C++ONLINE

EDUARDO MADRID

EMPOWERMENT WITH C++'S GENERIC PROGRAMMING PARADIGM

2024

Hi!
We Are:

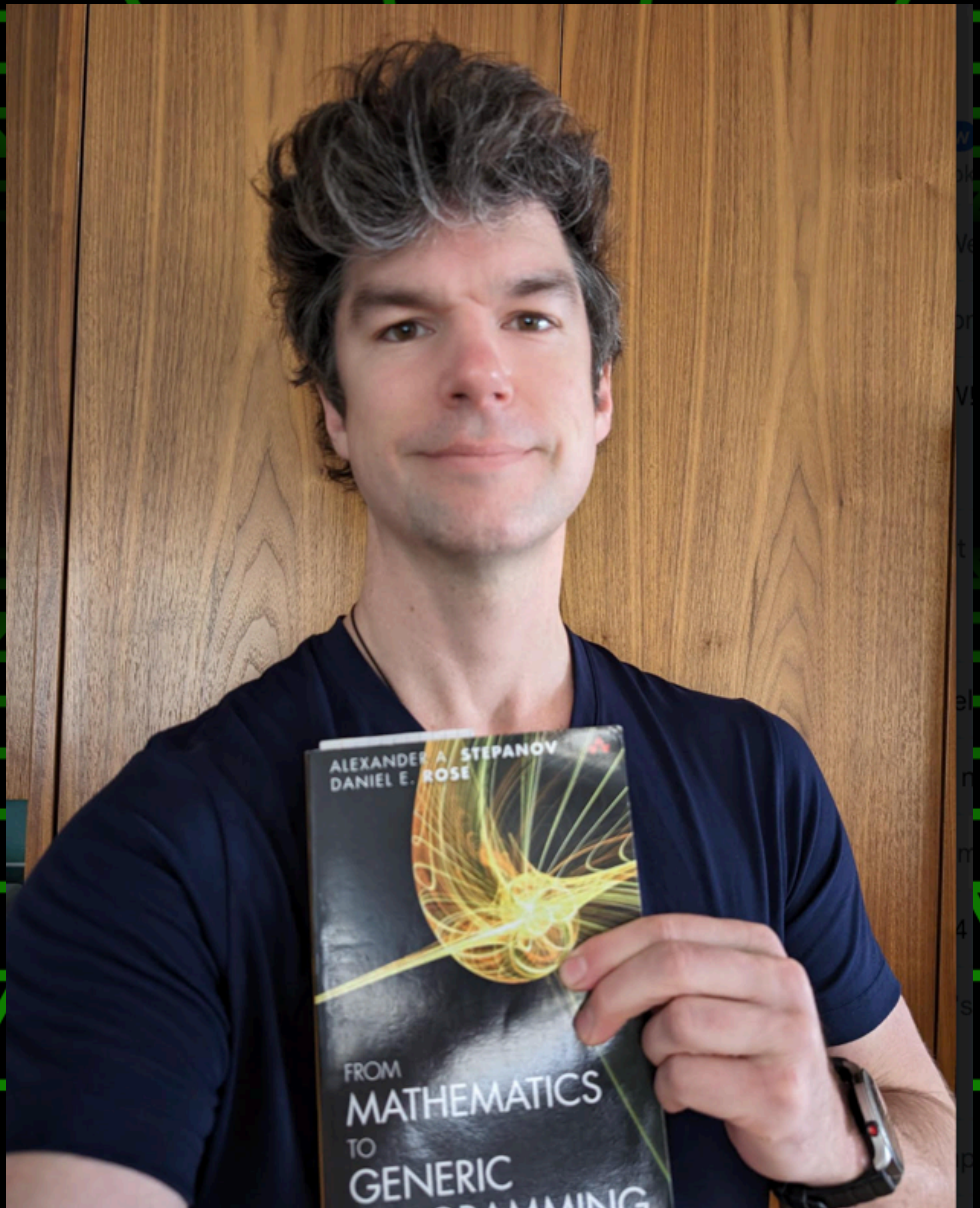
Eduardo

Jamie Pond

Hi!
We Are:

Scott Bruce

Hi!
We Are:



Proper introductions a bit later

But: my opinions, my mistakes.
Their merit for any successes today

Objective

Empowerment

Achieving more with less effort

How?

- **Guidance** to express more knowledge into source code
- **Guidance** about examples:
  - Successes of Generic Programming, well established concepts
  - Failures where Generic Programming is absent
  - **Demonstration** of Generic Programming Concepts **being made!**

Why Us, “Los Tres Amigos”?

- We have applied what we bring to share:
  - Frankly, great successes at large scale (Google, Snap), Scott, Eduardo
- Friends having fun at old and new Open Source showcasing the approach we talk about:
  - A solution for Runtime Polymorphism that is best-of-its-kind
  - Release to the public of a production strength SWAR library:
    - Thoroughly benchmarked implementations/demonstrations of useful functions:
      - Simple code, yet significantly better performing than the results by large teams presumably doing their best
- Generic Programming concepts being made!
- The complement of a fresh, new perspective: Jamie

Yes! Lot’s of Code & Godbolts!

Buckle up! Lots of material!

“Electromagnetic Wave Progress” metaphor

If at the end you know what this means, you grokked this presentation

Generic Programming Paradigm

Whenever I tell a colleague ‘I like the Functional Programming Paradigm’ I get the feeling they understand I mean ‘I like practices that help make stuff that works’, …as if there would be an alternative, not that I mean there is a whole theory full of useful techniques, of profound wisdom, that ease making great software.

Generic Programming Paradigm

“Generic Programming” is worse: they seem to understand “generic” as indistinct, code without cleverness. The attitude that code must be dumb so that even bad programmers can work with it, that cleverness must be rejected unless there is solid evidence for it, otherwise, it is showing off, an evil. That cleverness is evil.

GPP

The practitioners of Generic Programing agree that it is the attempt to find the most abstract representation of algorithms and data structures, so that we **GENERALIZE** to the maximum

GPP

- We solve one problem once and for all:
  - No repeating ourselves and making the occasional mistake
  - Subsequent improvements are very impactful:
    - See examples of containers and move semantics.
- Abstraction removes specificities that impede **composition**
- Abstraction leads to a deeper understanding that guides deeper improvements
- GPP cures you from a pervasive vice in our discipline, wait for the big reveal of what that vice is.

GPP

- Abstraction requires formulating new concepts:
  - Note I don’t refer to the language feature we call concepts.
  - I call a bunch of related concepts a “conceptualization”, I say, for example, the “design of iterators in the C++ standard library” only for mainstream audiences, I would like to say “iteration conceptualization”.
  - I call very large conceptualizations a “theory”: “Stepanov’s theory of data structures and algorithms.”

GPP

- Thus GPP is the opposite of code without cleverness: you maximize the expression of knowledge into source code:
  - Very strange looking code, perhaps intimidating.
  - First partial explanation why GPP is not popular:
    - IMO GPP potentiates the programmer in a literal way, like a base to an exponent, some times it empowers a lot, for programmers just being able to carry their own weight in projects, not useful, for the others, GP makes their shortcomings more noticeable.
    - Intentionally provocative opinion: the mean of programmer ability is a lot lower than the average. Still, the social aspect of our discipline dictates that it caters to the programmer of mean ability, which might be hostile to the upper range:
      - Use the opportunities to learn GP, since they are not abundant, because they have no sponsors.
- Perhaps tools like Copilot, ChatGPT are devastating the market for below average colleagues, but for us, it seems quite helpful, as if they increase the value of our work. What a paradox. One example coming.

Principles:

Minimize Coupling
Maximize Cohesion
Maximize **Orthogonality**

--Coupling ++Cohesion ++Orthogonality

- Orthogonality example:
  - Unix streams:
    - The APIs for consuming data are have nothing to do with the APIs to create, clean up the streams: the idea of *dimensions*.
    - It is better if your software components act only on their own dimensions, not affecting the others.
  - Orthogonality is one of the things that Object Orientation struggles with:
    - Object Orientation is limited to variations on a single aspect.
      - It forces you to shoe-horn hierarchies into whatever you are modelling:
        - It is terrible for modeling refining of independent aspects, even worse in supporting covariant or contravariant aspects.
    - Generic Programming, templates, have no problem to vary on multiple axis. We will see this momentarily.

First Example: Iterators

Iterators

- Iterators abstract and generalize the idea of pointers
- The pointer is the key concept with regards to *accessing* the data and *traversing* the data structure and pointers have semantics of *optionality*.
- The way iterators model pointers is more nuanced: they don't replicate the pointer arithmetic rules, they make a distinction between traversing capabilities (forward, bidirectional, random access, …), and see how they replicate optionality:
  - They do not require a special or "magic" value such as nullptr to denote the absence of a valid value, they have this brilliant quirk of referring not to elements in a data structure, but to the borders between elements according to the traversal order:
    - even empty ranges have one position, the end() sentinel
    - The end() sentinel allows synthesizing of optionality.

Standard Template Library

- Talking about iterators without talking about algorithms is akin to clapping with one hand.
- Sort:
  - `std::sort(RandomIterator b, RandomIterator e, Comparator)`
  - Compare to C:
    - `void qsort( void* ptr, size_t count, size_t size, int (*comp)(const void*, const void*) )`

Skip live: sort lessons

- Sort uses a mountain of concepts: iteration, iterator powers, copy and move abilities, noexceptness, exception level guarantees, …
- It still has “native” or handcrafted performance
- Qsort is hopeless: limited to arrays, value semantics.

Skip live: sort lessons

- Sort uses a mountain of concepts: iteration, iterator powers, copy and move abilities, noexceptness, exception level guarantees, …
- It still has “native” or handcrafted performance
- Qsort is hopeless: limited to arrays, value semantics.

Iterators - Criticism

- Be attentive because this departure from how everybody else does things, regardless of its merit, is bound to be misunderstood, underappreciated, and misused; at the same time, the criticism may reveal opportunities for improvement:
    - Tomorrow Arno Schödl is going to tell us “why iterators got it all wrong”, I have seen his argument, thus, regardless of my agreement with the points, I can recommend people to listen to what Arno says. My guidance is to keep in mind the question of how to reproduce optionality, which is inherently complex, which seems to be the reason for the duality between iterators pointing to an object or to a border.
    - Just the same, there are other great resources: Barry Revzin’s "Iterators and Ranges: Comparing C++ to D to Rust” (https://www.youtube.com/watch?v=d3qY4dZ2r4w) that seems unconvinced about the superiority of C++’s standard library design:
        - One point he mentions is the excessive boilerplate around `const` and non-`const`: but is this the library or has he surfaced a deficiency in the language itself, that does not allow the expression of the symmetry between `const` and non-`const`? Is this corrected in the language with something like “deducing this”?

Iterators: Empowerment Opportunity

- I think the majority of the benefit of Generic Programming comes from refining concepts.
  - Are you attentive to criticism of established concepts?
    - What are you able to do with regards to what you agree of the criticism?
      - In the benchmarks we will share with you, we created multiple benchmarking corpus, so the benchmarks are iterating over the corpus. Like Barry Revzin pointed out, C++ iterators require crazy amount of boilerplate, I did not bother, I used just about the simplest iteration model, for the ad-hoc purpose of showing corpus to you.
  - Do your codebases allow you to refine concepts?
  - What about introducing new concepts as soon as they are helpful, regardless of how poorly articulated they may be?
    - Can you imagine a codebase where most of what is done is to refine concepts? It looks strange!

Big Reveal #1

Electromagnetic Wave Progress

“Mathematics Is Organized Reasoning”

Richard P. Feynman, NPW

Let there be light!

- Light is just electromagnetic radiation in a very tight range of frequencies. Electromagnetic radiation is the consequence of electric energy propagating in waves. Why does electric energy propagates in waves?
  - Changes to the electric field induce a change in the magnetic field. In turn the change in the magnetic field induces a change in the electric field. That's how you get a propagating wave.
  - Mind you, the magnetic force can't make "work".

And there was light

- Here’s the metaphor:
  - Substitute propagation with progress, as if there is a desired direction, higher software quality.
  - Substitute the magnetic field for the theory of software engineering
  - The electric field for the practice of software engineering
  - Insights in the practice of software engineering prompt theoretical insights, that then prompt practical insights, progress.

FM and AM radio

- Notice that the theory does not give you direct results, just like you can't get work out of the magnetic force, but people have no faith into being able to transform theory into practical benefits. Here's where progress stops!
- The analogy holds nicely on closer inspection:
    - If in the analogy what we do is FM Radio, of around 100 MHz, the standard progresses in the AM Radio frequencies, of around 1000 KHz.
        - For us, the step of going from theory to practice occurs so often that it is very real
        - Progress in the standard overcomes much tougher obstacles than what we can.

Generic Programming is “From mathematics to code and back”

Second Example: Reality Check: Benchmarks

All of this ‘yada, yada’, does it have any substance?

- Scott and I have been collaborating on developing a SWAR library to apply it to things like Robin Hood hash tables. Then Jamie joined us. Because Scott and I had recently (re)read “From Mathematics To Generic Programming” we have been on the mindset of systematical improvements.

- We wanted to prove the value of our efforts by demonstrating in real life what we get following our methods:

    - Something realistic but very well constrained so the causality analysis is easy

    - Hopefully something important

    - We meant “business”: comparing to real world production code

    - Prepared for this very presentation

- The expectation was to nearly match the results, thinking that, after all, our approach is rather simple and generic, as in the pejorative meaning of “generic”.

We didn’t really match prior results.

Not appropriate to call our results a match.

Our results blew past!

Benchmarks today

- Mind you that the results of today add to the performance results I've been communicating over the years concerning my work, that now span several application areas, so, this is not specific to SWAR, rather a representative of what you get with the electromagnetic wave.
- Let's take the implementations in GLIB C for `strlen` and `atoi`:
  - It can be said, literally, that GLIBC runs the world: deepest foundational library in Mac OS, Linux, iOS
  - Both functions are very well known, ideal to have a broad conversation about.
  - However, not the best example to portray the advanced capabilities of our library.

```
--------------------------------------------------------------------------------------------------------
Benchmark                                                             Time             CPU    Iterations
--------------------------------------------------------------------------------------------------------
runBenchmark<CorpusLeadingSpaces, InvokeGLIB_Spaces>              35790 ns        35777 ns         18674
runBenchmark<CorpusLeadingSpaces, InvokeZooSpaces>                20545 ns        20537 ns         35738
runBenchmark<Corpus8DecimalDigits, InvokeLemire>                   1874 ns         1874 ns        373622
runBenchmark<Corpus8DecimalDigits, InvokeZoo>                      1720 ns         1720 ns        406367
runBenchmark<Corpus8DecimalDigits, InvokeLIBC>                    12665 ns        12661 ns         54563
runBenchmark<CorpusStringLength, InvokeLIBC_STRLEN>                5569 ns         5567 ns        124866
runBenchmark<CorpusStringLength, InvokeZOO_STRLEN>                 6579 ns         6577 ns         99661
runBenchmark<CorpusStringLength, InvokeZOO_NATURAL_STRLEN>         6808 ns         6803 ns        100047
runBenchmark<CorpusStringLength, InvokeGENERIC_GLIBC_STRLEN>      10537 ns        10533 ns         67749
runBenchmark<CorpusStringLength, InvokeZOO_AVX>                    2376 ns         2376 ns        287489
runBenchmark<CorpusAtoi, InvokeGLIBC_atoi>                        35106 ns        35099 ns         19625
runBenchmark<CorpusAtoi, InvokeZOO_ATOI>                          10637 ns        10634 ns         64833
runBenchmark<CorpusAtoi, InvokeCOMPARE_ATOI>                      45735 ns        45723 ns         15510
runBenchmark<CorpusStringLength, RepeatZooStrlen>                  6847 ns         6846 ns         98706
```

```
InvokeLIBC_STRLEN>                       5569 ns
InvokeZOO_STRLEN>                        6579 ns
InvokeZOO_NATURAL_STRLEN>                6808 ns
InvokeGENERIC_GLIBC_STRLEN>             10537 ns
InvokeZOO_AVX>                           2376 ns
```

References To Benchmarks

- Conventional CMake project:
  - https://github.com/thecppzoo/zoo/blob/master/benchmark/CMakeLists.txt
- We consider the code to be production-strength.
- The corpus for the benchmarks is deliberately constructed to represent, either:
  - Realistic use
  - Statistical profiles to highlight the differences between implementations
- Godbolt link (updated): https://godbolt.org/z/4qa15G6MK

What is compared?

- GLIBC refers to the platform implementations for this functions.
  - In the case of strlen, AVX2 and NEON assembler
  - The non-SIMD “generic” GLIBC implementation
- Zoo:
  - Zoo strlen: what I think is the most performing implementation
  - Zoo “natural”: the most straightforward implementation using our library
  - Zoo AVX/Neon: our calque of “natural” to use AVX/NEON
    - Notice the code is practically identical to the “natural”

What have we just seen?

- GLIBC shows a mountain of effort to code these routines.
  - Counterproductive effort, ultimately
  - Hand-crafted assembler with manual loop unrolling
  - Complicated, really hard to decipher
  - Self-contradictory
  - Insufficiently explained
  - Untouchable?

What have we just seen?

- Zoo
  - Simple, natural, straightforward, relatively understandable
  - Principled:
    - conspicuous **absence** of conditional branching
    - Almost Always Auto: This is actually very important, we’ll see in a moment
    - All functions are the same:
      - One loop, its invariant is being able to process one full block
      - Prolog: make the loop invariant by filling in the block in a way that won’t change the end result
      - Epilog: the loop invariant is broken, fill the block to collect the last part of the result, exit.

Big Reveal #2: The nasty vice is…

OVERSPECIFICATION

Overspecification

- You overspecify, you box yourself into suboptimal scenarios
- You need maximum freedom
- Adopt the practice of erring on the side of under-specifying
  - My “Rehashing hash tables” at C++ Now 2022 shows a good example.
- Generic Programming dis-incentivizes over specification:
  - It quickly leads to compilation errors because you use *details* of your templates you ought not to.

Third Example: Polymorphism

Net Negative Knowledge

- One big problem in Java is that data abstraction, encapsulation, is only achieved by referential semantics.
- Referential semantics is so expensive, that it motivates mitigation such as caching.
- These mitigations of the cost of referential semantics make classes deeply stateful
- Deeply stateful instances of objects create the conditions for particularly hilarious situations:
- java.net.URL.equals(Object) doesn’t compare the string URLs, the real equality requires figuring out what the things refer to (they may be different string URLs to the same thing), so, hilarity ensues: “equals” may invoke a DNS request!
- Worse: because of equals, even URL.hashCode() may also trigger DNS requests!!!!
- URL instances have referential semantics to data (a string) that itself is a reference to a network resource that requires the activation of just about the whole network stack for a triviality such as a hash code.
- This also exhibits the anti-pattern of “stringly typed programming”.

Runtime Polymorphism

- Motivation for this example: Showing things that would be impossible without Generic Programming, one example of extreme potentiation, empowerment.
- In many of my presentations I've explained why Inheritance & `virtual` overrides are no good. Also:
  - Proved how my implementation of Type Erasure has unbeaten-as-optimal performance,
  - Including *better* performance than the language feature specifically dedicated to runtime polymorphism
  - Yet, even more staggering benefits, not yet possible in any other Type Erasure framework nor even programming language.

Too Good To Be True? How?

I’ve answered them in many ways including presentations, but a recap of the most relevant points:

- Systematic application of first principles painstakingly expressed with performance preserving templates (pure application of Generic Programming).
- Went past Louis Dionne’s Dino’s capabilities because of the identification and formulation of whole new Generic Programming concepts:
    - Profound reformulation of the concept of “affordance”,
    - Invention of the concept of “Value Manager”,
    - The application of the Generic Programming design pattern of Alexandrescu’s Policy
- Deconstruction of the theory of value semantics so that users could pick and choose fundamental operations as stock or user affordances.

Sounds like an epic amount of work

- Certainly a lot of work, but not as much as you might imagine, because of the electromagnetic wave progress metaphor

- In particular, I had the benefit of submitting that code to the challenge of the hundreds of millions daily active users of Snapchat and Snapchat's stewards, my awesome former coworkers.

- The ultimate success surprised even me. The incredible situation of having a lot to learn from successes of your work beyond your expectations.

  - I say this because I want to prove to you the method works, so that you take it seriously. The method is the electromagnetic wave that seems only possible via Generic Programming.

Alexandrescu’s Policy pattern

- Increases *cohesion* by bringing several independent aspects together.
- In zoo Type Erasure, the policy is what *configures* (its own pattern) all of the operation of zoo::AnyContainer. The configuration maintains these *orthogonal* aspects:
  - The characteristics of the local buffer (size, alignment) to hold or to refer to the managed objects
  - The set of capabilities of the managed objects, their common public interfaces (the “affordances”)
  - The interface between AnyContainer, the “value managers”, and the objects managed.
  - It is worth noting that the value managers are covariant with the affordances:
    - The exact type of Value Manager is directly related to the needs from value management of affordances, and vice-versa: the capabilities of the managed objects are influenced by the support that value managers can provide: covariance.

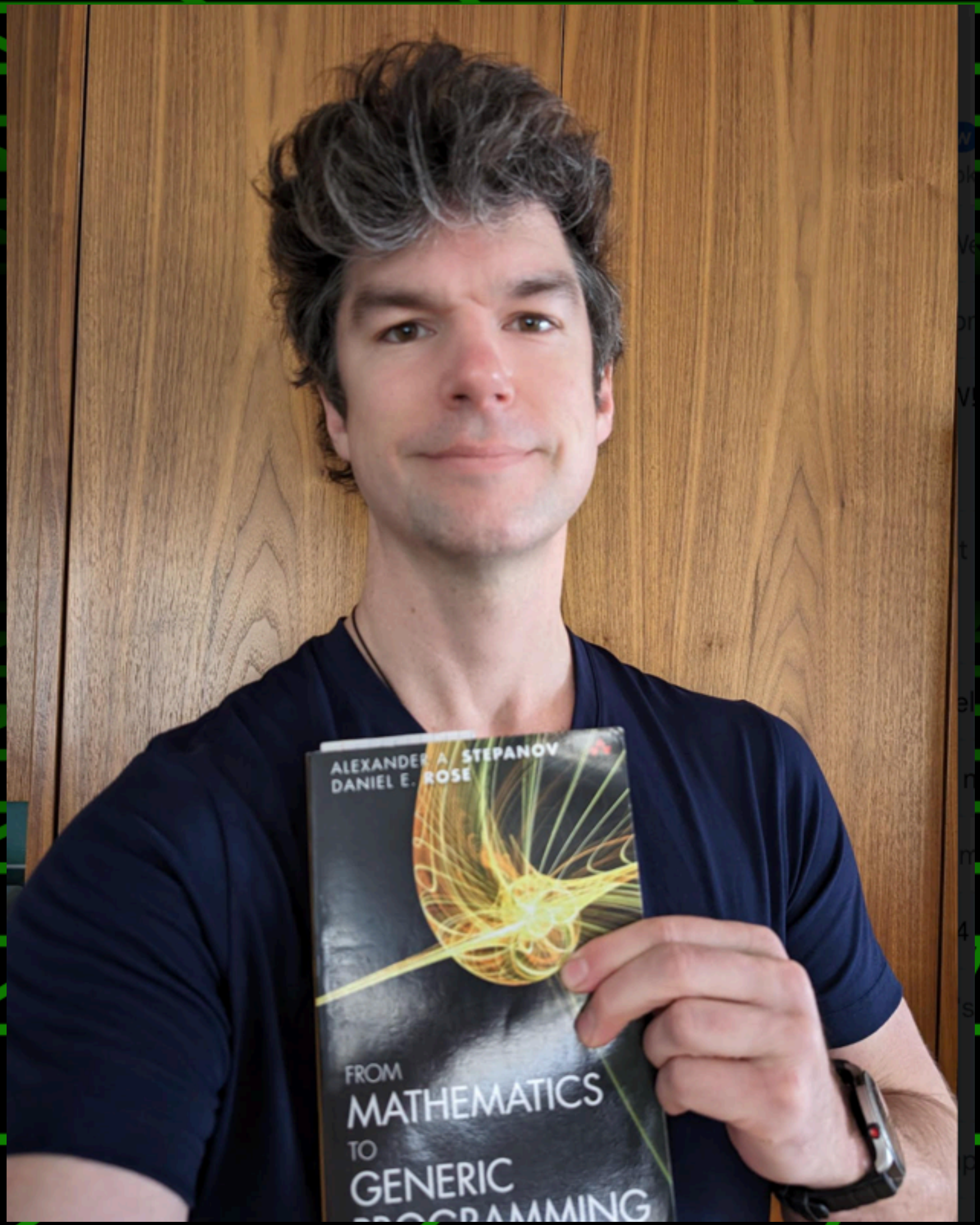
Sounds theoretical, not relevant to you?

- That’s the point!
- If you don’t understand the concepts, then you miss out on what this framework can do for you.
- You may object: “but I’m a very good programmer, and since I’ve never needed to know any of the things you just mentioned, I tend to think none of that stuff is useful in practice”
- The essence of becoming much more capable than what you used to be, empowerment, is that you have had a shift in understanding:
  - you’ve become more powerful because you’ve gained understanding of more powerful concepts.
- Don’t be dismissive of what you don’t understand
- Go and learn Functional Programming, go watch Ben Deane’s weirdest sounding presentation titles.

Jamie Pond

- Jamie is lead audio software engineer and LA-based startup mayk.it

Scott Bruce



- The best software engineer I’ve met
  - Career at Google, Snap, where we met.
- Previous collaborator