, Jamie’s PR.
- That is my idea of “demystification”: showing you the real, production stuff, so that you know that what you see is as bad as it gets… unfortunately, other people demystify in ways that I think are oversimplifications.

Internal-External Polymorphism #3

- Much more radical: The given objects to infuse them with runtime polymorphism are never stored, but rather, “scattered” as in “Data Orientation Scattering” into collections of homogeneous data types.
  - We get rid of the types of given objects, and represent them *internally* in radically different ways—while still preserving all of the runtime polymorphic interface!
    - We have the cake and eat it too!
    - The process of abstraction inherent in the Liskov’s substitution principle reduces unnecessary details and this allows us to get more performance!
    - Negative performance cost abstraction!

END!