C++ONLINE

EDUARDO MADRID

TALK:

EXTERNAL POLYMORPHISM AND TYPE ERASURE

A VERY USEFUL DANCE OF DESIGN PATTERNS

2025

Redux

Eduardo Madrid

Introduction a bit later

Objective

See How Useful Type Erasure is

This is rather unique to C++

- C++ is perhaps unique in allowing users to implement features intrinsic to programming languages with practical benefits:
  - Better performance
  - Higher modeling power
  - A library that rivals, successfully, the very inheritance + `virtual` override mechanism in the language!

Leading Edge

- We will talk about capabilities we didn’t know we have:
  - A reaction might be to believe the new capabilities are “over engineering”, “solutions in search of problems”
  - But we can use them because they don’t introduce hard problems, just:
    - More awful compilation errors
    - Still demonstrably better performance and codegen
- Most of this talk is material I’ve not seen discussed elsewhere

Substitutability

If we get substitutability right, We've partially succeeded!

Substitutability: Telephone

- We want to place a call, but we don’t have a telephone:
  - We can ask a friend for a smartphone.
  - We can install an app in our computer to simulate a physical phone: a virtual/app phone
  - We can use an old telephone plugged to a landline
    - Even a rotary!

Substitutability: Telephone

- We have an abstract concept of a phone, and we have concrete types of phone (smart, virtual/app, landline) that themselves have their own more specific concrete characteristics
- We typically model this with **subclassing**:
  - A “base class” or “interface” with “`virtual`” functions
  - That are “overridden” by derived classes

Substitutability: Subclassing

- Suitable for runtime only
- It is:
  - A pre-determined need
  - Intrusive: it “intrudes” into the writing of the software component

Substitutability: Container Iteration

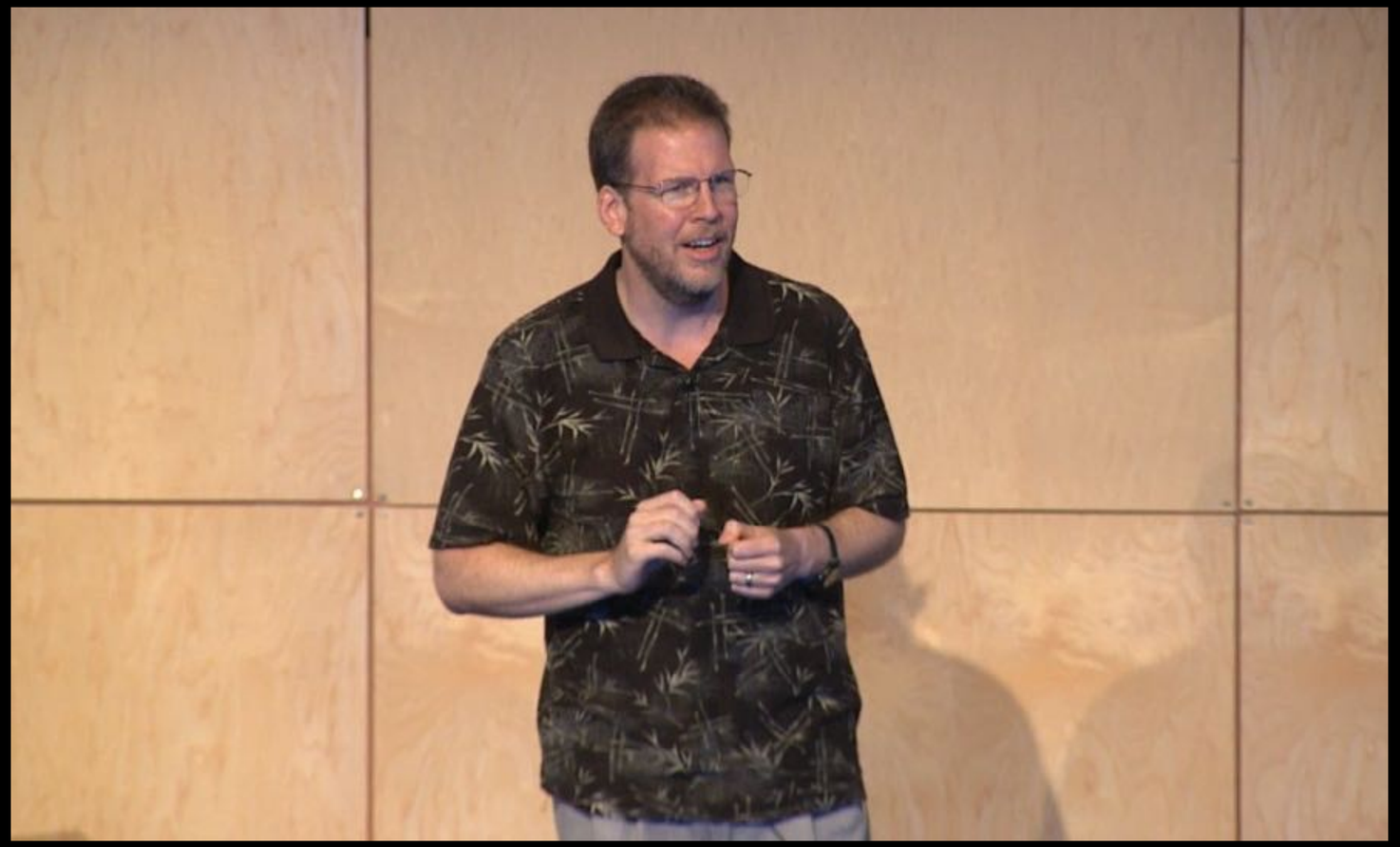
- Iterators:
  1. Equality (== and `!=`)
  2. Sentry (`end`)
  3. Dereferencing (`operators * and ->`)
  4. Increments (++)
  5. Semantics (not just sytax) of traversal
- This is incredibly general!
- Works great at program writing, a.k.a. “compilation time”

Substitutability:
We can substitute a concrete type where an abstract concept is required and things work.

That’s the famous Liskov’s Substitution Principle

This ability to substitute is called polymorphism

Pains of Subclassing



Pains:

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

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.

Referential Semantics Pains: Their own hell

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

From Substitutability to C++ templates

We can use Compile-Time polymorphism for runtime polymorphism!

Important note:

- In our industry, because of the poverty the other programming languages, substitutability, polymorphism, runtime polymorphism, subclassing and the Liskov's Substitution Principle are essentially the same thing.

- If this presentation helps you realize there are key differences, the partial success is much better.

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

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

`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

Conflict of Interest

Zoo Type Erasure

- Provably optimal performance solution, codegen.
  - Partial proof of performance: 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.
- 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!

Who Am I?

- The author of the zoo libraries:
  - SWAR
  - Type Erasure
- Before, things like Financial Exchange Connectivity to exchanges such as the CME via “MDP3”, in production, for a Hedge Fund (my CPPCon 2016).
- Things in common of what I share with the public:
  - Production code, in very demanding scenarios
  - Out of the ordinary results.
  - Examples:
    - SWAR: beats, objectively, highly optimized code such as GLIBC’s.
    - Type Erasure: Used by Snap in Snapchat, at the critical places where std::function was used. Surfaced subtle errors in pre-existing code
    - Financial market data connectivity: Foundational work for other people who has also presented at these conferences.

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!
      - Even if underneath there might be a reference!
- Emergence of complexity: unpredictably interesting and useful behaviors of things in between contradicting design goals.

Internal-External Polymorphism

- 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

- 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!

External Polymorphism and the Internal Polymorphism of controlling objects interact in very interesting and useful ways: The Dance.

Outgoing Remarks

- If you feel the advanced capabilities of Type Erasure are “over engineering”, I have good news: I think you got important parts of this presentation, because at least you know new things, those you are skeptical about.
  I just hope that by the time your imagination catches up to the capabilities, you still remember enough of them.

END!