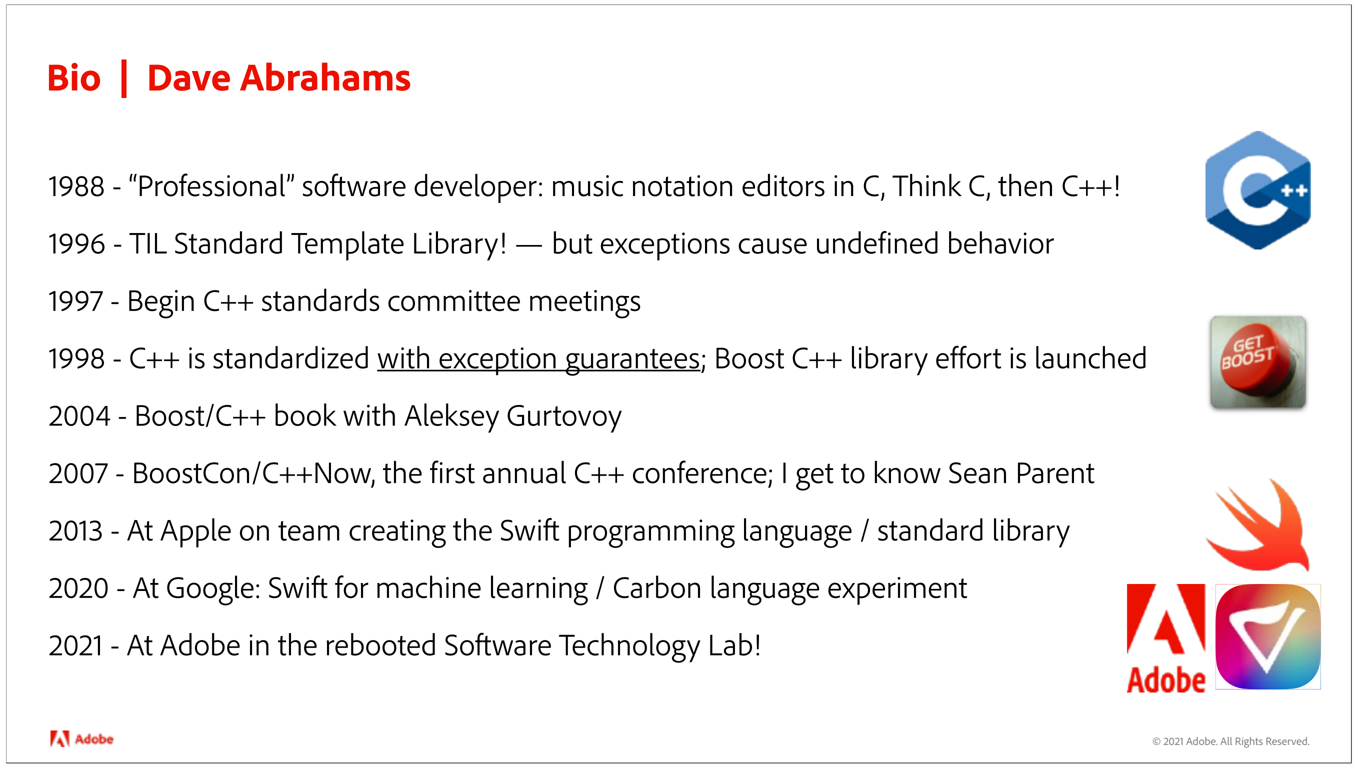
Values

Regularity, Independence, Projection, and the Future of Programming

DAVE ABRAHAMS

Cppcon The C++ Conference

2022 September 12th-16th



Bio | Dave Abrahams

1988 - “Professional” software developer: music notation editors in C, Think C, then C++!

1996 - TIL Standard Template Library! — but exceptions cause undefined behavior

1997 - Begin C++ standards committee meetings

1998 - C++ is standardized with exception guarantees; Boost C++ library effort is launched

2004 - Boost/C++ book with Aleksey Gurtovoy

2007 - BoostCon/C++Now, the first annual C++ conference; I get to know Sean Parent

2013 - At Apple on team creating the Swift programming language / standard library

2020 - At Google: Swift for machine learning / Carbon language experiment

2021 - At Adobe in the rebooted Software Technology Lab!

I normally don't start talks this way, but I've been away from the C++ community
for a while and some of you may not know me. Briefly,

- I started out with the C++ committee in 1997 and am responsible for the fact that the library has exception safety.
- I was a founding member of Boost and then
- I founded the first annual C++ conference, known first as BoostCon and then as C++Now
- In 2013 I went to Apple, where I was one of the original creators of the Swift programming language
- And now I'm at Adobe, an awesome place to work where I get to pursue my professional mission to empower programmers.

So that's why I'm here, to improve the tools you use to develop software, including the mental tools.

But first, real talk. People, we have a problem.

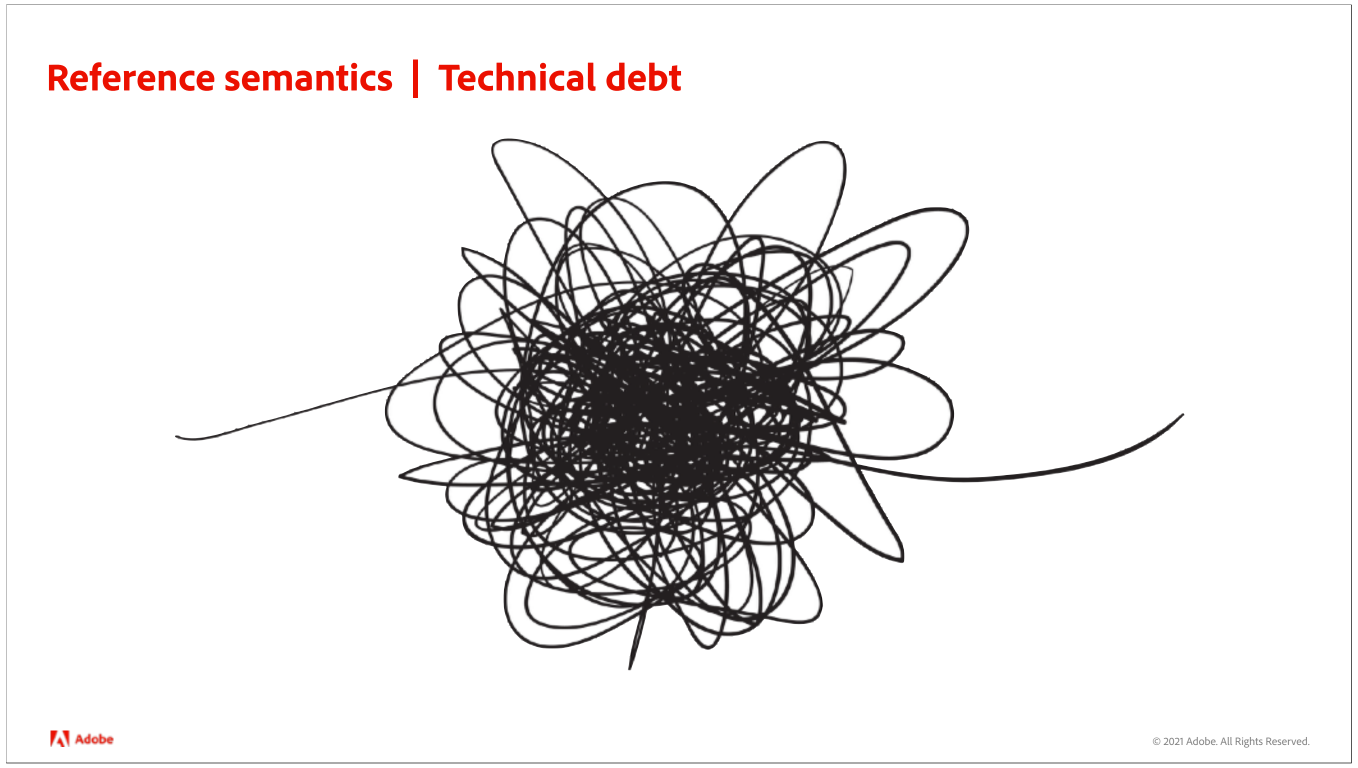
It's one that we don't like to face, and maybe can't even see because we're swimming in it. It's like water to the fish.

I'm talking about reference semantics.

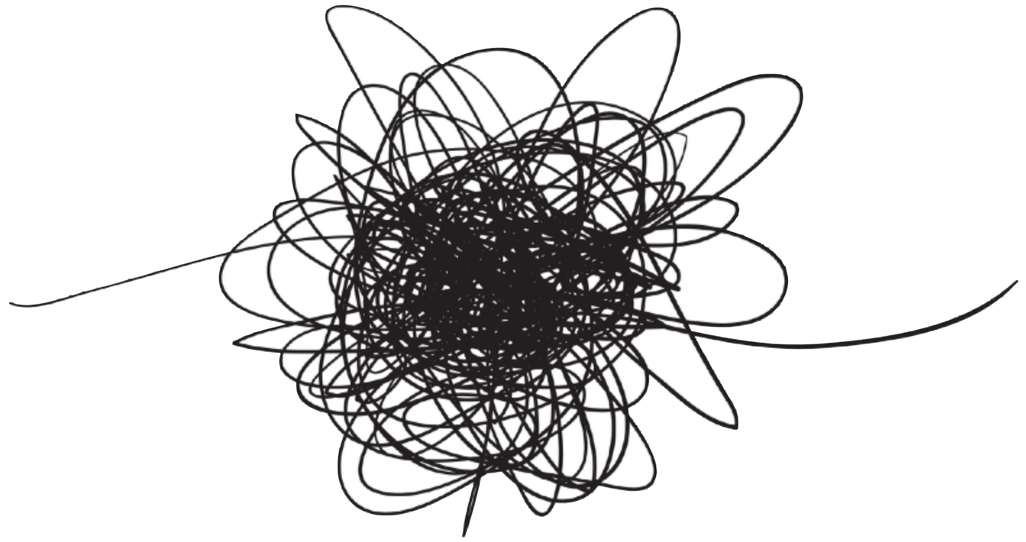
it's what you get when there are multiple paths to the same value.

If you've ever struggled with global variables, you know some of the pain I'm talking about. A global is a just a special case of reference semantics: a value that has a path from everything else.

As a culture, we seem to understand that global variables cause problems, and we're pretty good about weeding them out of our code. But all the same issues apply to pointers and references!



Reference semantics | Technical debt



First, there's an engineering cost; designs become more coupled, more rigid, and more fragile.

Removing a global variable from a program can be hard, but the same goes for anything that's multiply-referenced.

As with global variables, the temptation is always to keep adding data to these objects for convenience, so that you can access it everywhere, and then you're locked into what we call a “reference hub”. If you've ever worked on the LLVM sources, all those "contexts" are a great example.

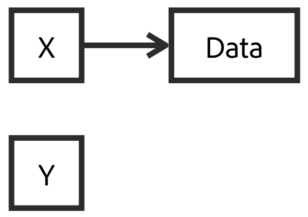
Reference semantics | Spooky action: a Lovecraftian tale

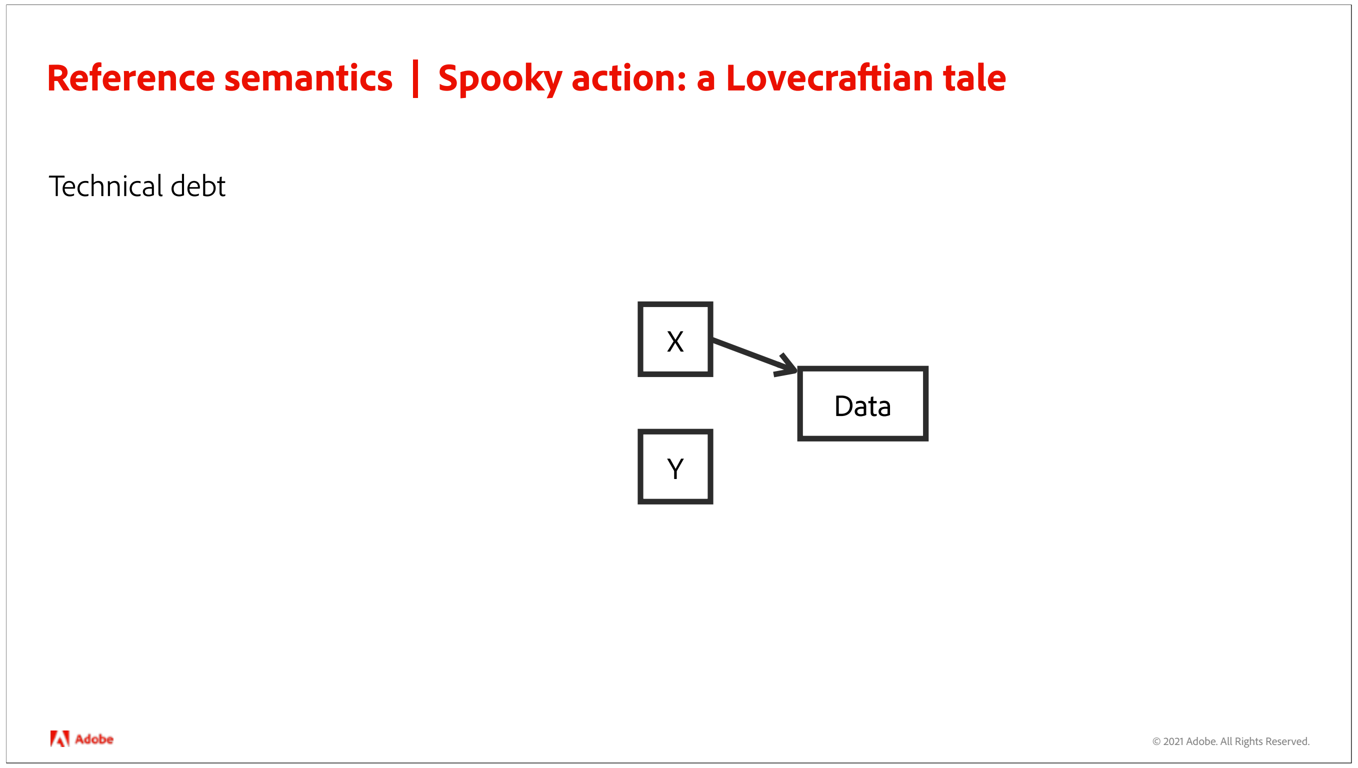
Technical debt

Next, there's implicit sharing. We all know how this looks. X hands Y some piece of data

Reference semantics | Spooky action: a Lovecraftian tale

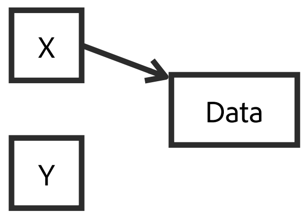
Technical debt





Reference semantics | Spooky action: a Lovecraftian tale

Technical debt

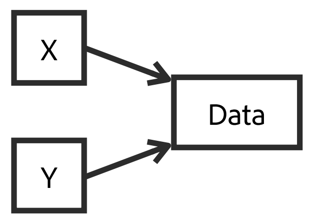


And Y says “thank you kindly; I'll just go about my business.”

And now X's local view of the world

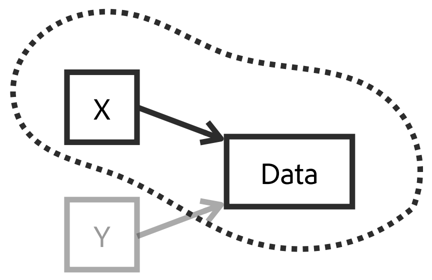
Reference semantics | Spooky action: a Lovecraftian tale

Technical debt

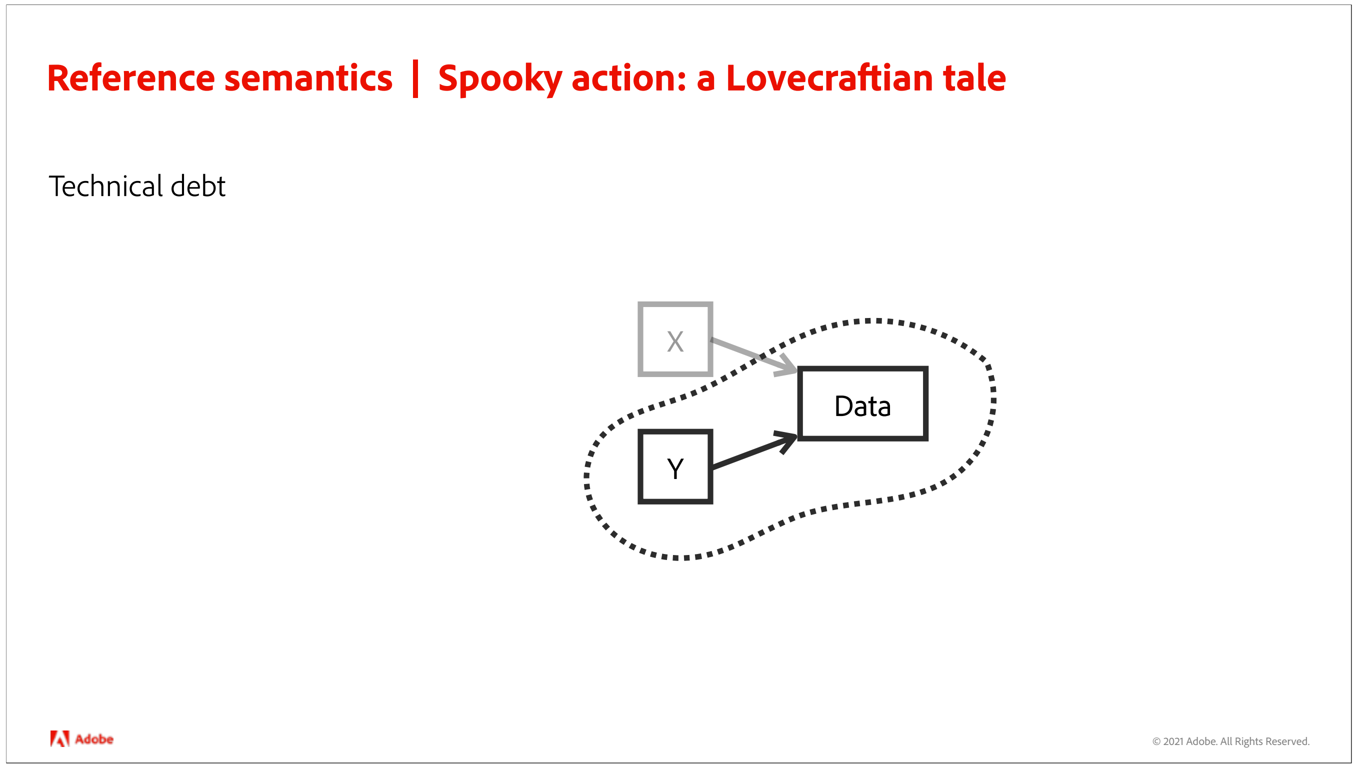


Reference semantics | Spooky action: a Lovecraftian tale

Technical debt

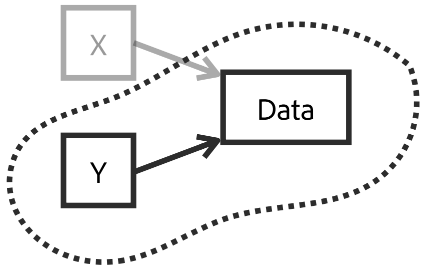


and Y's local view

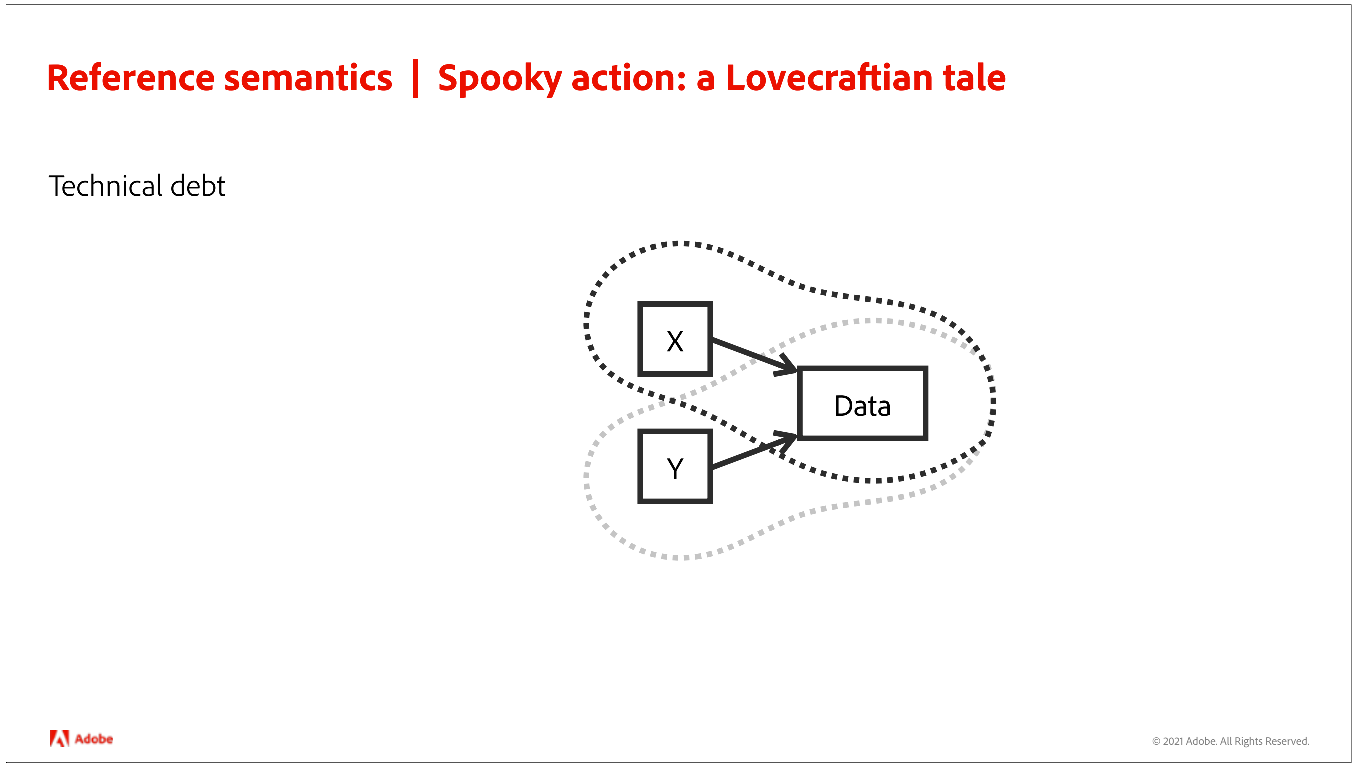


Reference semantics | Spooky action: a Lovecraftian tale

Technical debt

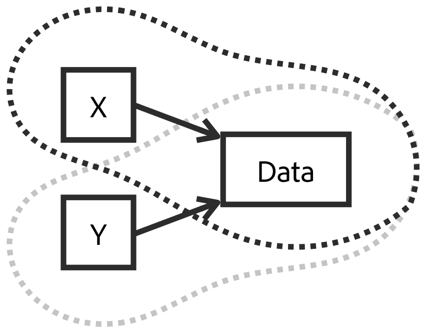


are each perfectly plausible; when this



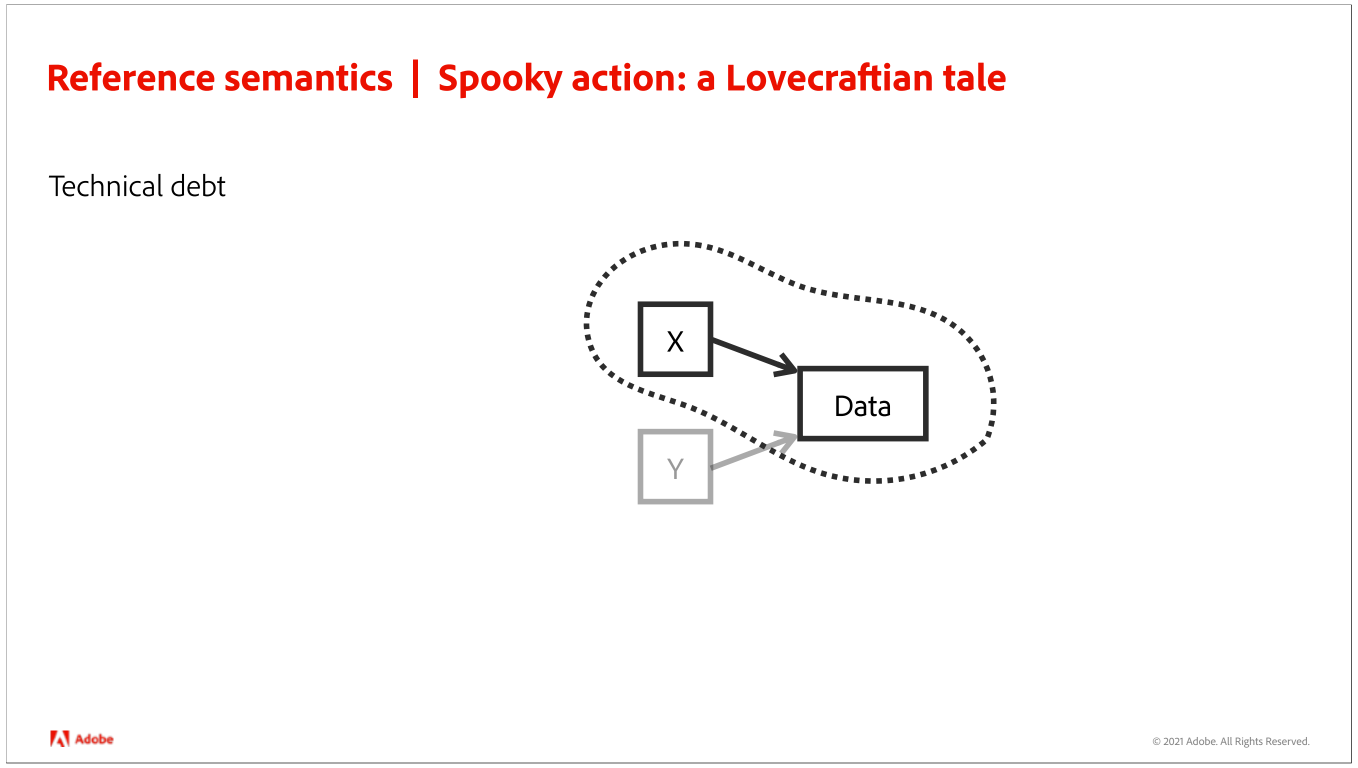
Reference semantics | Spooky action: a Lovecraftian tale

Technical debt



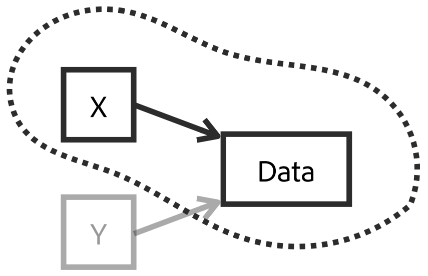
Is the reality.

But it's a happy illusion, and as far as our heroes know, everything is still right with the world



Reference semantics | Spooky action: a Lovecraftian tale

Technical debt

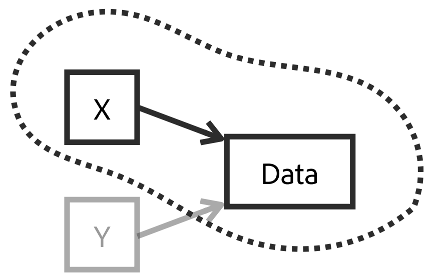


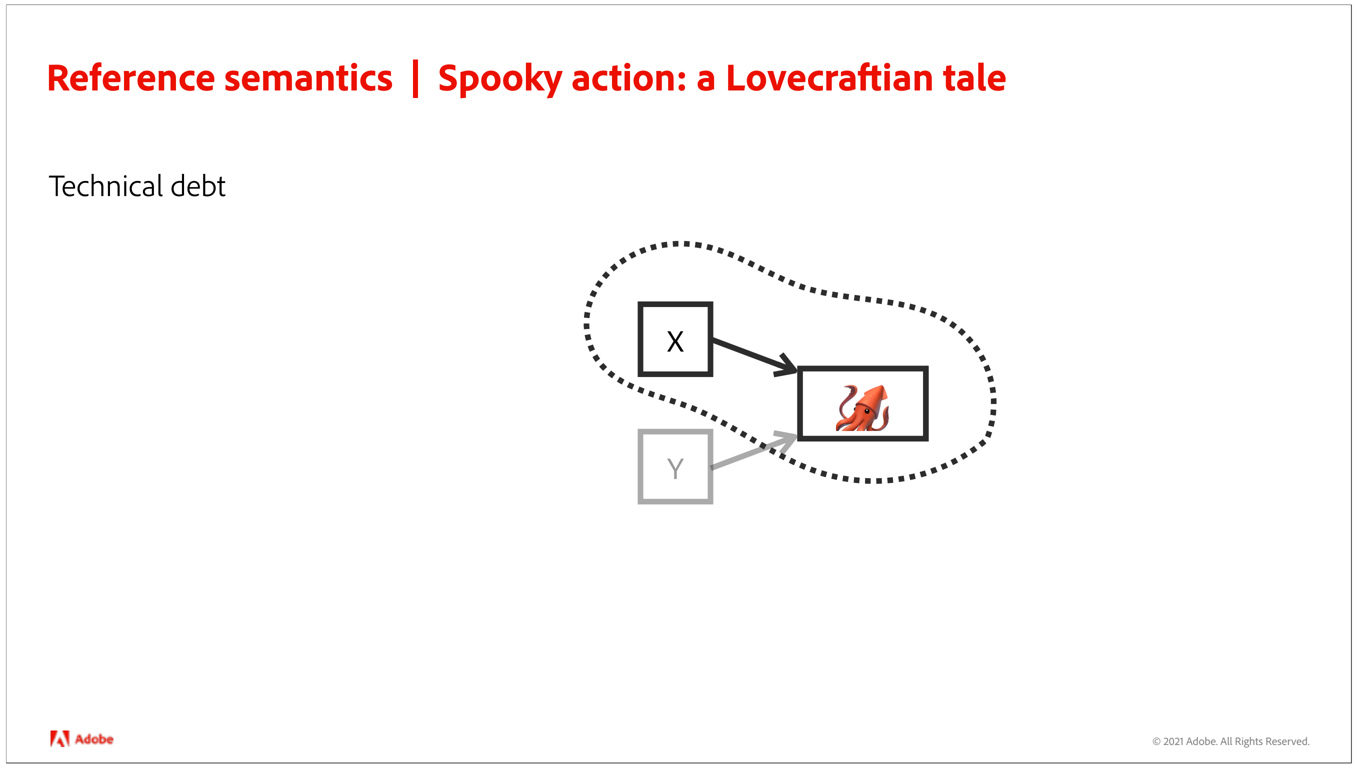
Adobe

Until X, with its local view of reality, decides to replace its data with a giant squid

Reference semantics | Spooky action: a Lovecraftian tale

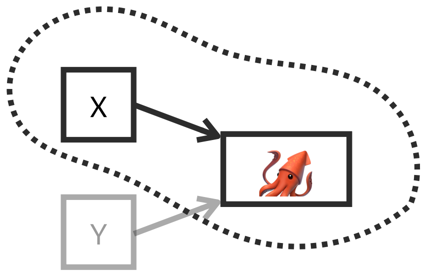
Technical debt





Reference semantics | Spooky action: a Lovecraftian tale

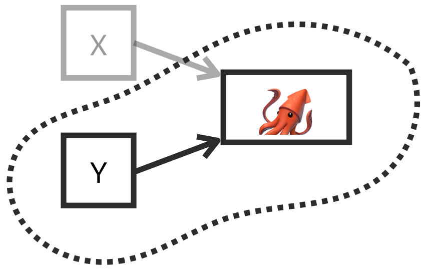
Technical debt



Now, from Y's perspective, nothing has happened

Reference semantics | Spooky action: a Lovecraftian tale

Technical debt



Until it goes to look for its data and finds…
>>monstrous tentacles.

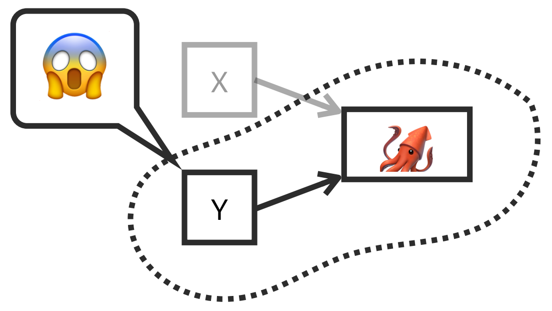
We call this "spooky action at a distance," with apologies to Einstein.

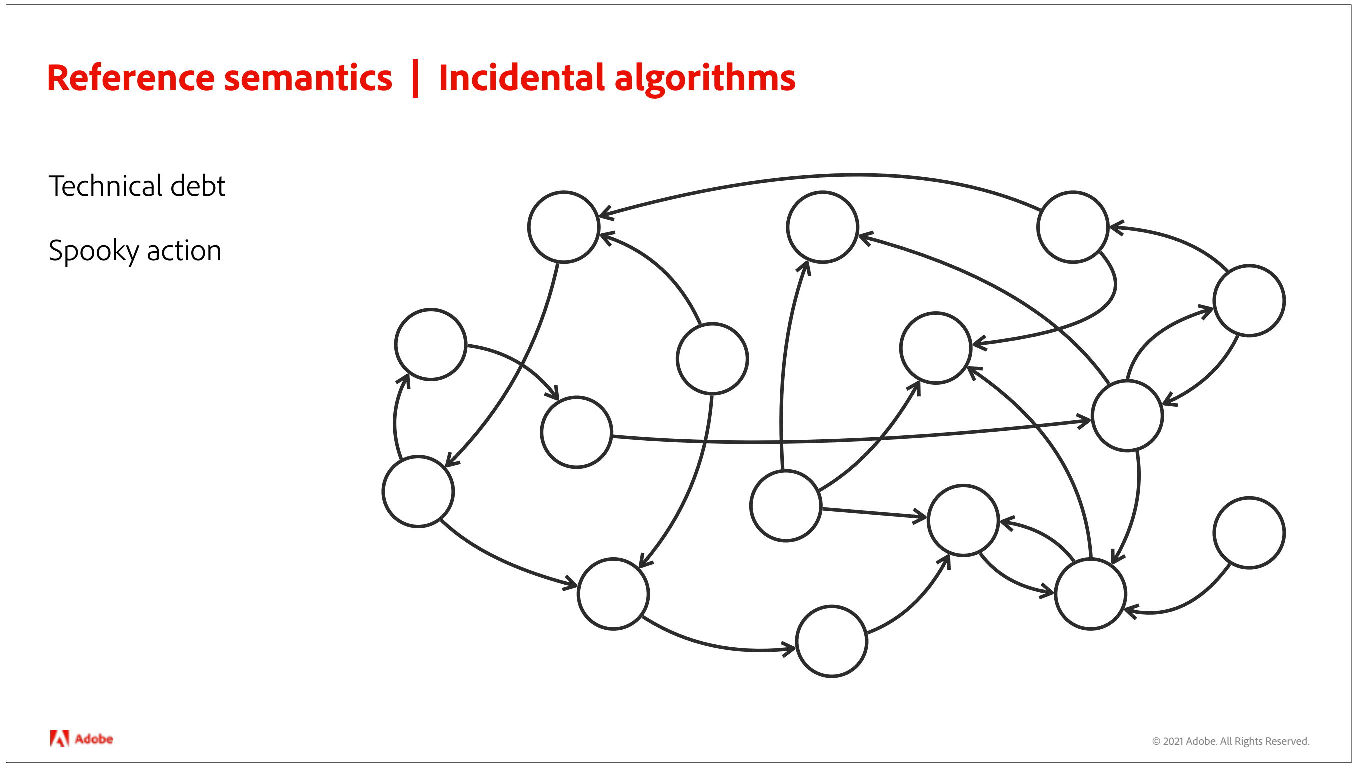
In case you don't believe this problem is real, look for a technique called "defensive copying," which people use to avoid spooky action… but it's inefficient, and, like all patterns, error-prone.

But this is just a simple example

Reference semantics | Spooky action: a Lovecraftian tale

Technical debt

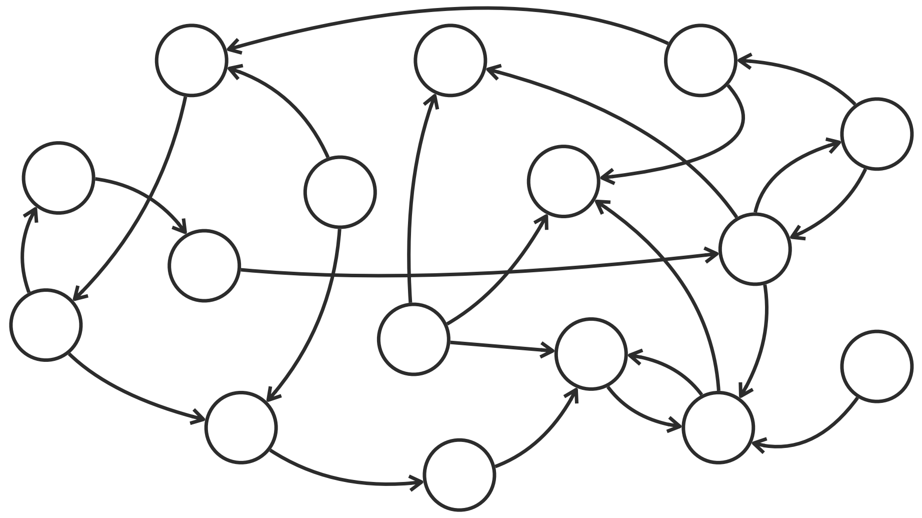




Reference semantics | Incidental algorithms

Technical debt

Spooky action



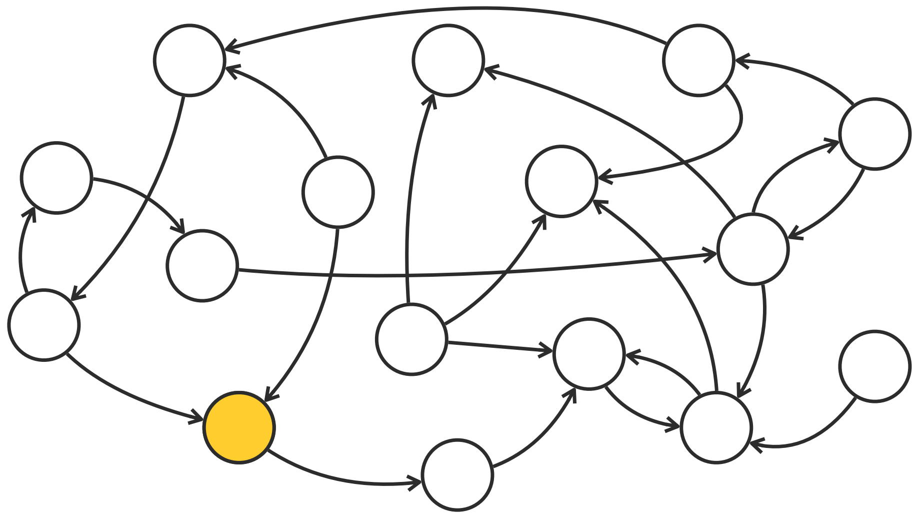
More likely if you have reference semantics, your object graph looks something like this, in which
>> a local action
>> has a whole **set** of nonlocal consequences
Does this picture look like your system? If so, do you have a sense of control over those consequences?

As Sean Parent likes to point out, there's an algorithm encoded in all these interactions.
But, it's hard to understand what's going on, because the code is spread over many dynamically connected objects, instead of being gathered into a function about which you can reason locally.

Reference semantics | Incidental algorithms

Technical debt

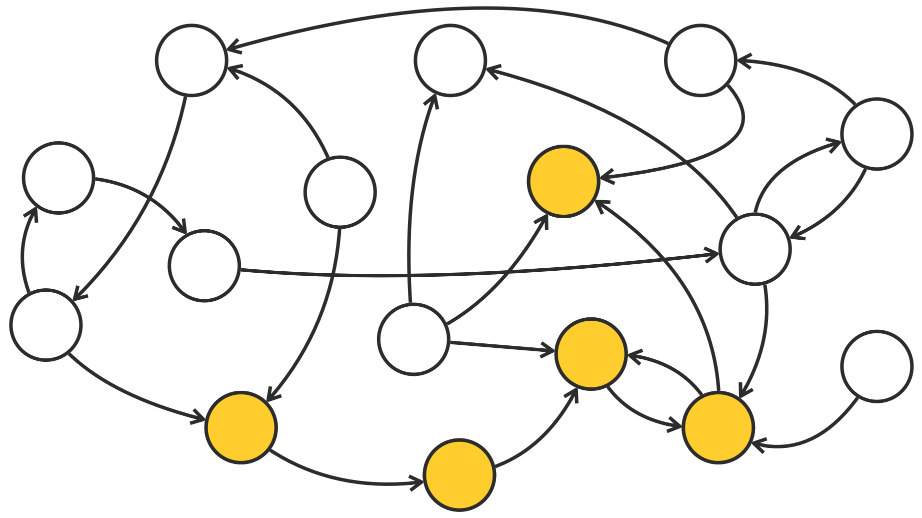
Spooky action

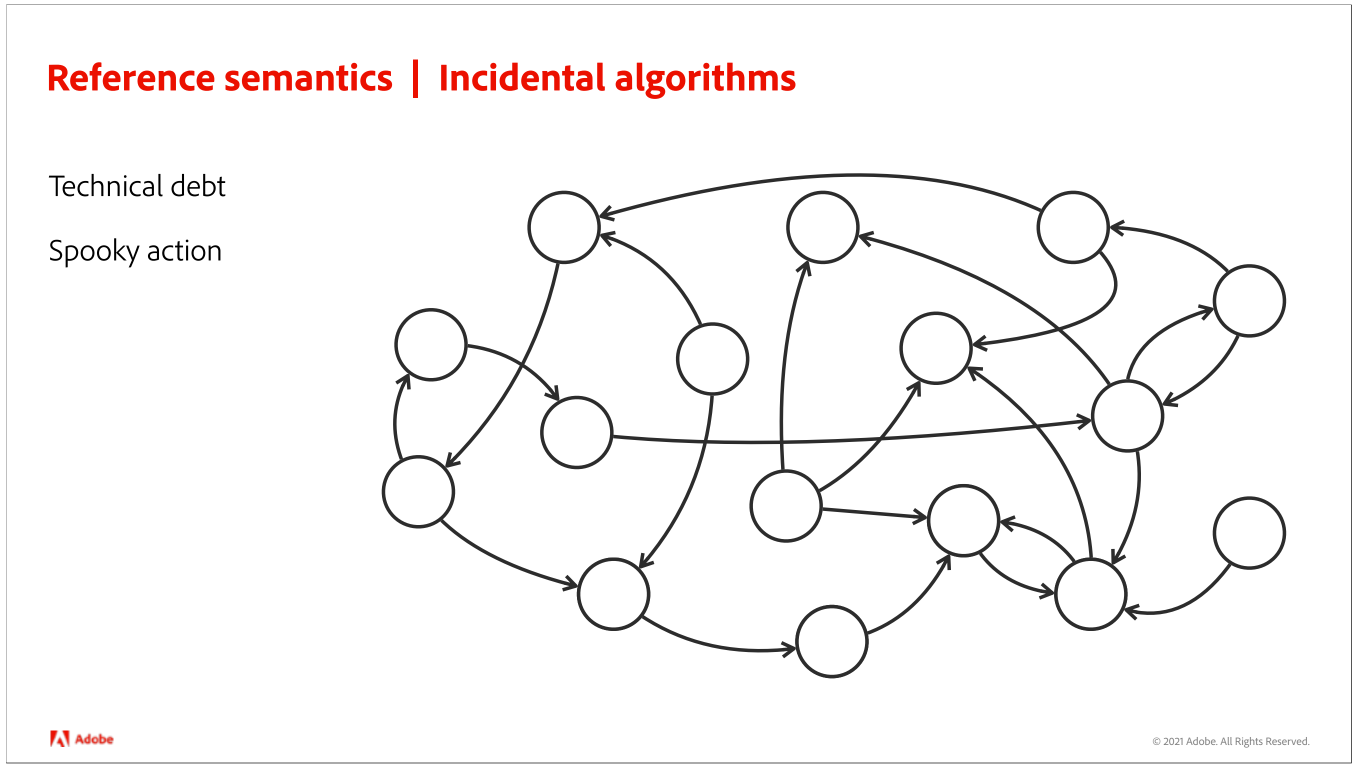


Reference semantics | Incidental algorithms

Technical debt

Spooky action

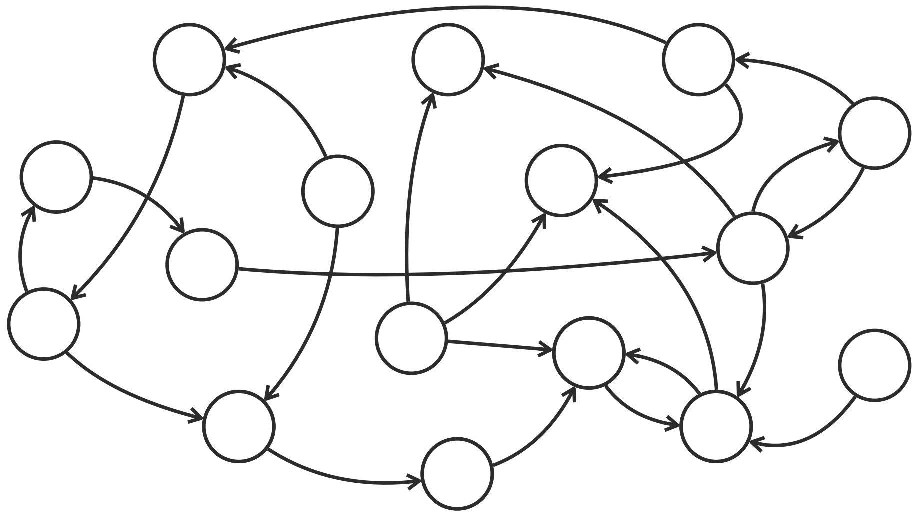




Reference semantics | Incidental algorithms

Technical debt

Spooky action

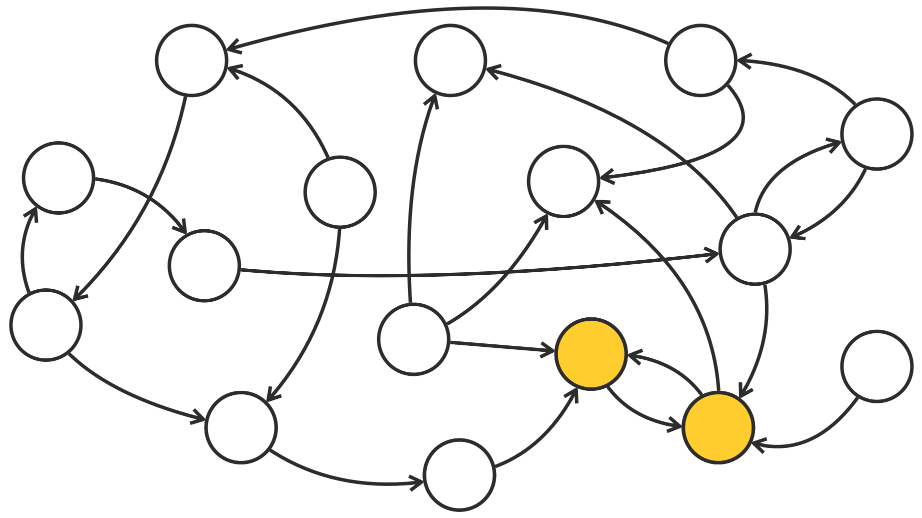


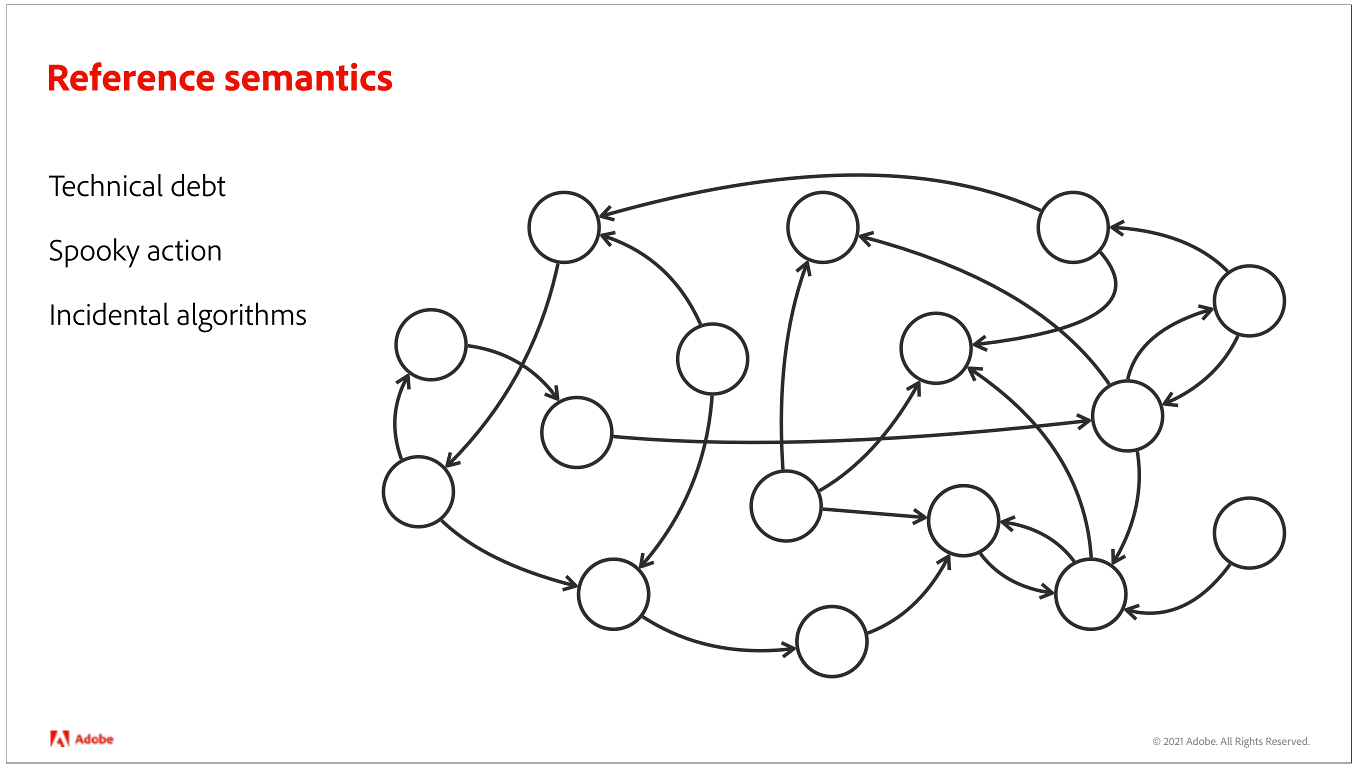
>> It might even go into a loop.

Reference semantics | Incidental algorithms

Technical debt

Spooky action



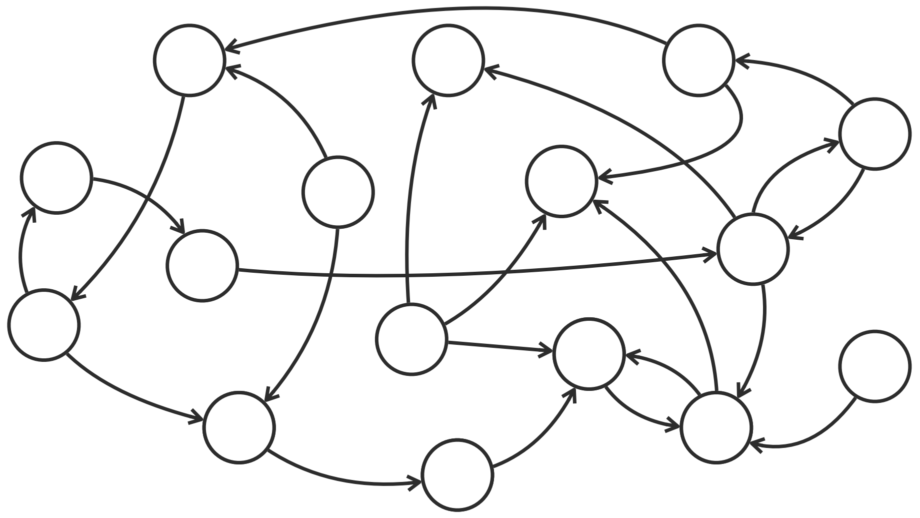


Reference semantics

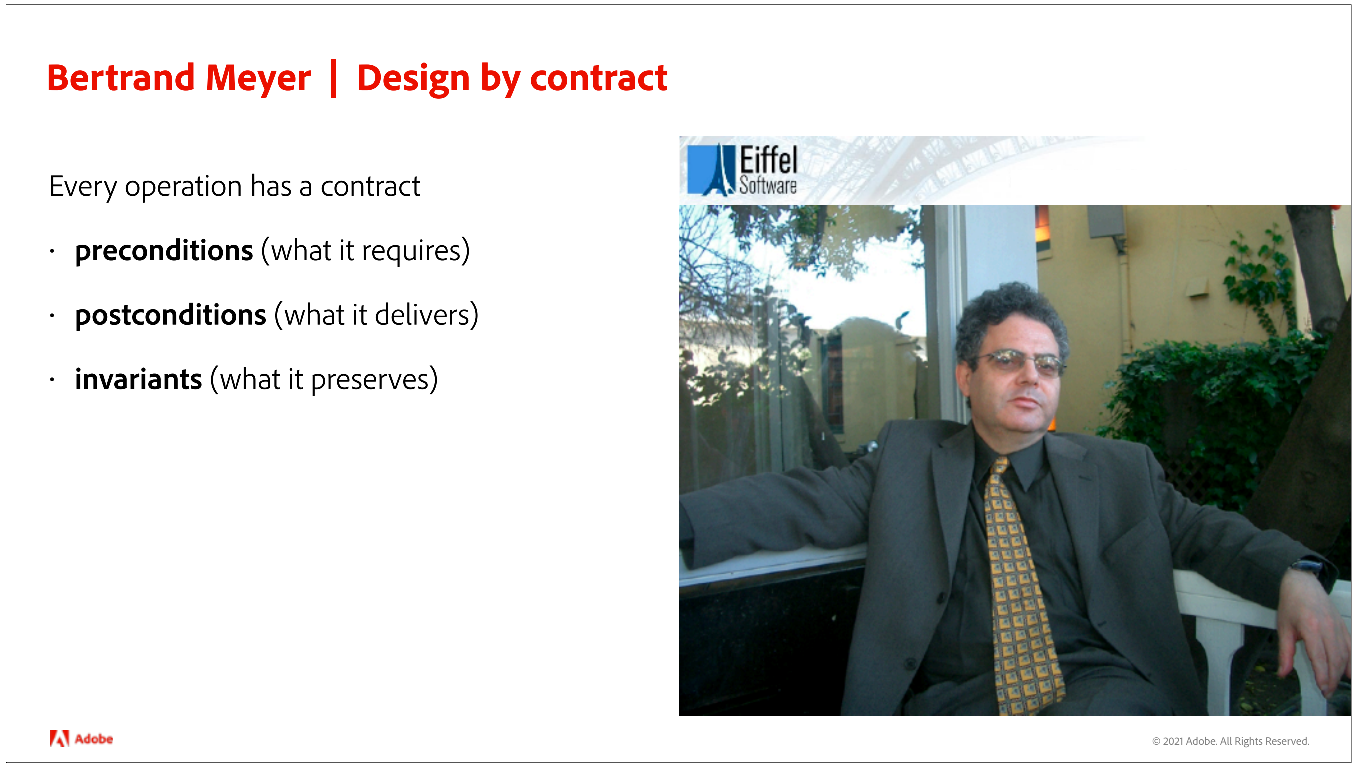
Technical debt

Spooky action

Incidental algorithms



Now, because we need to talk about invariants, and I 💘 the fundamentals, we need to make a brief digression.



Bertrand Meyer | Design by contract

Every operation has a contract

- **preconditions** (what it requires)
- **postconditions** (what it delivers)
- **invariants** (what it preserves)

Adobe

In the mid 1980s, Bertrand Meyer, who has to be the coolest cat in Object Oriented Programming, came up with a discipline called Design by Contract. Some languages like Eiffel and D have features for contract *checking*, but the baseline requirement for this discipline that contracts need to be documented.

In design by contract, a function's documentation covers at least three things:

- preconditions (what's required from the caller)
- postconditions (what's done and/or returned by the callee)
- invariants (conditions that the function preserves)

Having these three things allows clients to use it without diving into the function's implementation to see how it works.

It's required for correct software bigger than a few files, and if you correct software at scale, even if you don't use these words, you're using this discipline.

Invariants also extend to types
>> And every type has its own invariant, that's preserved by all operations on that type.
My favorite example is this type that holds two vectors, and the invariant
>> is that the vectors always have the same length. Maybe it's a container that presents pairs of x and y as elements, where storing actual pairs could waste lots of memory due to alignment and padding.

An invariant <u>always</u> has to hold for the program to be correct, with one exception: during a mutation. If we want to add a new pair, we have to grow one of these vectors first,

>> which breaks the invariant until we've done the other push_back.
>> That's not a problem because the vectors are private and we encapsulate the invariant inside a mutating method, that appends a pair. By the time that method returns, everything is back in order.

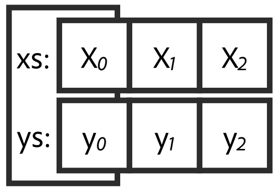
But let's see what happens in our object graph:

Bertrand Meyer | Design by contract

Every operation has a contract

- **preconditions** (what it requires)
- **postconditions** (what it delivers)
- **invariants** (what it preserves)

Every type has a **class invariant**

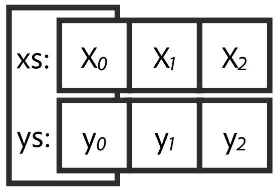


Bertrand Meyer | Design by contract

Every operation has a contract

- **preconditions** (what it requires)
- **postconditions** (what it delivers)
- **invariants** (what it preserves)

Every type has a **class invariant**



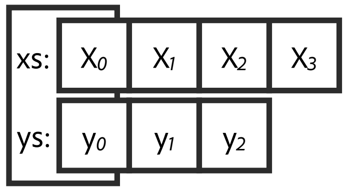
invariant: `xs.size == ys.size`

Bertrand Meyer | Design by contract

Every operation has a contract

- **preconditions** (what it requires)
- **postconditions** (what it delivers)
- **invariants** (what it preserves)

Every type has a **class invariant**



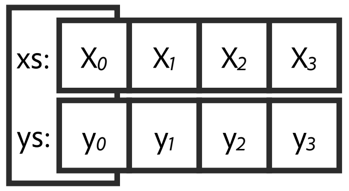
invariant: `xs.size` $\cancel{=}$ `ys.size`

Bertrand Meyer | Design by contract

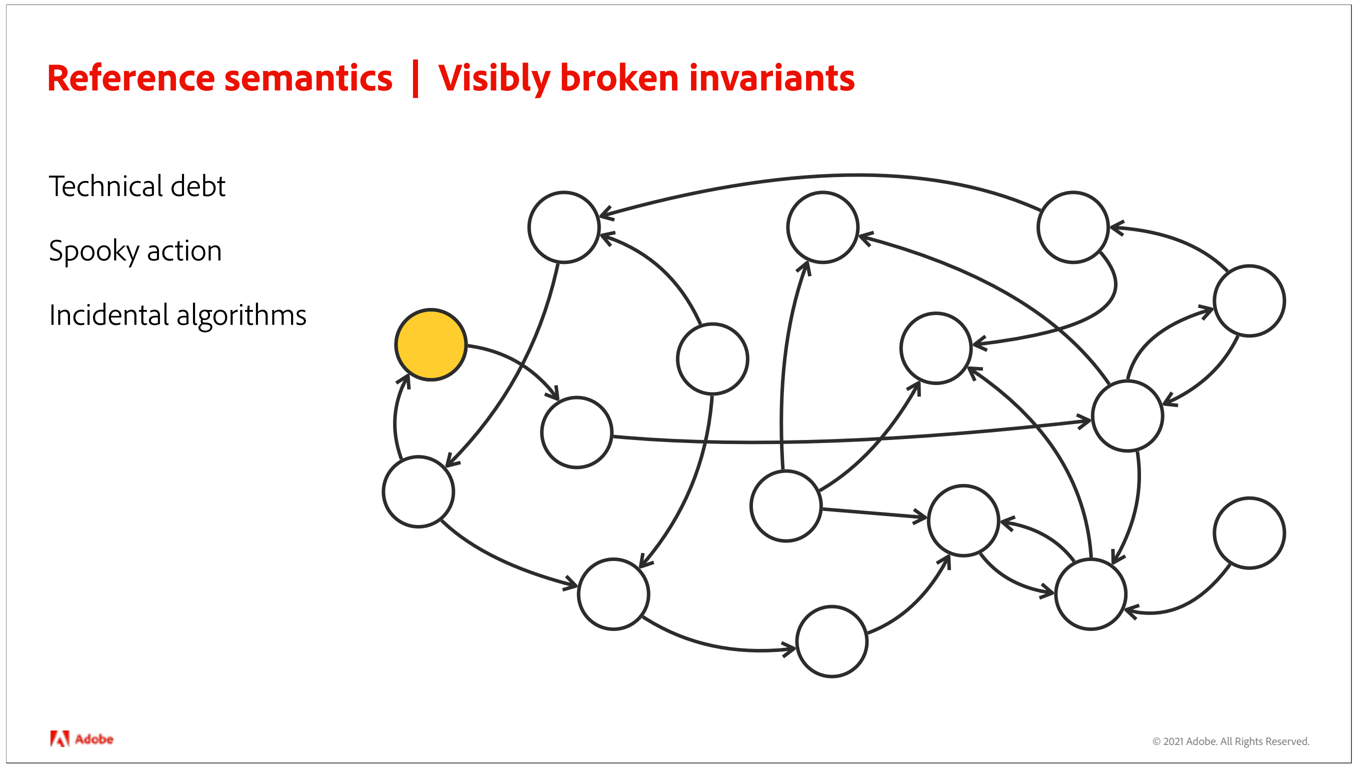
Every operation has a contract

- **preconditions** (what it requires)
- **postconditions** (what it delivers)
- **invariants** (what it preserves)

Every type has a **class invariant**



invariant: `xs.size == ys.size`



Reference semantics | Visibly broken invariants

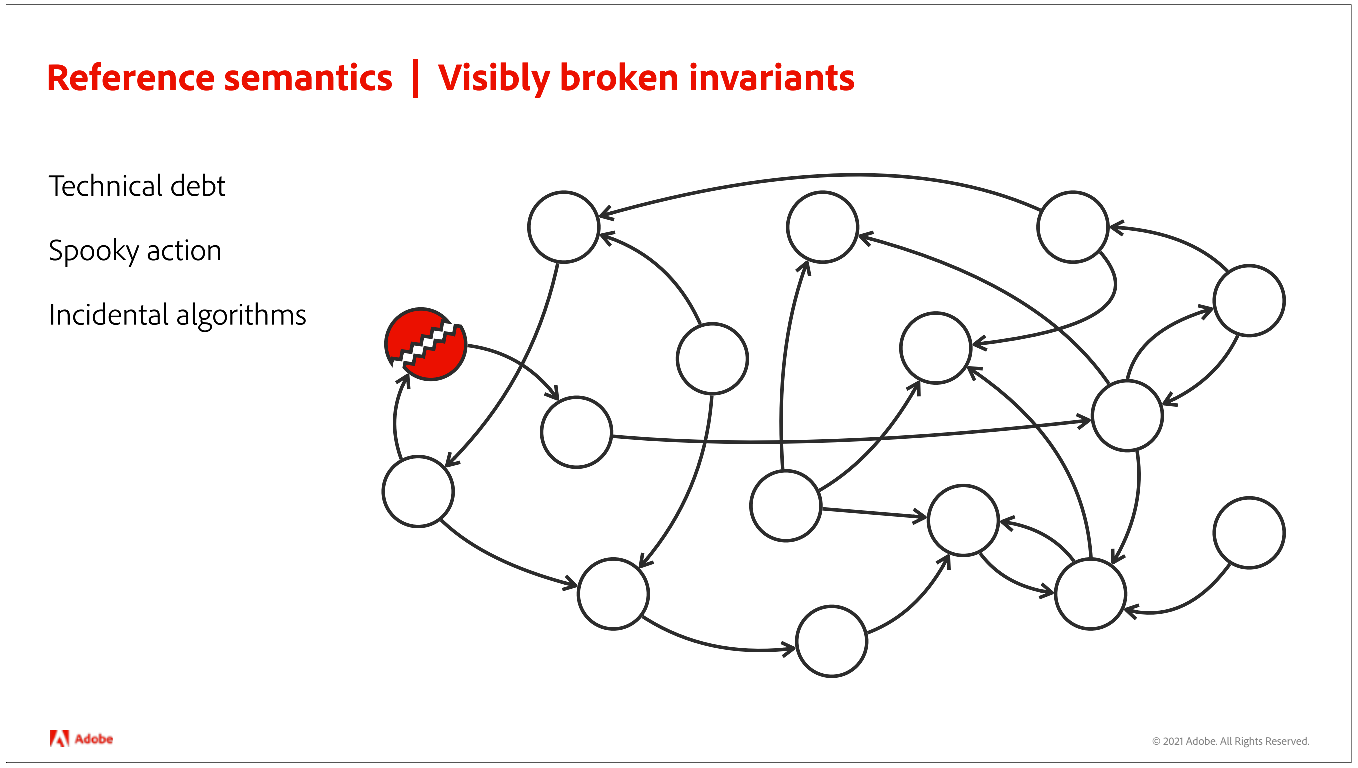
Technical debt

Spooky action

Incidental algorithms

Adobe

Say we're mutating the yellow object.

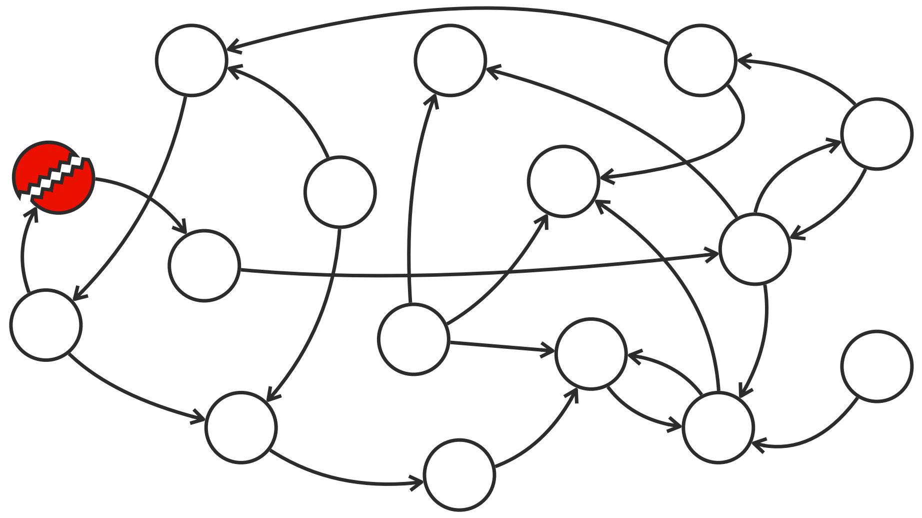


Reference semantics | Visibly broken invariants

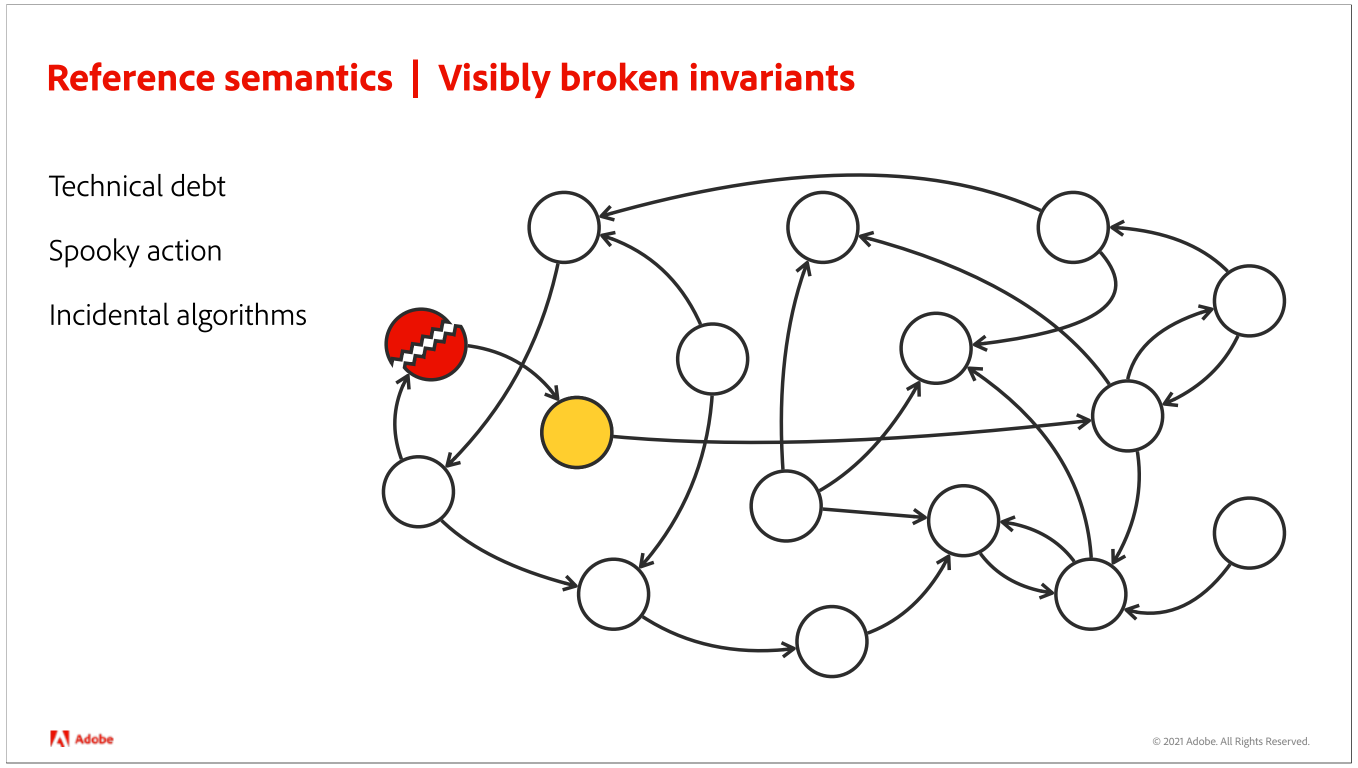
Technical debt

Spooky action

Incidental algorithms



If part of that mutation requires using another object

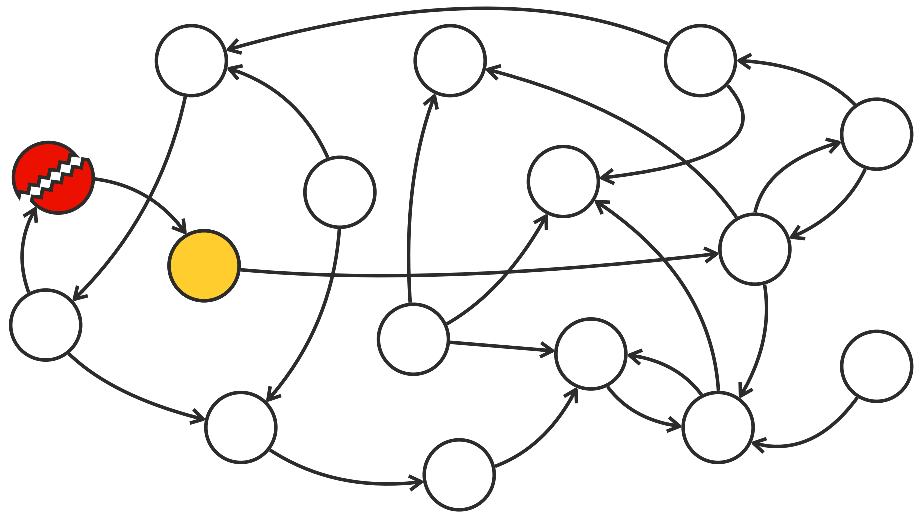


Reference semantics | Visibly broken invariants

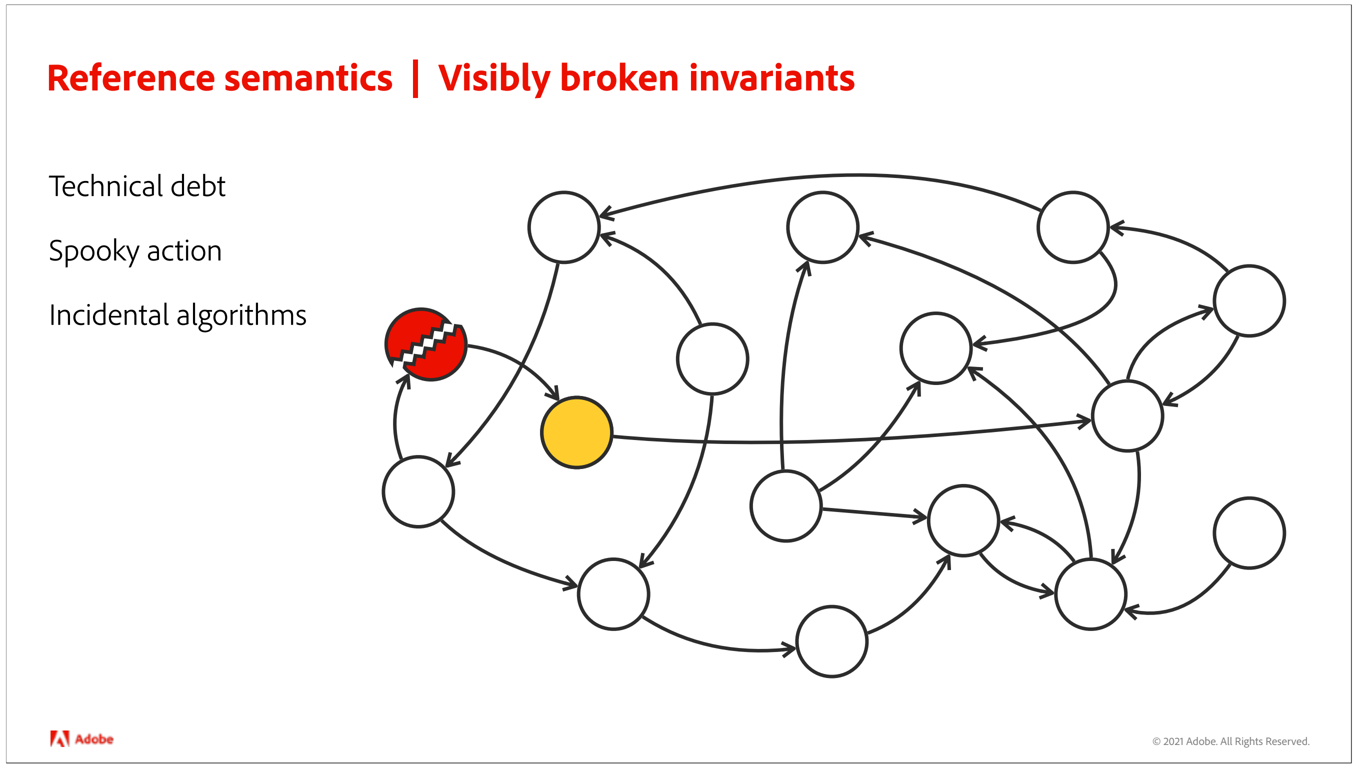
Technical debt

Spooky action

Incidental algorithms



Adobe

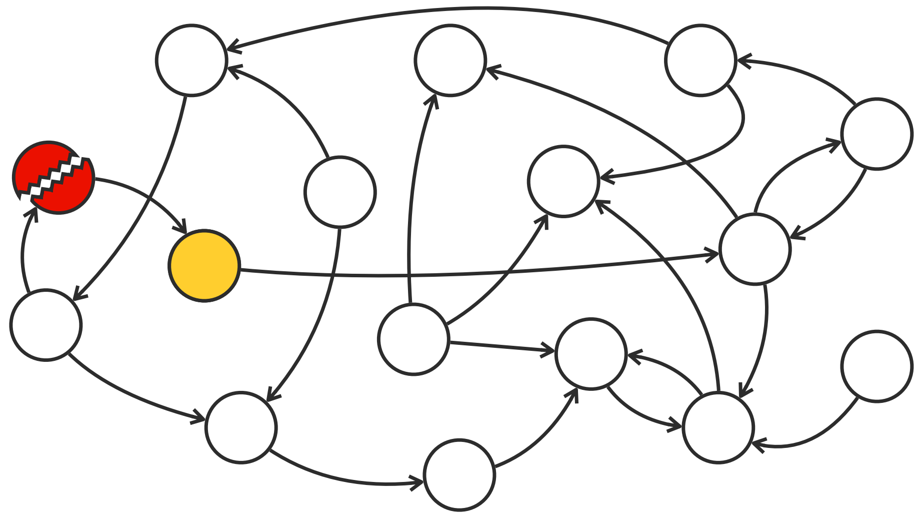


Reference semantics | Visibly broken invariants

Technical debt

Spooky action

Incidental algorithms



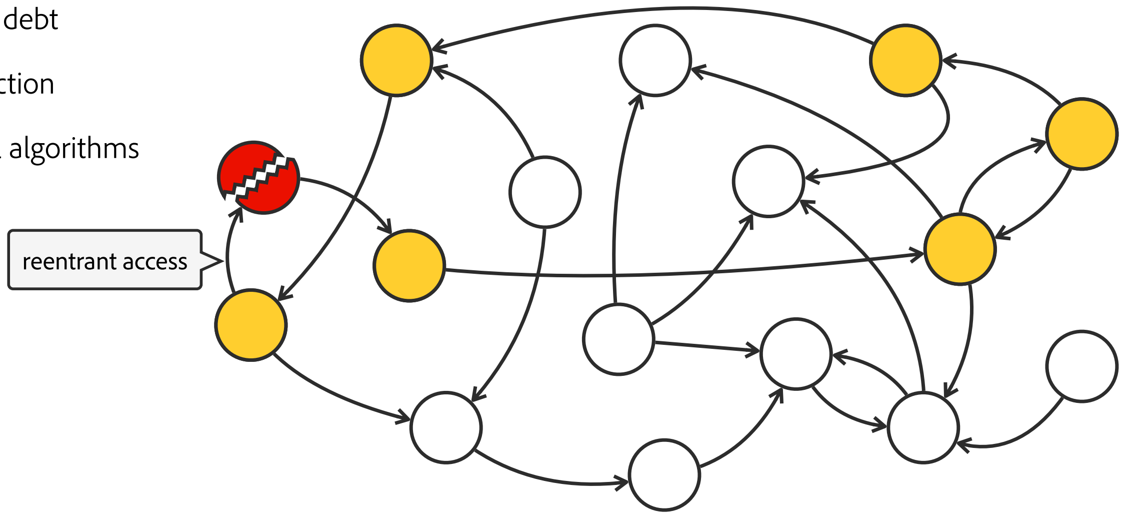
The chain can come back to observe the original object with its invariants broken.

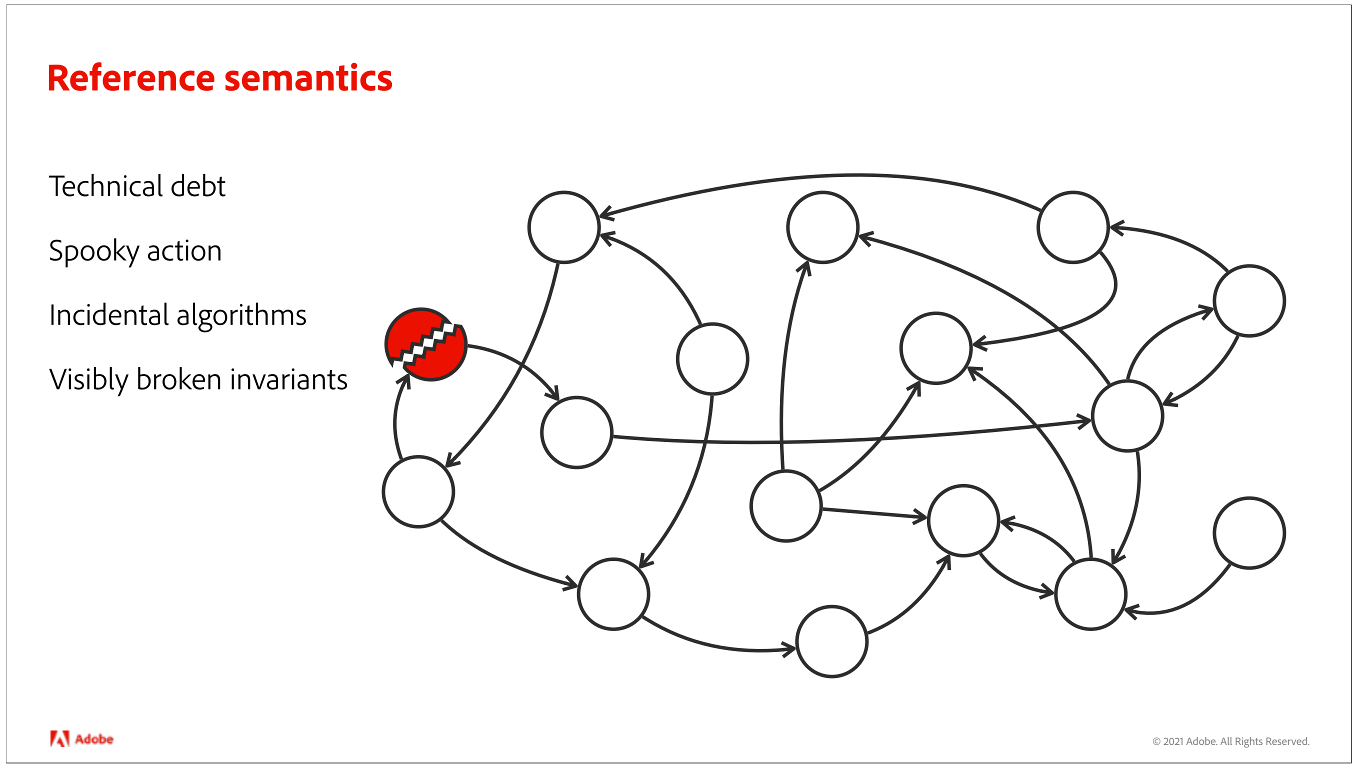
Reference semantics | Visibly broken invariants

Technical debt

Spooky action

Incidental algorithms





Reference semantics

Technical debt

Spooky action

Incidental algorithms

Visibly broken invariants

And then race conditions, which are just what we call this problem when the accesses are in different threads. Remember, references bring all the same problems as global variables.
We all know the future is multicore and yet we keep building structures like this one, which are fundamentally hostile to concurrency.

>> Then there's the way, even if we have an immutable pointer or reference, mutation can still occur behind our backs.

And if all these practical problems weren't enough for you, there's actually a deeper one.

Reference semantics

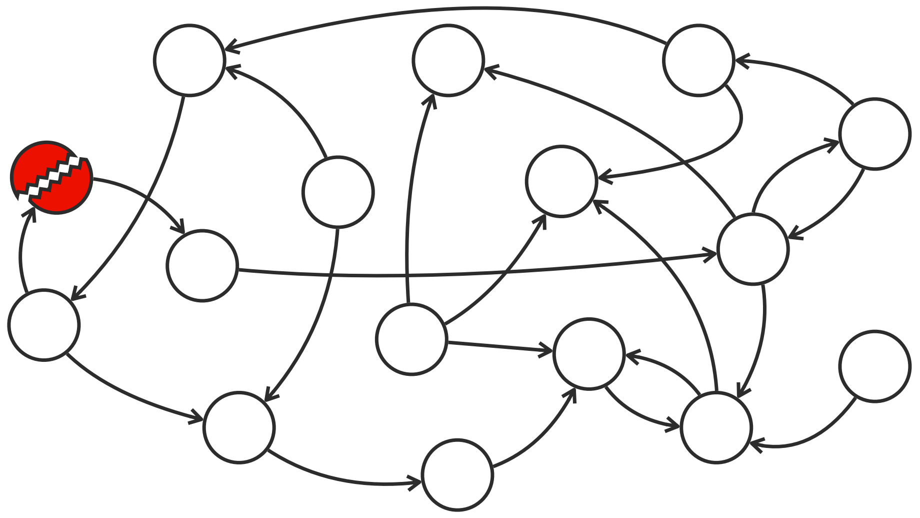
Technical debt

Spooky action

Incidental algorithms

Visibly broken invariants

Race conditions



Reference semantics

Technical debt

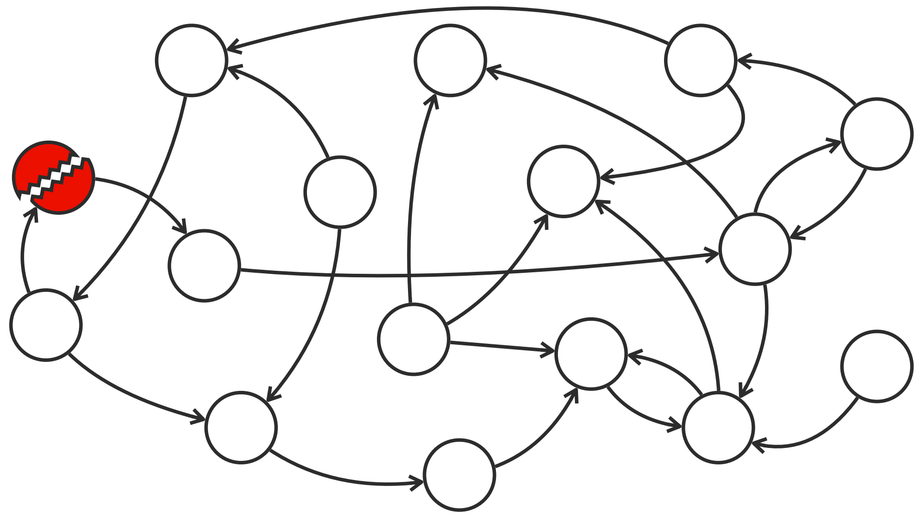
Spooky action

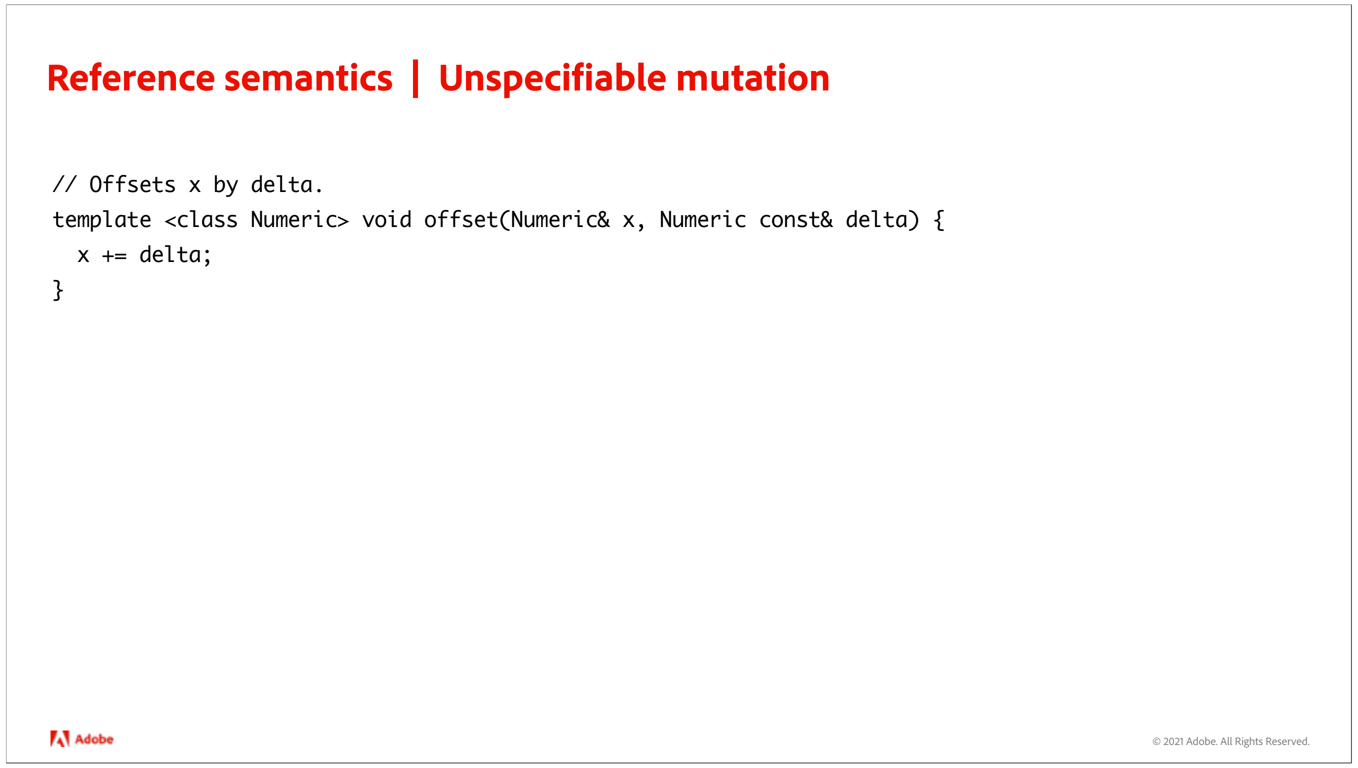
Incidental algorithms

Visibly broken invariants

Race conditions

Surprise mutation





Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}
```

Take a look at this example; it's the simplest one I've found so far.

I've got a generic function to offset one numeric value by another. We want it to work efficiently on floats and on arrays of floats, so we're passing delta by reference. Can we all agree that this should be considered correct code?   I promise you this isn't a trick; there's nothing up my sleeve.

>> OK, now I'm going to write a similar function that uses the first one. This might not be the most efficient implementation, but it's equally correct.

>> Finally, as a sanity check, let's test this on an int. What does this print, anybody? >>

If everything we did was correct so far, it should print 9, but it prints 12. Why? Because every time we offset x, we're also offsetting delta.

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}
```

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

3 + 2*3 == ?

Adobe

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

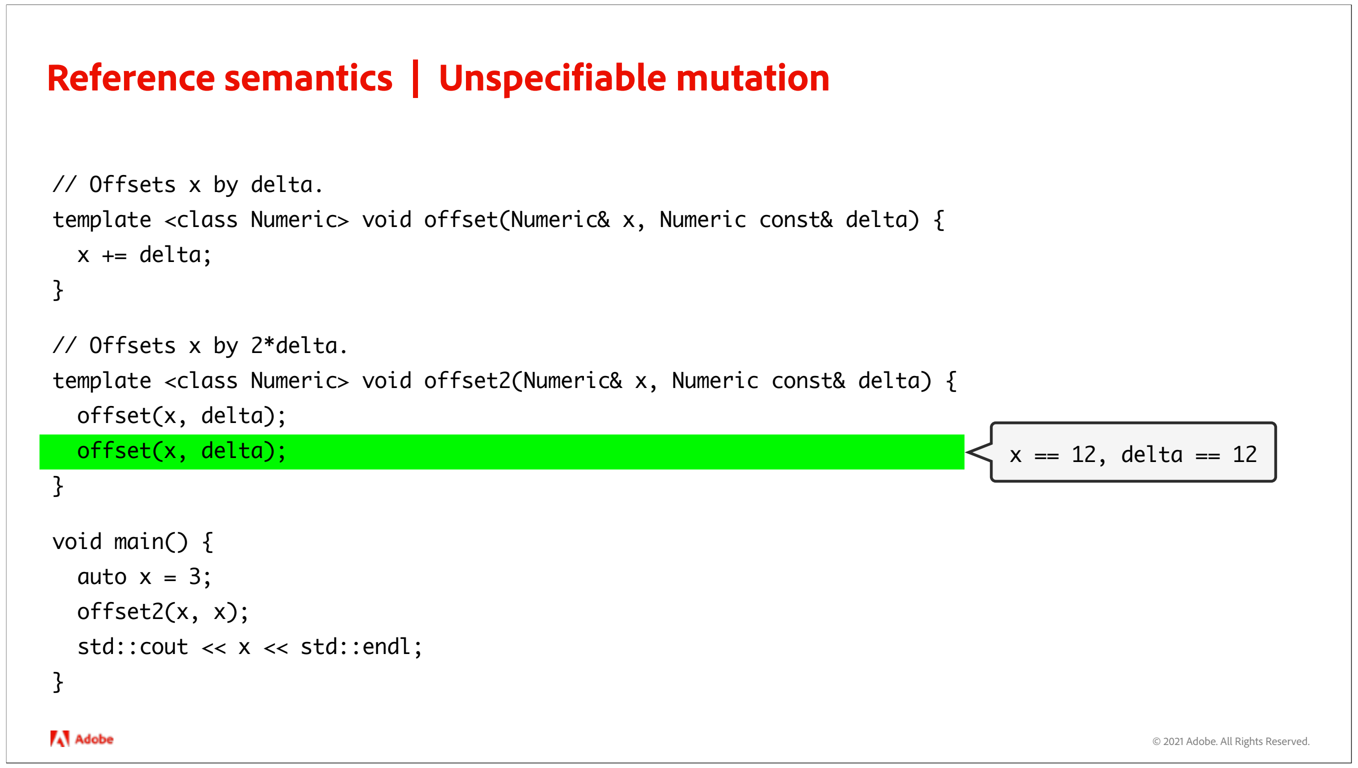
```
x == 3, delta == 3
```

Adobe

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

```
x == 6, delta == 6
```



Reference semantics | Unspecifiable mutation

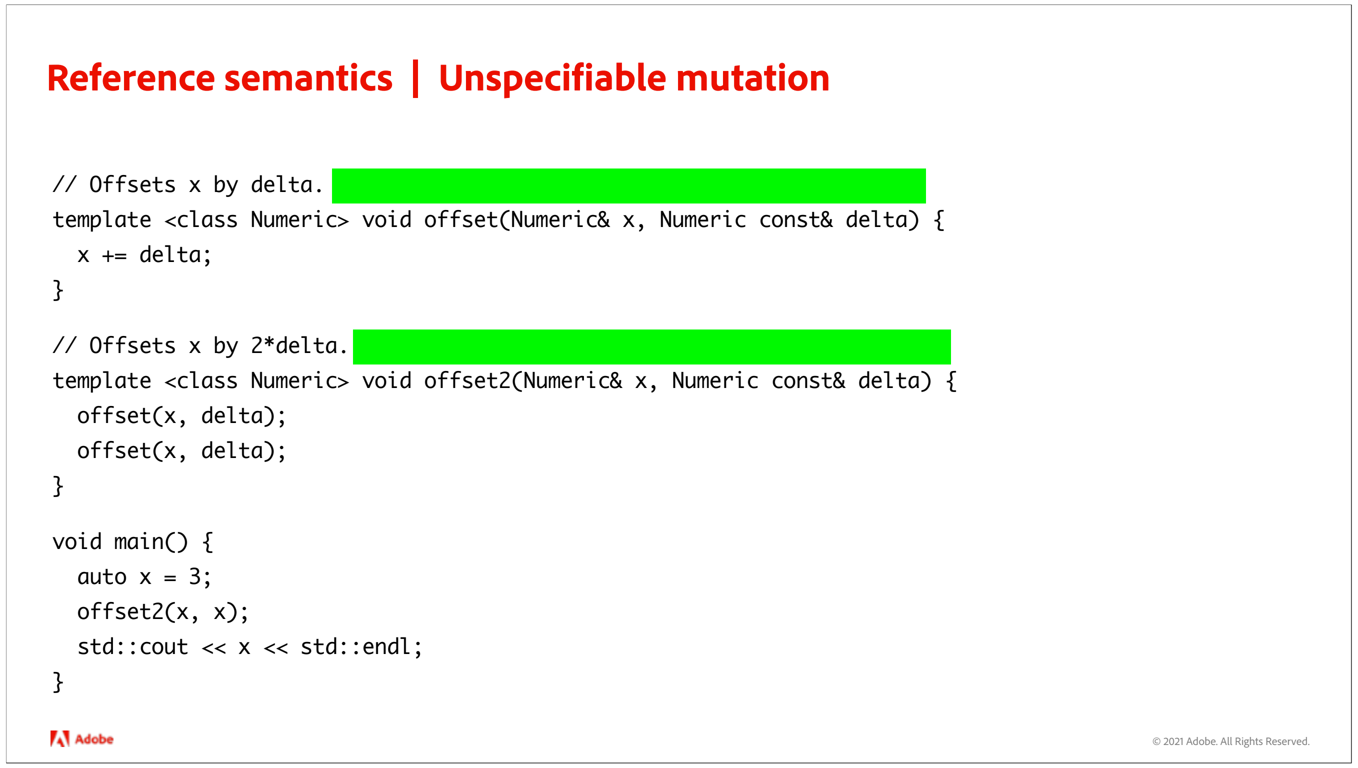
```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

```
x == 12, delta == 12
```

So something is very wrong. Even if you think offset and its documentation are OK, it clearly doesn't compose into offset2 the way functions are supposed to.

How would you change the description of our offset functions' effects—their postconditions—to account for this case?

I don't think you can, at least not without making the description so complicated that it becomes unusable.



Reference semantics | Unspecifiable mutation

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

Adobe

In fact, I think the very best we can do is to add this precondition.

This is not a unique example. >>

Klaus showed the same problem using a different example in his back to basics talk, and the issue is everywhere in the standard. The effects of this replace call are unspecified, although the standard doesn't come right out and say so. As one expert said to me, if the inputs overlap, “you get what you get.”

Now let's update our list and talk about safety for a moment

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

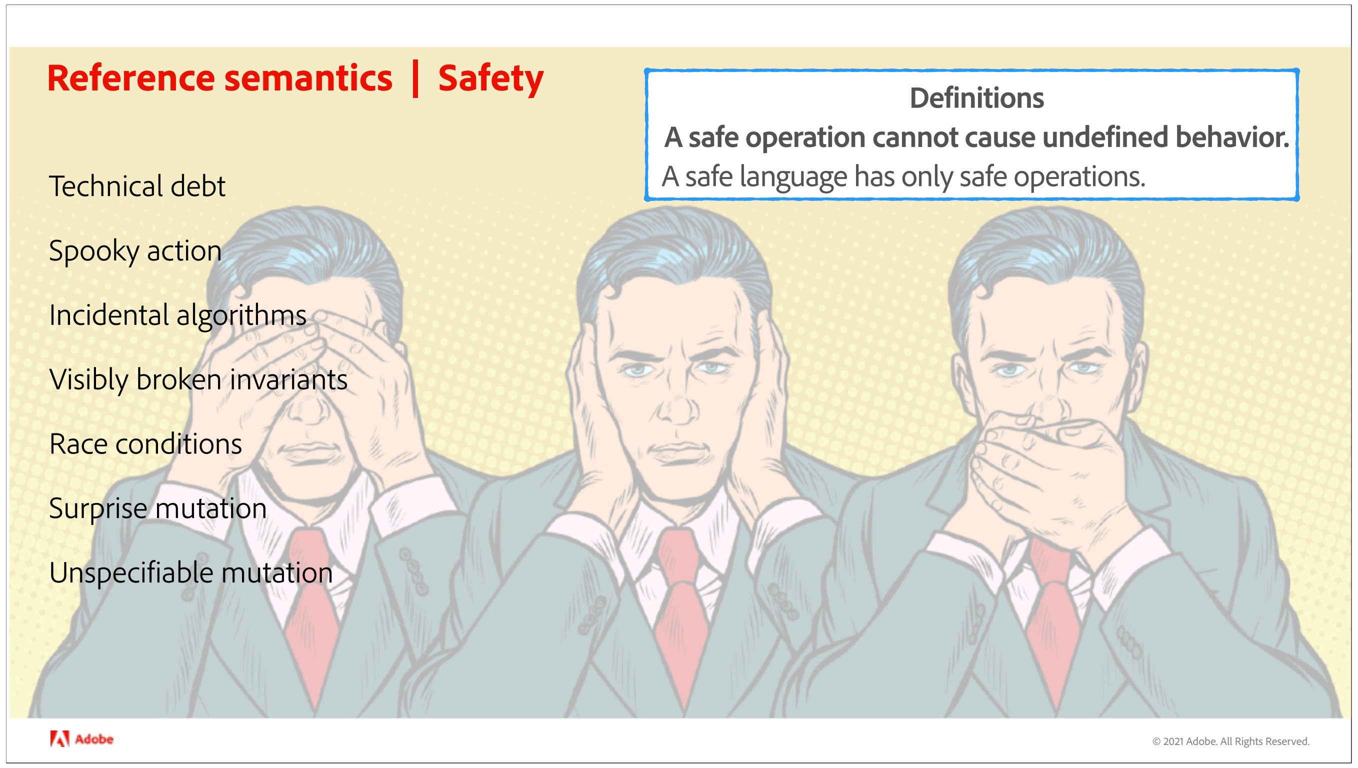
Reference semantics | Unspecifiable mutation

```
// Offsets x by delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

See Also

Klaus Iglberger, *Back to Basics: Value Semantics* CPPCon2022

```
std::replace(v.begin(), v.end(), v[3], v[2]);
```



Reference semantics | Safety

Definitions

A safe operation cannot cause undefined behavior.

A safe language has only safe operations.

- Technical debt
- Spooky action
- Incidental algorithms
- Visibly broken invariants
- Race conditions
- Surprise mutation
- Unspecifiable mutation

Next, reference semantics causes safety problems, by which I mean something very specific: A safe operation is one that cannot cause undefined behavior.

There are other ways you *could* define safety, but this one is consistent with the usual expectation, that if we violate safety in C++ we get undefined behavior, and maps well onto what people mean when they say a programming language is safe.

By this definition C++ isn't a safe language, but there are safe operations, like incrementing an unsigned integer.

Safety matters for two reasons. First, even the best programmers occasionally make mistakes. If their mistake used safe operations, the possible damage is bounded: either the program has stopped (e.g. because the error was detected) or it's executing some code that was actually in the program. With an unsafe operation, the program can do literally anything.

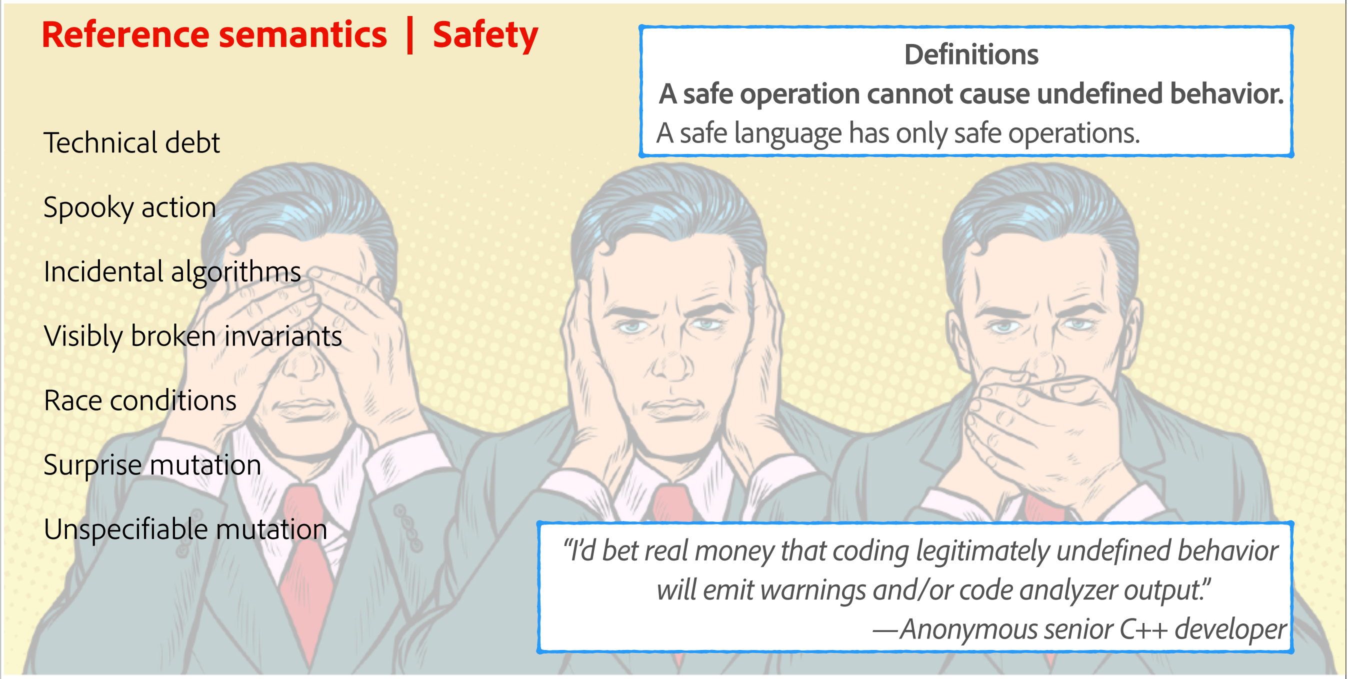
Second, painting with a broad brush, many people can't come to grips with undefined behavior. They don't know what it means, where it comes from, how likely it is, or what the consequences are. Let me show you what I mean>>

This was said by an experienced, senior C++ programmer in all seriousness.

When I said that undetectability by code analyzers is actually a reason undefined behaviors exist in the first place, they said I was just being pedantic.

My point here is, even if you know better, lots of people don't, and don't *want to*. If you work with other people on software, you want safe operations.

Reference semantics | Safety



Technical debt

Spooky action

Incidental algorithms

Visibly broken invariants

Race conditions

Surprise mutation

Unspecifiable mutation

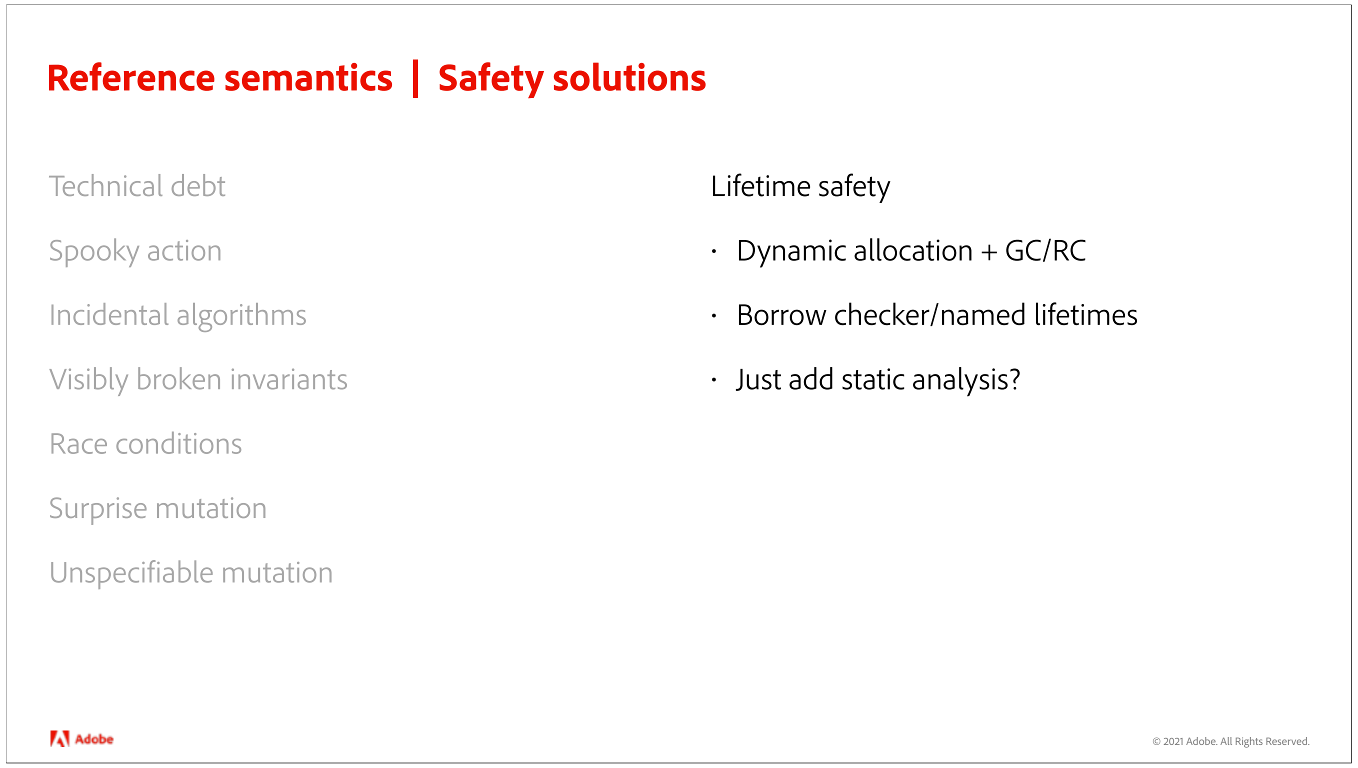
Definitions

A safe operation cannot cause undefined behavior.

A safe language has only safe operations.

"I'd bet real money that coding legitimately undefined behavior will emit warnings and/or code analyzer output."

—Anonymous senior C++ developer



Reference semantics | Safety solutions

Technical debt

Spooky action

Incidental algorithms

Visibly broken invariants

Race conditions

Surprise mutation

Unspecifiable mutation

Lifetime safety

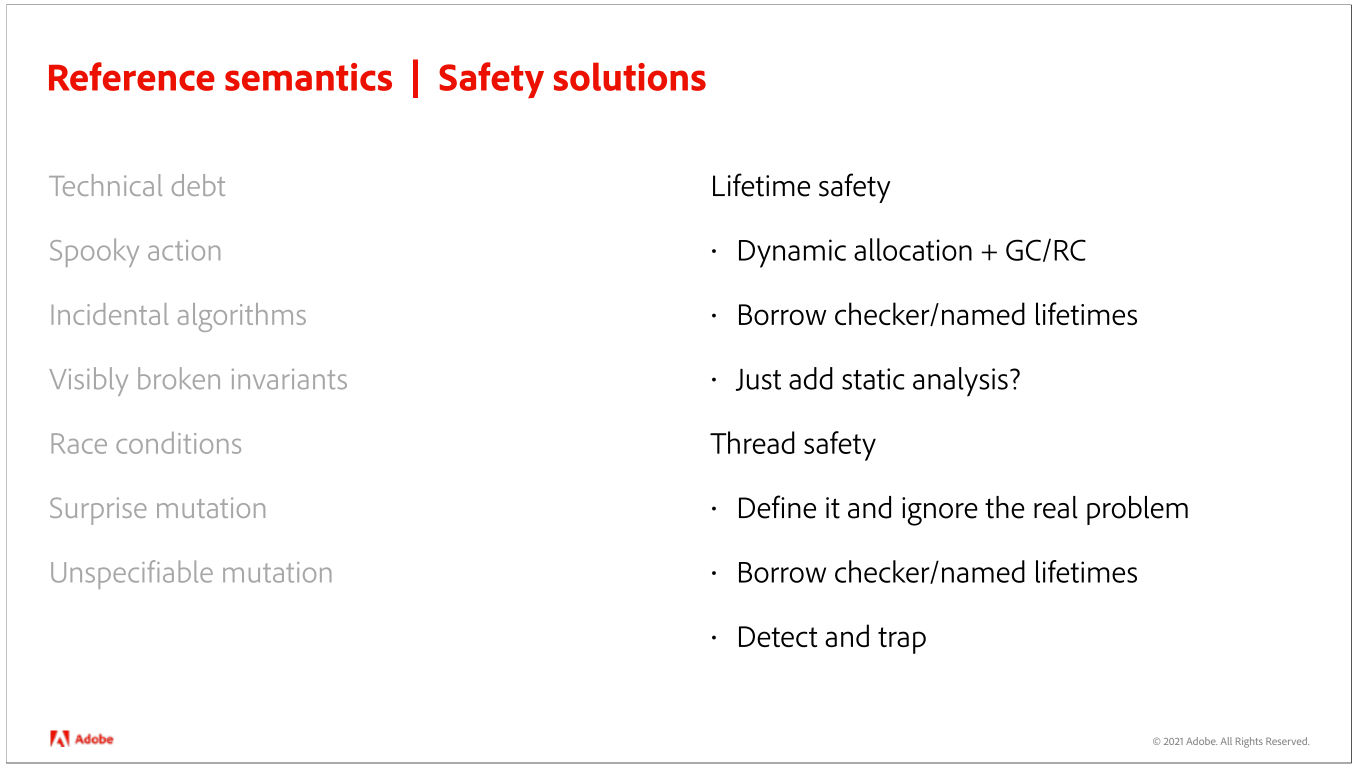
- Dynamic allocation + GC/RC
- Borrow checker/named lifetimes
- Just add static analysis?

The two main safety problems associated with reference semantics are lifetime errors and race conditions, and solutions do exist, which I'll go through very quickly.

The classic approach is to allocate everything on the heap, and then use a garbage collector or reference counting to make sure an object never disappears when it's still accessible. That's a lot of complexity and performance overhead, but if you really want reference semantics, it might be worth it.

Rust uses a static approach that avoids allocation and lets you use special pointers and references in cases when you can prove to the compiler that mutable state is never actually shared. It works, but the ergonomics aren't great, and "fighting the borrow checker" has become a common meme among Rust programmers.

Some people say you can take **raw** pointers and references as they are, and use static analysis to make them safe. The analyzers that can do that job without false positives or negatives don't exist today, so I'm not sure if that actually counts as a solution.



Reference semantics | Safety solutions

Technical debt	Lifetime safety
Spooky action	· Dynamic allocation + GC/RC
Incidental algorithms	· Borrow checker/named lifetimes
Visibly broken invariants	· Just add static analysis?
Race conditions	Thread safety
Surprise mutation	· Define it and ignore the real problem
Unspecifiable mutation	· Borrow checker/named lifetimes
	· Detect and trap

For race conditions, I know of three basic approaches.

First, you could make the behavior defined and continue running, like Java does. Unfortunately, the defined behavior you get from Java is basically useless, and almost certainly masks a bug in your code.

A more responsible approach is static, like the Rust borrow checker. Again, works great but hard to use.

Finally, if you can't statically prove to Rust that you don't have a race, you can bypass the borrow checker by using an atomic_refcell, which dynamically checks for race conditions and terminates via “panic” if one is found. Halting the program is a legit way to avoid unsafety. The alternative is to produce meaningless results and hide bugs. But maybe you'd rather not find your bugs at runtime, especially if they're nondeterministic threading bugs.

To sum up, making reference semantics safe is possible but nontrivial, and you <u>will</u> be making tradeoffs.

OK, stepping back,

Reference semantics | What about immutability?

Technical debt

Spooky action

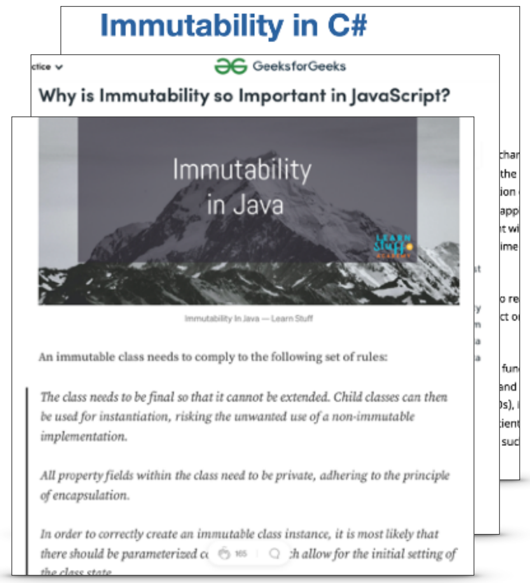
Incidental algorithms

Visibly broken invariants

Race conditions

Surprise mutation

Unspecifiable mutation



Immutability in C#

GeeksforGeeks

Why is Immutability so Important in JavaScript?

Immutability in Java

Immutability In Java — Learn Stuff

An immutable class needs to comply to the following set of rules:

The class needs to be final so that it cannot be extended. Child classes can then be used for instantiation, risking the unwanted use of a non-immutable implementation.

All property fields within the class need to be private, adhering to the principle of encapsulation.

In order to correctly create an immutable class instance, it is most likely that there should be parameterized c… h allow for the initial setting of the class state

If you look at this list, it turns out most of these issues are connected to mutation, and especially in languages where everything is by-reference, people have seized on immutability as an answer.

When you hear Haskell programmers talk about how it's hard to create bugs, believe them! Between their garbage collector and strict immutability, Haskell defines away many of these problems.

Of course, giving up on mutation has costs too. It's just a bad fit for some problems, to say nothing of our mental model as programmers. This>>
Is a wonderful little paper about how hard it is to implement one of the oldest known algorithms in Haskell.

Finally, the underlying machine has mutable memory, and switching to a model where nothing is mutated can be super-inefficient. To get that efficiency back, optimizers need to be smart enough to recreate the mutation you've taken out of your programs—and they often fail.

But, immutability is really the only way to solve these problems in languages like JavaScript. In C++ we have alternatives.

Reference semantics | What about immutability?

Technical debt

Spooky action

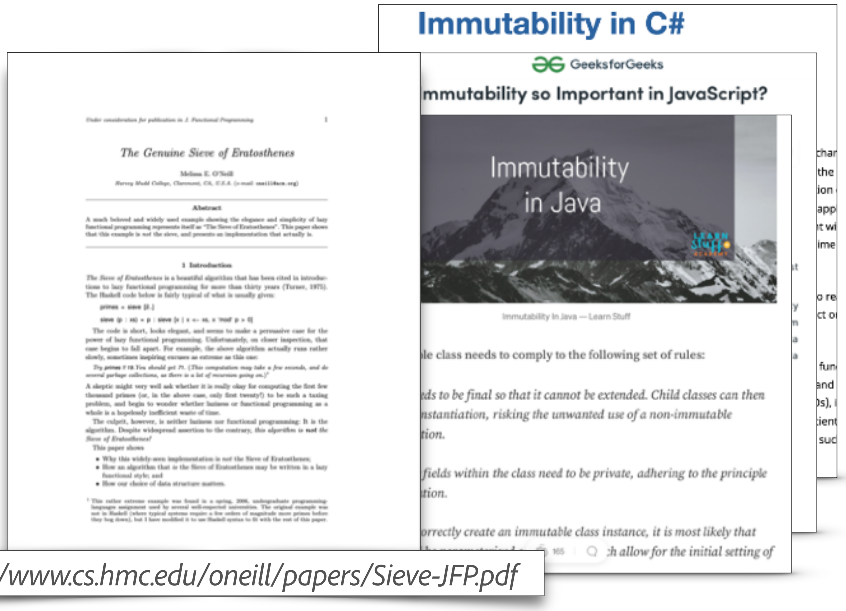
Incidental algorithms

Visibly broken invariants

Race conditions

Surprise mutation

Unspecifiable mutation



Immutability in C#

GeeksforGeeks

mmutability so Important in JavaScript?

Immutability in Java

Immutability In Java — Learn Stuff

le class needs to comply to the following set of rules:

ds to be final so that it cannot be extended. Child classes can then nstantiation, risking the unwanted use of a non-immutable tion.

fields within the class need to be private, adhering to the principle tion.

orrectly create an immutable class instance, it is most likely that h allow for the initial setting of

Under consideration for publication in J. Functional Programming 1

The Genuine Sieve of Eratosthenes

Melissa E. O'Neill

Harvey Mudd College, Claremont, CA, U.S.A. (e-mail: oneill@acm.org)

Abstract

A much beloved and widely used example showing the elegance and simplicity of lazy functional programming represents itself as "The Sieve of Eratosthenes". This paper shows that this example is not the sieve, and presents an implementation that actually is.

1 Introduction

The Sieve of Eratosthenes is a beautiful algorithm that has been cited in introductions to lazy functional programming for more than thirty years (Turner, 1975). The Haskell code below is fairly typical of what is usually given:

```
primes = sieve [2..]
sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

The code is short, looks elegant, and seems to make a persuasive case for the power of lazy functional programming. Unfortunately, on closer inspection, that case begins to fall apart. For example, the above algorithm actually runs rather slowly, sometimes inspiring excuses as extreme as this one:

Try **primes !! 19**. *You should get 71. (This computation may take a few seconds, and do several garbage collections, as there is a lot of recursion going on.)*[1]

A skeptic might very well ask whether it is really okay for computing the first few thousand primes (or, in the above case, only first twenty!) to be such a taxing problem, and begin to wonder whether laziness or functional programming as a whole is a hopelessly inefficient waste of time.

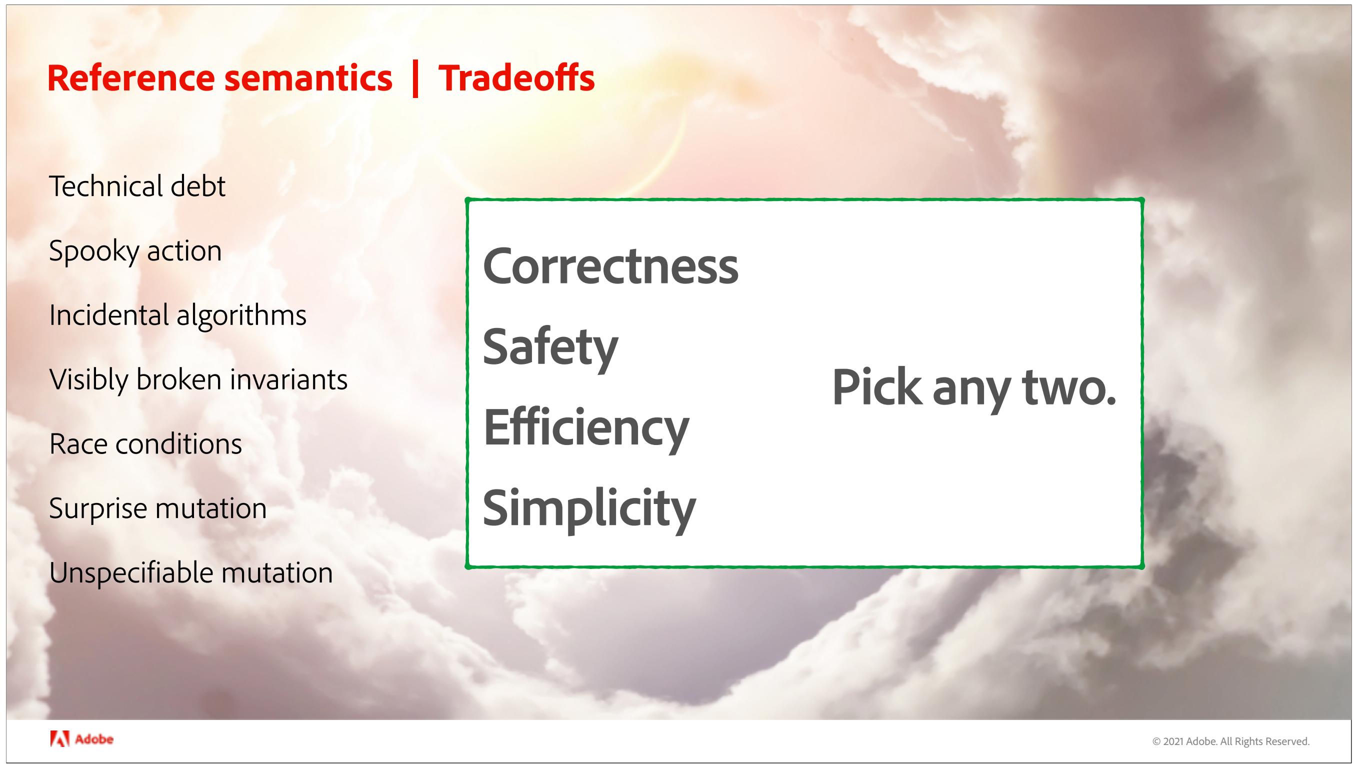
The culprit, however, is neither laziness nor functional programming: It is the algorithm. Despite widespread assertion to the contrary, *this algorithm is* **not** *the Sieve of Eratosthenes!*

This paper shows

- Why this widely-seen implementation is not the Sieve of Eratosthenes;
- How an algorithm that *is* the Sieve of Eratosthenes may be written in a lazy functional style; and
- How our choice of data structure matters.

[1] This rather extreme example was found in a spring, 2006, undergraduate programming-language assignment used by several well-respected universities. The original example was not in Haskell (where typical systems require a few orders of magnitude more primes before they bog down), but I have modified it to use Haskell syntax to fit with the rest of this paper.

https://www.cs.hmc.edu/oneill/papers/Sieve-JFP.pdf



Reference semantics | Tradeoffs

- Technical debt
- Spooky action
- Incidental algorithms
- Visibly broken invariants
- Race conditions
- Surprise mutation
- Unspecifiable mutation

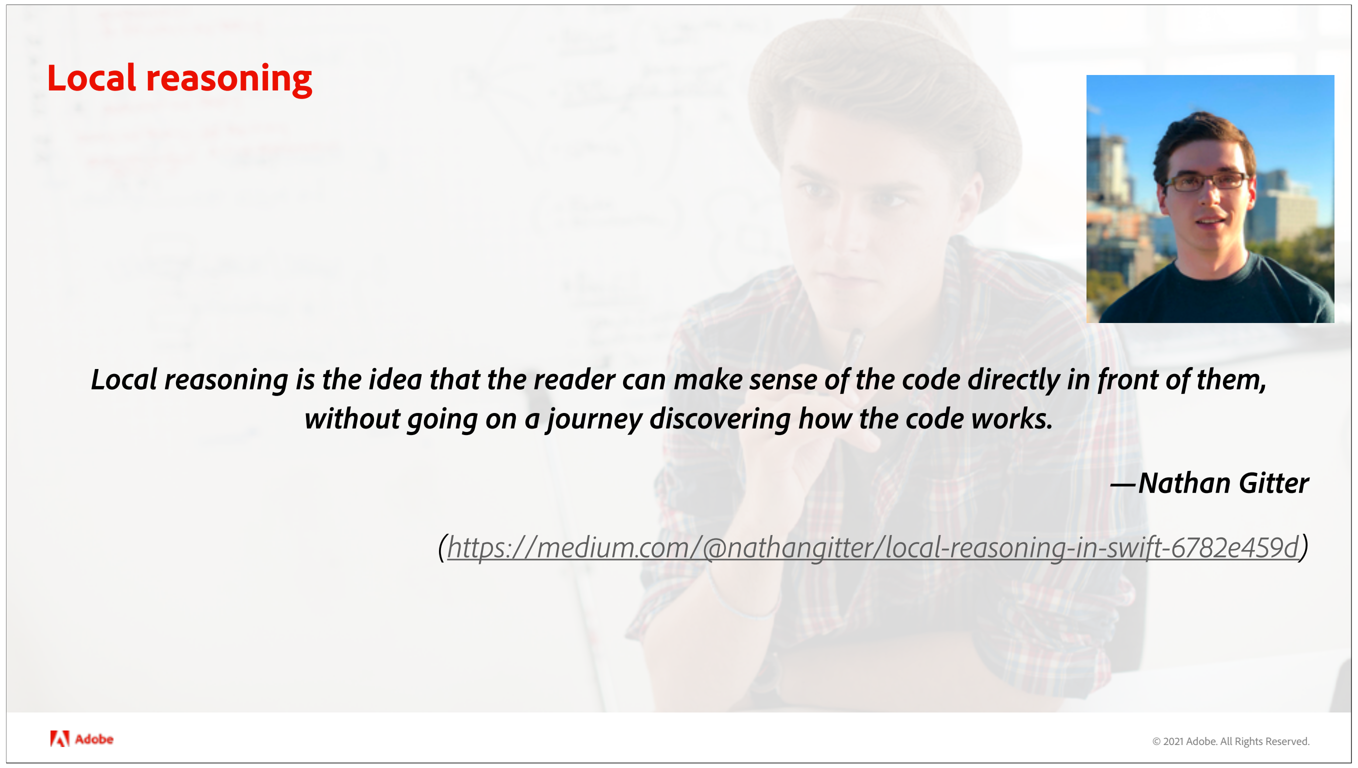
Correctness
Safety
Efficiency
Simplicity

Pick any two.

So this is what I'm left with. I believe there are deep truths in engineering.

I'm not talking about the pure abstract math kind of truth; what I mean is grounded in physical reality and the constraints of our problems.

If you listen, the universe will help you find these answers, and in this case, it's telling us that reference semantics is a bad deal.



Local reasoning

Local reasoning is the idea that the reader can make sense of the code directly in front of them, without going on a journey discovering how the code works.

—Nathan Gitter

(https://medium.com/@nathangitter/local-reasoning-in-swift-6782e459d)

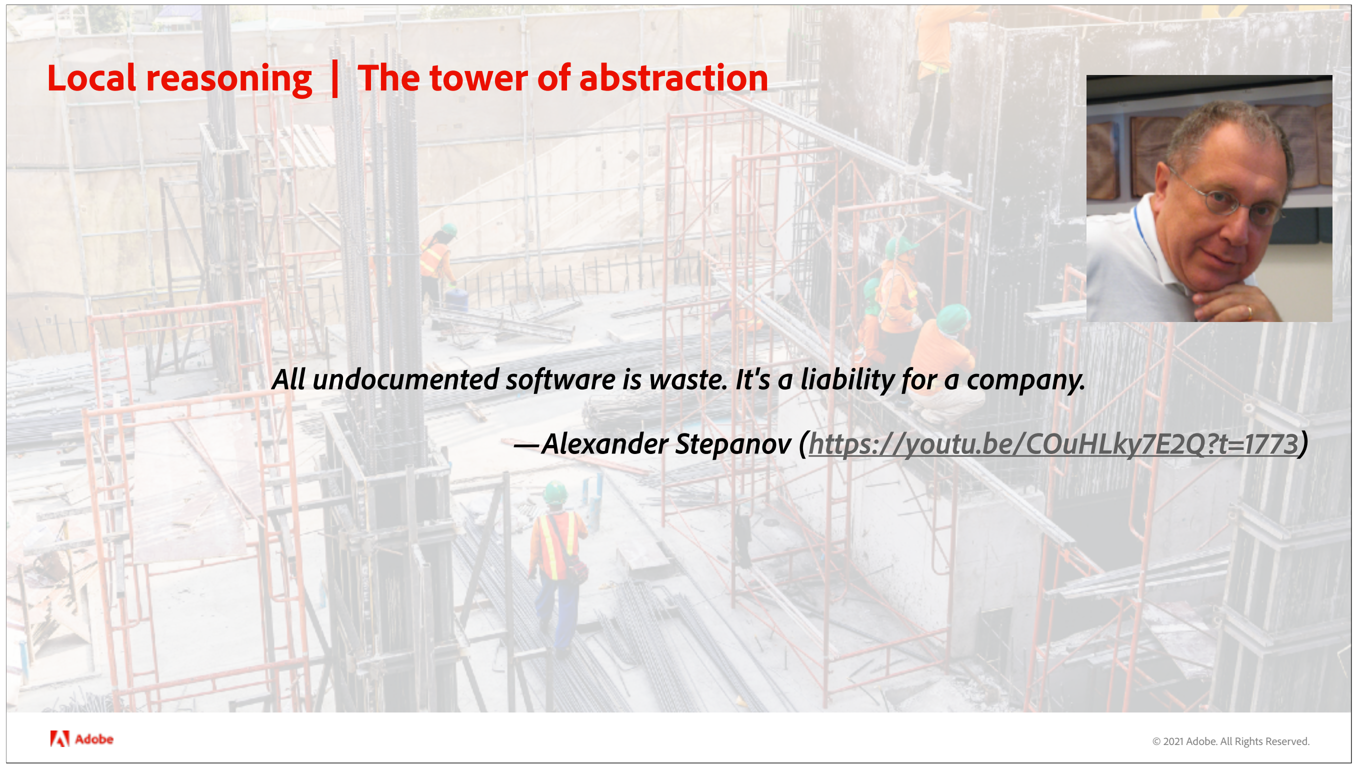
Adobe

You may have noticed local reasoning is a bit of a theme, here. The clearest definition I've found is an informal one from Nathan Gitter: "Local reasoning is the idea that the reader can make sense of the code directly in front of them, without going on a journey discovering how the code works"

My brain has limited capacity. And I've found a lot of other peoples' brains are limited too. Not yours of course, but a lot of peoples' are. People like us can't keep the whole program in our heads. Of course local reasoning is about more than just the problems with references, and it goes beyond programming; it's how humans deal with complexity

In fact, it's so fundamental that most of our programming best practices are there just to enable it. It's why we make data members private, why we break programs into components like functions, types, and modules, and we try to keep them small. It's why we use class invariants, and why we avoid global variables.

Compilers use local reasoning too. If they can reason locally that data isn't being changed, they can keep it in registers and avoid re-fetching it from memory. A big part of the reason we get such a win from inlining is that the compiler has more local information to work with. But that strategy has limits; when the local scope gets too large, compilers have to give up trying to draw these conclusions so we don't wait all day for their output.



Local reasoning | The tower of abstraction

All undocumented software is waste. It's a liability for a company.

—Alexander Stepanov (https://youtu.be/COuHLky7E2Q?t=1773)

Local reasoning is why I don't have to be a physicist to program a computer.

Here's what I mean. As programmers, we're working on what my friend Sam Lazarus calls "a tower of abstraction" that stretches through the libraries and programming language we use, the operating system, and into the hardware, which ultimately rests on the laws of physics.

So what keeps us from recursing down to the limits of known physics when we think about our programs?

The answer is documentation. Documentation gets no respect, but If you're old enough to remember when the STL came out, you know what a huge difference concise but meticulous documentation makes. For some of us, the STL revolutionized software development, and a huge part of what made it so powerful was the documentation.

Alex Stepanov said, in a seminar he gave at Adobe, that

All undocumented software is waste; it's a liability for a company.

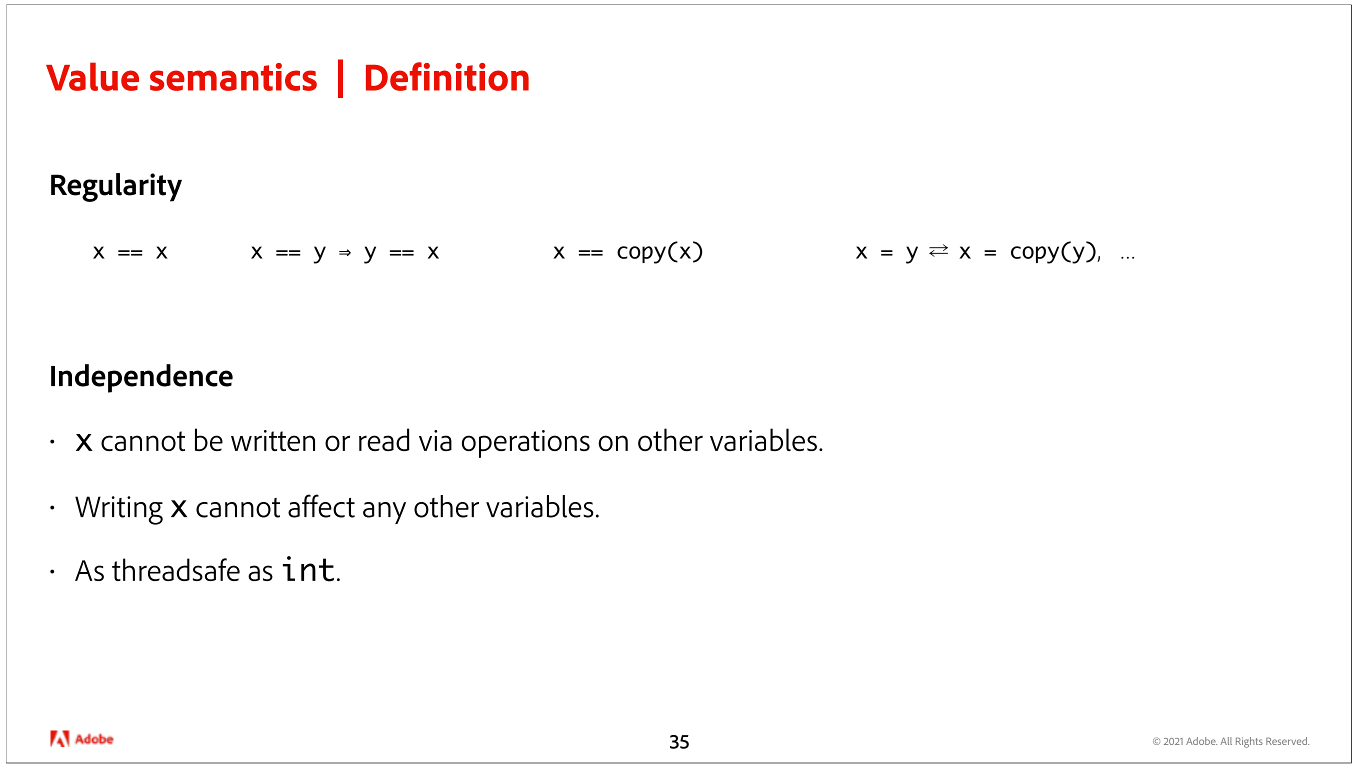
We can use libraries and our programming language without digging into their implementations because there's a solid spec. The compiler backend engineers can do their jobs because the hardware manufacturers document the architecture and instruction sets. The hardware designers succeed because the physicists document the laws of physics. That's the tower, and you're a part of it. The bad news, of course, is that you're not at the top of the tower. Someone else, or future-you, is going to have to build on the code you're writing. We're all library builders.

So local reasoning and the documentation that supports it is required for correct software construction at scale. We want to write correct code; people's lives and livelihoods are in our hands. So how can we uphold local reasoning?

Fortunately, the solution is under our noses. It's value semantics, and—I'm not kidding now—it's the future of programming.

Value semantics is a property of types, like the integer types in C++.

People often describe it as “doing as the ints do”.
But we really need a solid definition to proceed from here.



Value semantics | Definition

Regularity

`x == x`   `x == y ⇒ y == x`   `x == copy(x)`   `x = y ⇄ x = copy(y)`, …

Independence

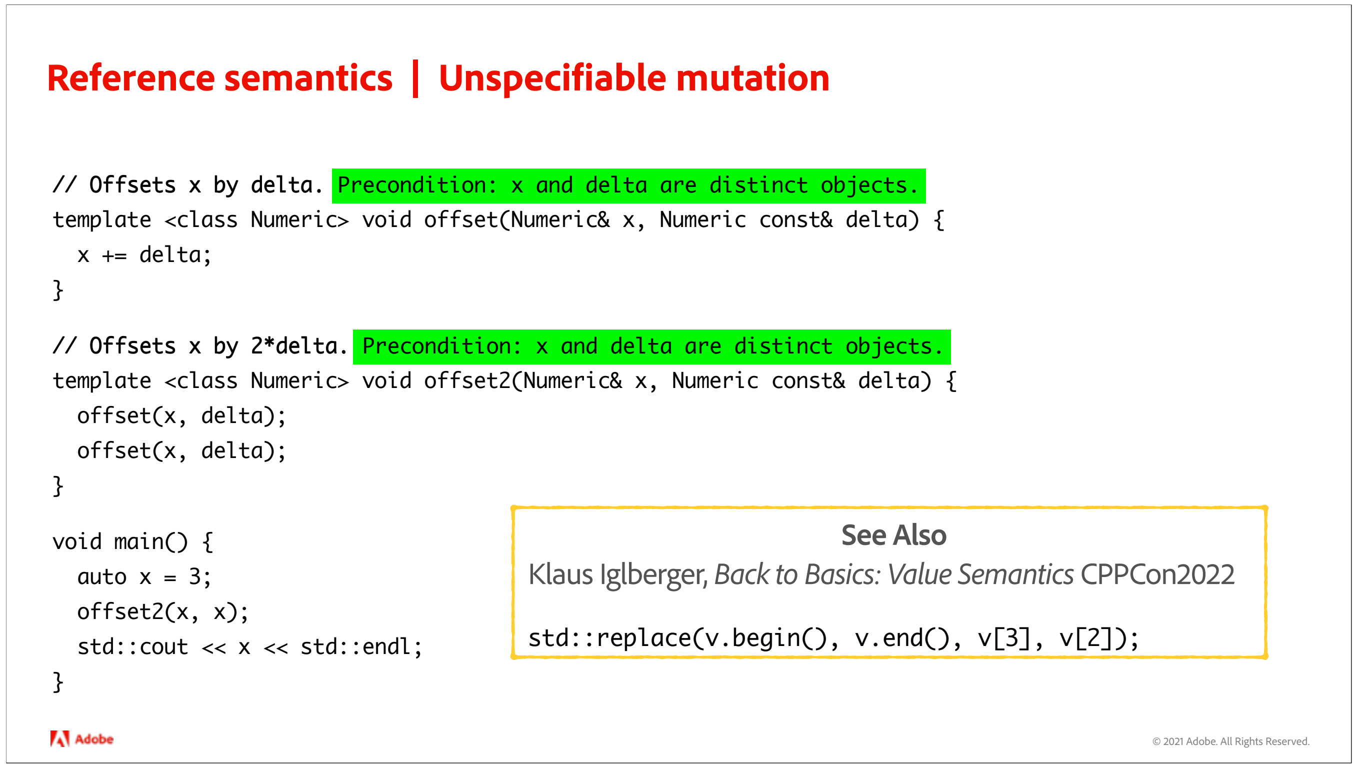
- `x` cannot be written or read via operations on other variables.
- Writing `x` cannot affect any other variables.
- As threadsafe as `int`.

Value semantics is really the combination of two simple ideas.

First, regularity is about the relationships between basic operations like of copy, assignment, and equality comparison. For example a thing always has to be equal to itself, and copying a thing has to make a thing that's equal to the original.

Independence is about the visibility of mutation: it's the idea that when you have a variable of value type, other code can't do anything to affect your locally-visible value, and nothing **you** do to the value can be observed by other code, even if you pass it to another thread… as long as you don't form a pointer or reference to it.

So if you flip back to this slide, where we were trying to rule out overlapping access



Reference semantics | Unspecifiable mutation

```
// Offsets x by delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

See Also

Klaus Iglberger, *Back to Basics: Value Semantics* CPPCon2022

```
std::replace(v.begin(), v.end(), v[3], v[2]);
```

Adobe

These two preconditions are just trying to achieve independence.

And it's not specific to this function: it hit me like a ton of bricks, when I realized this:
>> Without independence, the effects of any mutation cannot be fully specified!

Think about it for a second. You're mutating your parameters, and you can talk about that in your documentation... but you're also mutating some other stuff that you don't even have a way to describe. So every single mutating operation, from a single machine instruction to a database transaction, really depends on independence for its correctness.
It might be an unstated requirement, but it's there.

But have you ever seen an independence requirement documented? I haven't. Thinking about why not led me to another realization: we just assume references are accessing independent values, either uniquely accessible for mutation or truly immutable. We take this shortcut, because it's the only way to make sense of things.

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset(Numeric& x, Numeric const& delta) {
  x += delta;
}

// Offsets x by 2*delta. Precondition: x and delta are distinct objects.
template <class Numeric> void offset2(Numeric& x, Numeric const& delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

independence

independence

See Also

Klaus Iglberger, *Back to Basics: Value Semantics* CPPCon2022

```
std::replace(v.begin(), v.end(), v[3], v[2]);
```

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta. Precondition: x and delta are di
template <class Numeric
  x += de
}

// Offset
template <
  offset(x
  offset(x, delta);
}
```

pendence

pendence

```
                                              Numeric const& delta) {
```

Every single mutating operation
of any kind has independence
as a requirement

```
void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

See Also

Klaus Iglberger, *Back to Basics: Value Semantics* CPPCon2022

```
std::replace(v.begin(), v.end(), v[3], v[2]);
```

Reference semantics | Unspecifiable mutation

```
// Offsets x by delta. Precondition: x and delta
template <class Numeri
  x += de
}

// Offset
template <
  offset(x
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

pendence

pendence

const& delta) {

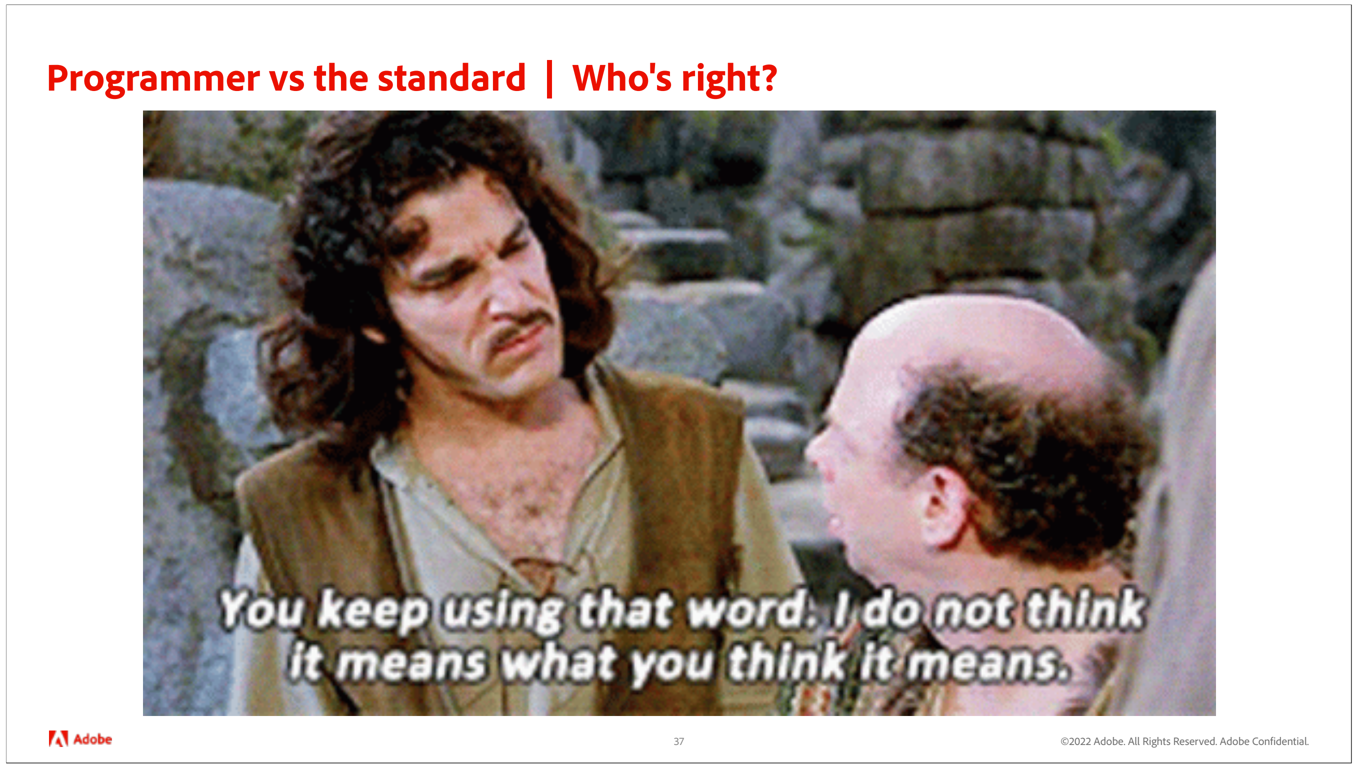
Every single mutating operation *of any kind* has independence as a requirement

Independence is already in our mental model for references.

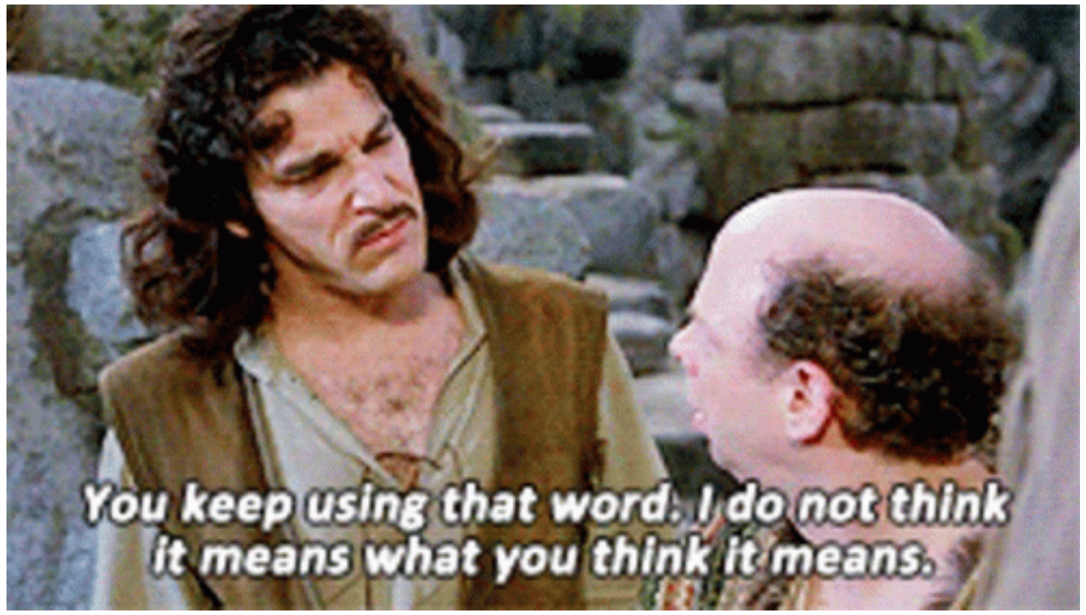
igberger, *Back to Basics: Value Semantics* CPPCon2022

```
std::replace(v.begin(), v.end(), v[3], v[2]);
```

Adobe



Programmer vs the standard | Who's right?


You keep using that word. I do not think
it means what you think it means.

So when we use references, we assume independence, but the compiler has a different view of the world: it has to play by the rules in the standard.

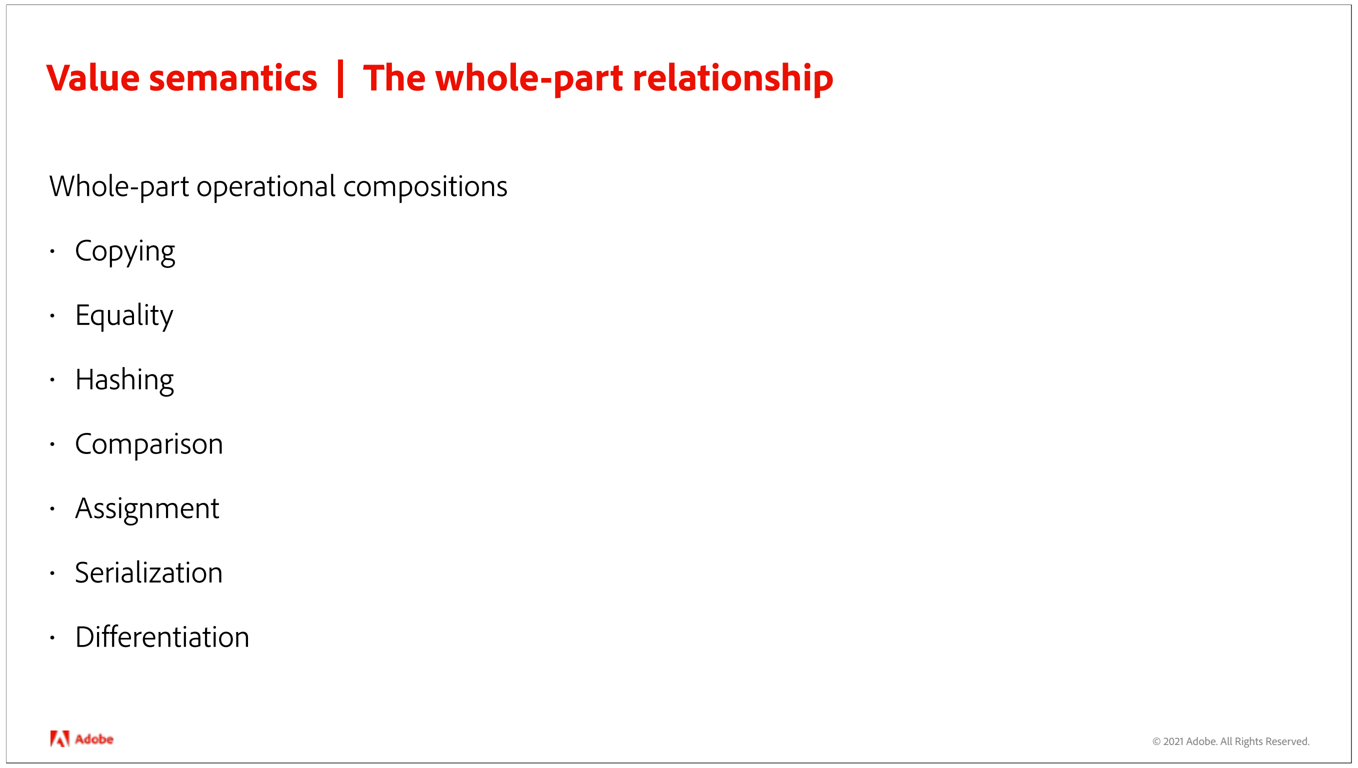
Not only is the compiler obliged to let our clients do stuff we're not prepared to handle, like mutating a thing we're handling via const reference, it also can't make any assumptions about which things might be changed by an opaque function call, and that means lost optimization opportunities.

I'm going to say something now that you won't hear often from me: the standard is wrong and you are right. Systems without independence are incredibly difficult to reason about, both for you and for the compiler, so the standard enshrines the wrong model.

So don't change your mental model. Instead, document it. That means telling the clients of your APIs (in one place) that there is a blanket assumption of independence they're expected to uphold: when your APIs take a parameter by const reference, no other code is allowed to modify the thing being referred to during the the call, and if it's a mutable reference, no other code is allowed to inspect the thing referred to.

None of that will help the compiler do better, but it will make it easier to avoid creating weird unspecifiable behaviors.

OK, back to value semantics.



Value semantics | The whole-part relationship

Whole-part operational compositions

- Copying
- Equality
- Hashing
- Comparison
- Assignment
- Serialization
- Differentiation

Adobe

One of the most important properties of value semantics is that, in the same way that a composition of safe operations is safe, a composition of value types is a value type.

And unlike with references, this composition always creates a whole-part relationship, which matches our mental model for combining things and maps onto the physical world.

This>>

Value semantics | The whole-part relationship

Whole-part operational compositions

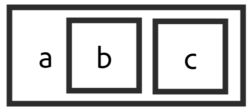
- Copying
- Equality
- Hashing
- Comparison
- Assignment
- Serialization
- Differentiation

$$\begin{tikzcd}
\boxed{a} \arrow[d] \arrow[rd] & \\
\boxed{b} & \boxed{c}
\end{tikzcd}$$

Value semantics | The whole-part relationship

Whole-part operational compositions

- Copying
- Equality
- Hashing
- Comparison
- Assignment
- Serialization
- Differentiation



Becomes this.

And now you can see that access to the parts requires access to the whole, which is key to many of the awesome properties of values.

Lastly, you can see that all of these operations here have an obvious definition: copying the whole means copying all the parts. Testing equality is done part-wise, and so on.

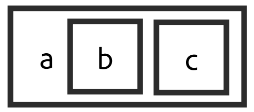
In fact, I want you to take this opportunity to get some bad words out of your vocabulary. I'm talking about “deep and shallow.” We only use phrases like “deep copy” when we don't understand the whole-part relationships; the boundary of our value.
>> Because there is no such thing as deep copy; there's just copy, which copies all the parts.
>> Likewise there's no such thing as shallow copy.
>> and the same goes for the rest of these operations

So “deep“ and “shallow” are red flags for you now.

Value semantics | The whole-part relationship

Whole-part operational compositions

- Copying
- Equality
- Hashing
- Comparison
- Assignment
- Serialization
- Differentiation

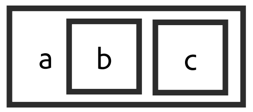


There is no deep copy 🥄

Value semantics | The whole-part relationship

Whole-part operational compositions

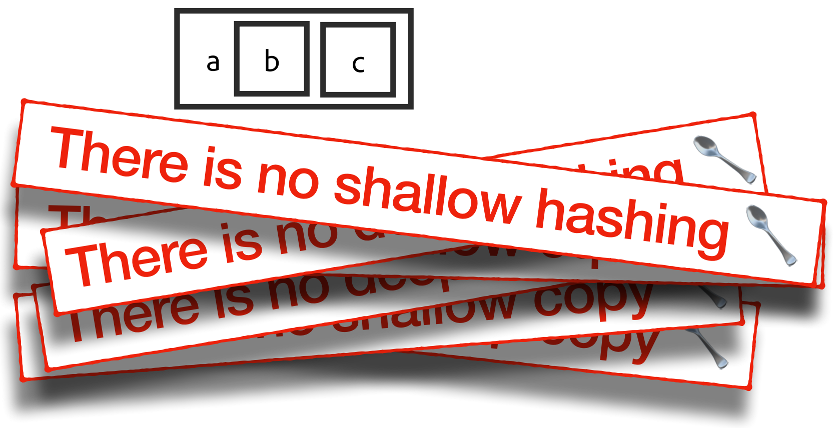
- Copying
- Equality
- Hashing
- Comparison
- Assignment
- Serialization
- Differentiation



Value semantics | The whole-part relationship

Whole-part operational compositions

- Copying
- Equality
- Hashing
- Comparison
- Assignment
- Serialization
- Differentiation



Value semantics | Solution checklist

Value semantics | Solution checklist

✅ Technical debt

Value semantics | Solution checklist

✅ Technical debt

✅ Spooky action

Value semantics | Solution checklist

✅ Technical debt

✅ Spooky action

✅ Visibly broken invariants

Adobe

Value semantics | Solution checklist

✅ Technical debt

✅ Spooky action

✅ Visibly broken invariants

✅ Race conditions

Adobe

Value semantics | Solution checklist

✅ Technical debt

✅ Spooky action

✅ Visibly broken invariants

✅ Race conditions

✅ Surprise mutation

Value semantics | Solution checklist

✅ Technical debt

✅ Spooky action

✅ Visibly broken invariants

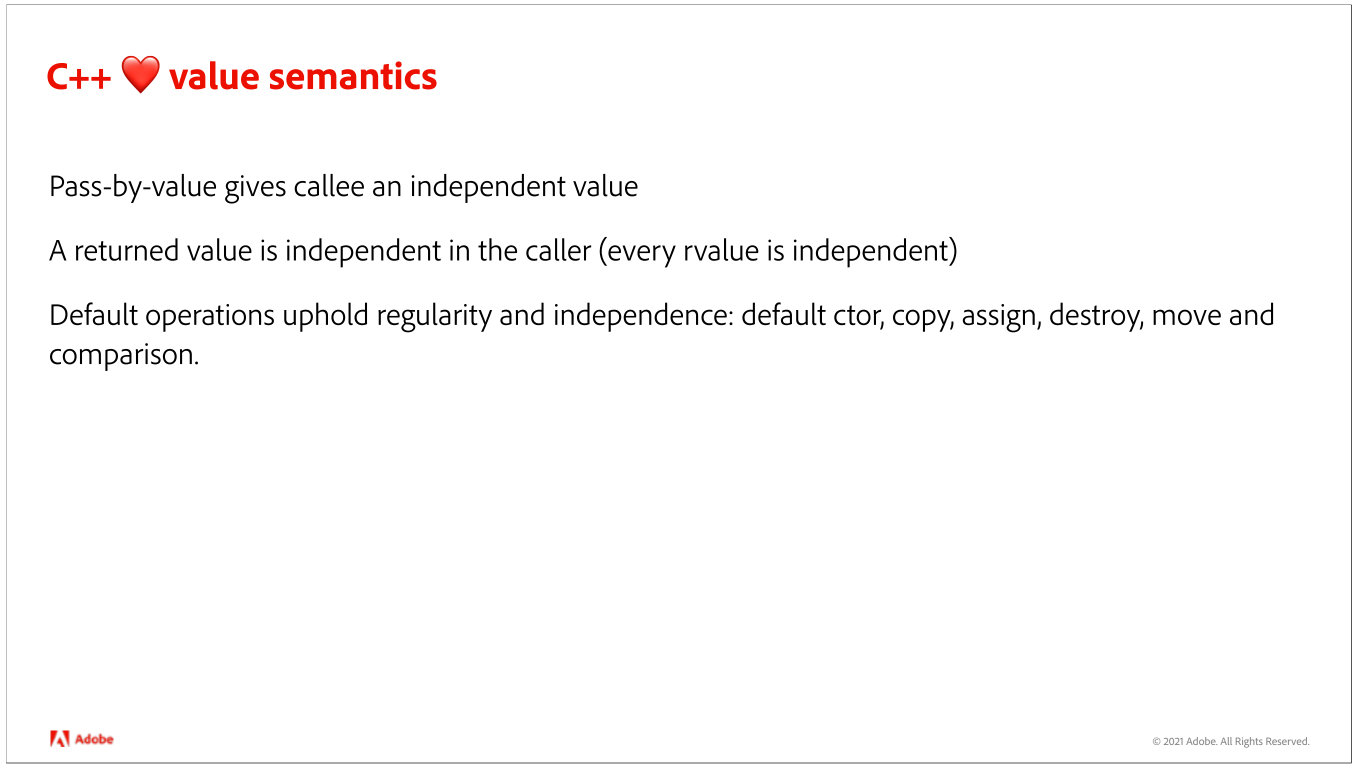
✅ Race conditions

✅ Surprise mutation

✅ Unspecifiable mutation

Value semantics | Solution checklist

- ✅ Technical debt
- ✅ Spooky action
- ✅ Visibly broken invariants
- ✅ Race conditions
- ✅ Surprise mutation
- ✅ Unspecifiable mutation



C++ ❤️ value semantics

Pass-by-value gives callee an independent value

A returned value is independent in the caller (every rvalue is independent)

Default operations uphold regularity and independence: default ctor, copy, assign, destroy, move and comparison.

Adobe

This is a rare and valuable thing, a forgotten wisdom from langauges pascal and Ada. For 30 years during the OO revolution almost every language was based on references.

>>Fun fact: UML has first-class notation for whole-part relationships, so they clearly knew it was important. And yet AFAIK nobody ever put support it in an Object Oriented language.

Unfortunately, though C++ also undermines value semantics
>> by achieving independence through copying.

Which prompted this quote:

C++ ❤️ value semantics

Pass-by-value gives callee an independent value

A returned value is independent in the caller (every rvalue is independent)

Default operations uphold regularity and independence: default ctor, copy, assign, destroy, move and comparison.

Fun Fact: UML has first class notation for whole-part relationships.

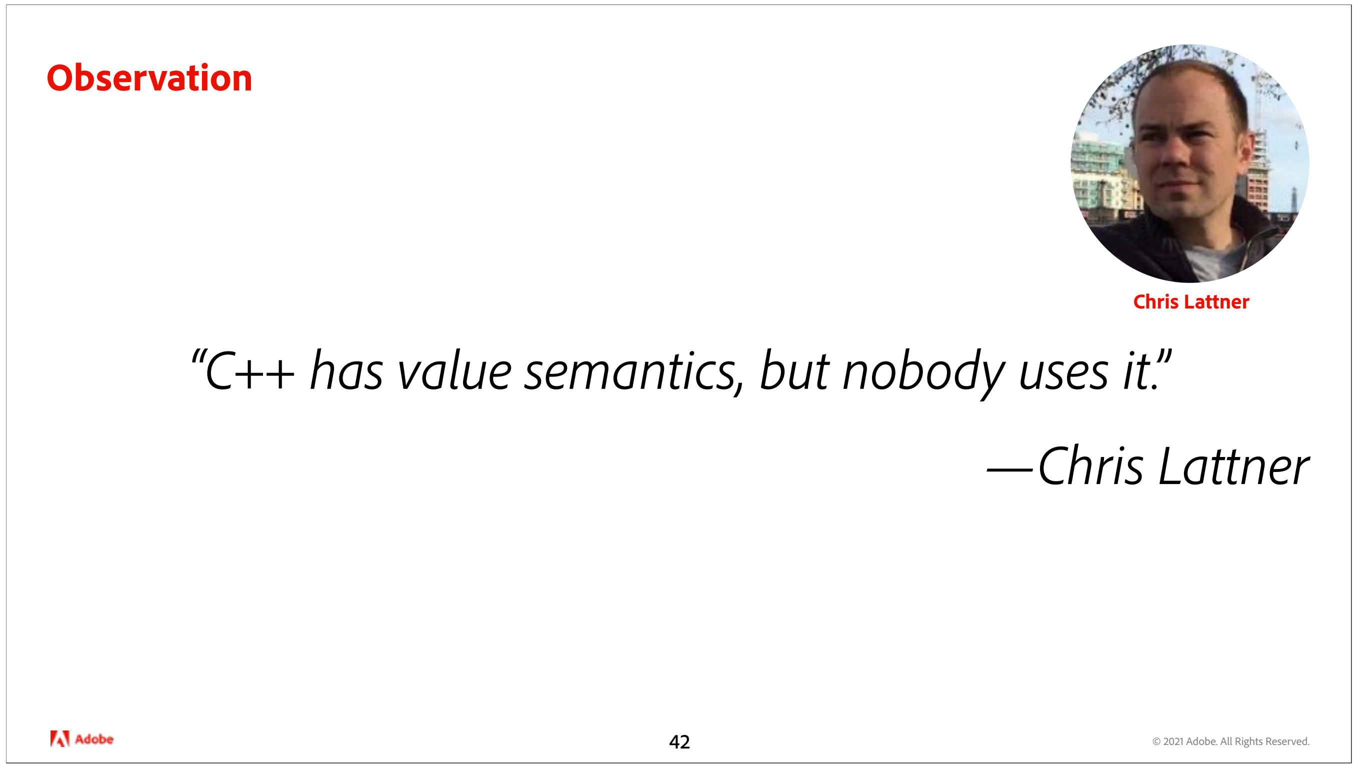
C++ ❤️ value semantics

Pass-by-value gives callee an independent value ***by copying it.***

A returned value is independent in the caller (every rvalue is independent)

Default operations uphold regularity and independence: default ctor, copy, assign, destroy, move and comparison.

Fun Fact: UML has first class notation for whole-part relationships.



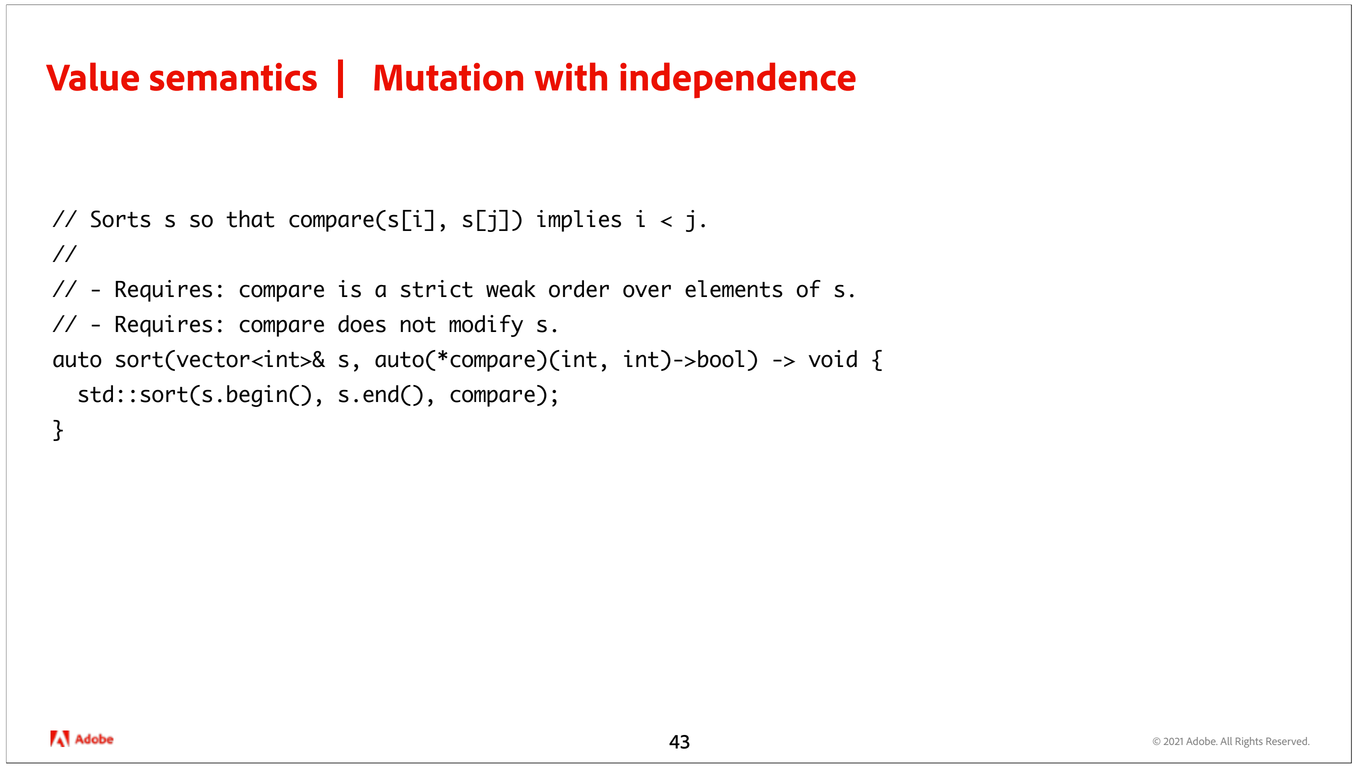
Observation

"C++ has value semantics, but nobody uses it."

—Chris Lattner

What Chris meant was that because C++ makes copies so eagerly, nobody really wants to pass things bigger than an int by value. It's a disincentivizing design choice.

It's true in other ways; there are places we'd like to use value semantics, like in-place mutation, where are only choices are pointers and references. But there is a way to do mutation, not in-place, that upholds value semantics



Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}
```

Here we have a simple sort function that works on a vector of int and just dispatches to std::sort.

>> for testing we declare a vector

>> and a comparison function

>> and here's our test.

>> Now notice that some incredibly, unrealistically conscientious person has actually documented that s and compare need to be independent. Otherwise, this call to std::sort would cause undefined behavior.

Now let's create statically-enforced independence so we can delete that line of documentation.

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };
```

Value semantics | Mutation with independence

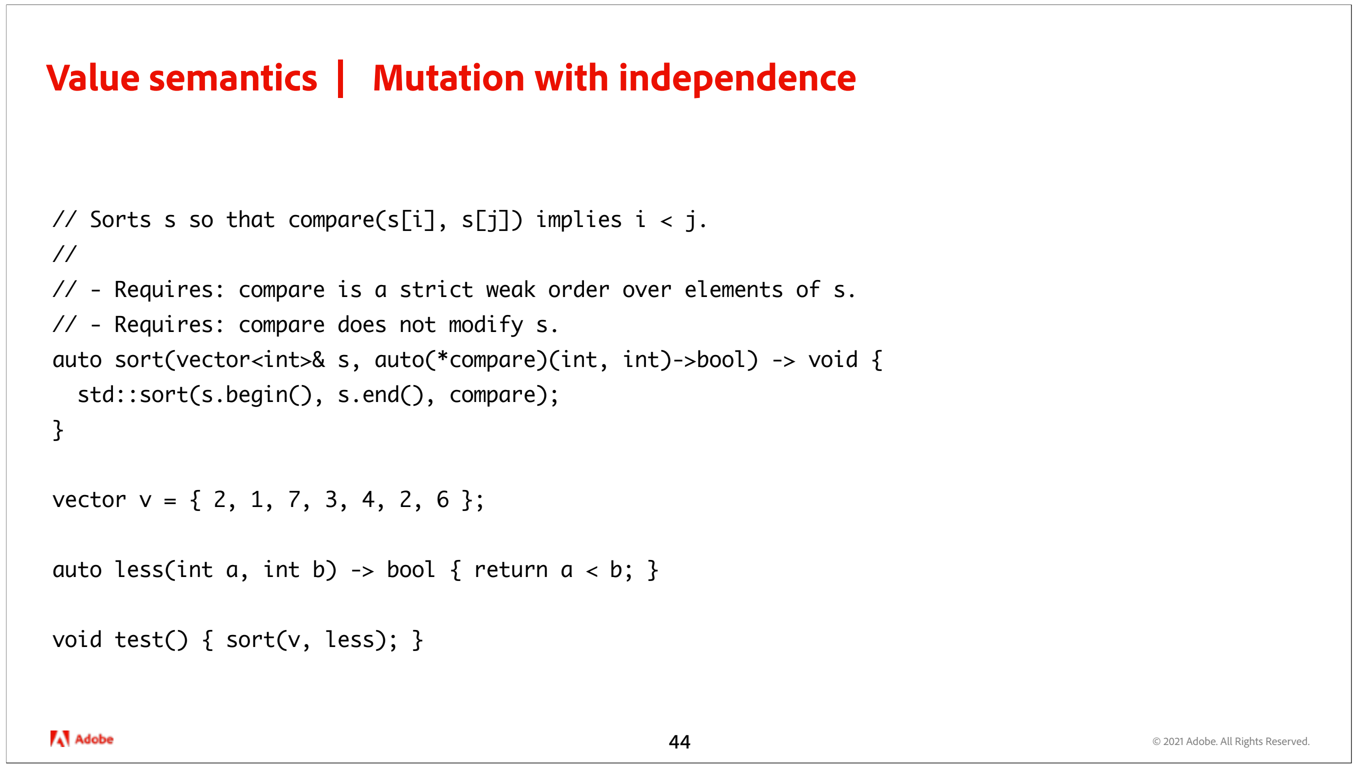
```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }
```

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.  <independence
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```



Value semantics | Mutation with independence

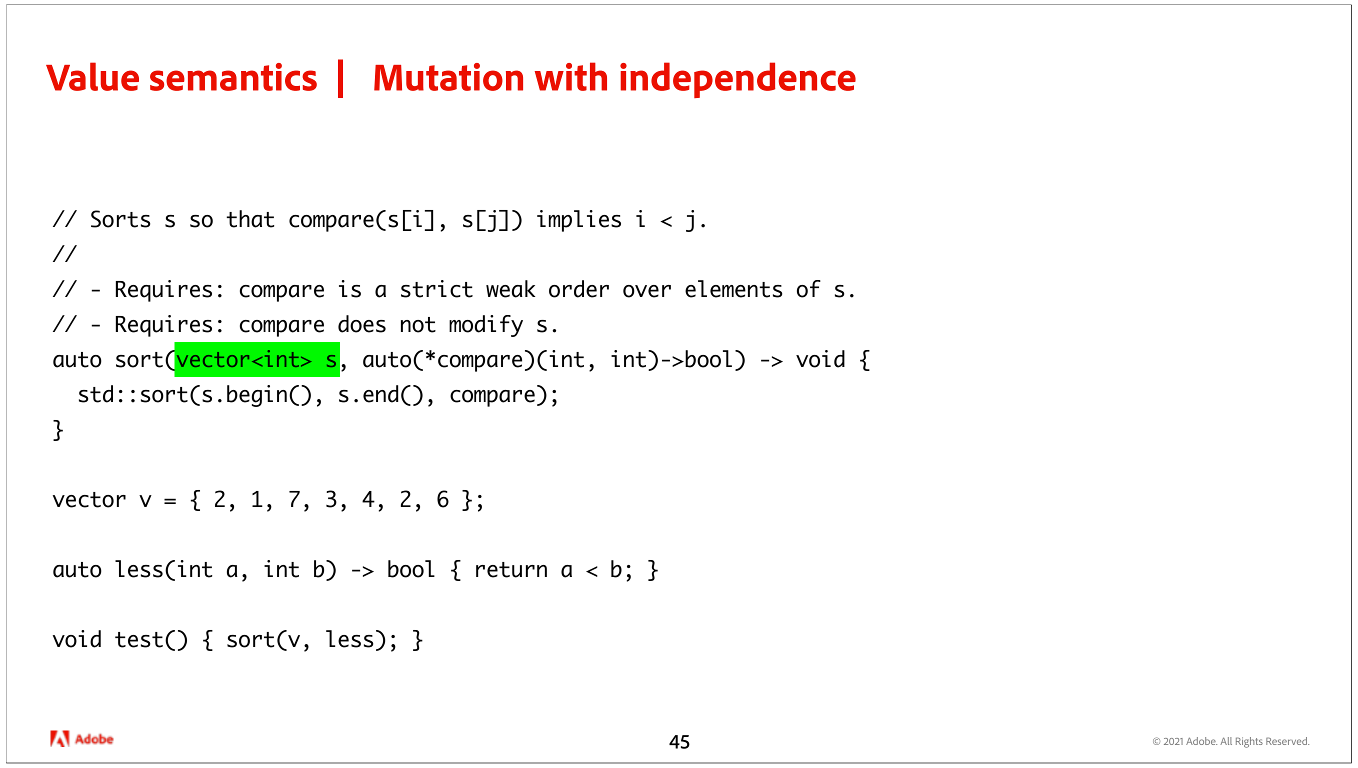
```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Adobe

First, we need to remove the reference from our argument.

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```



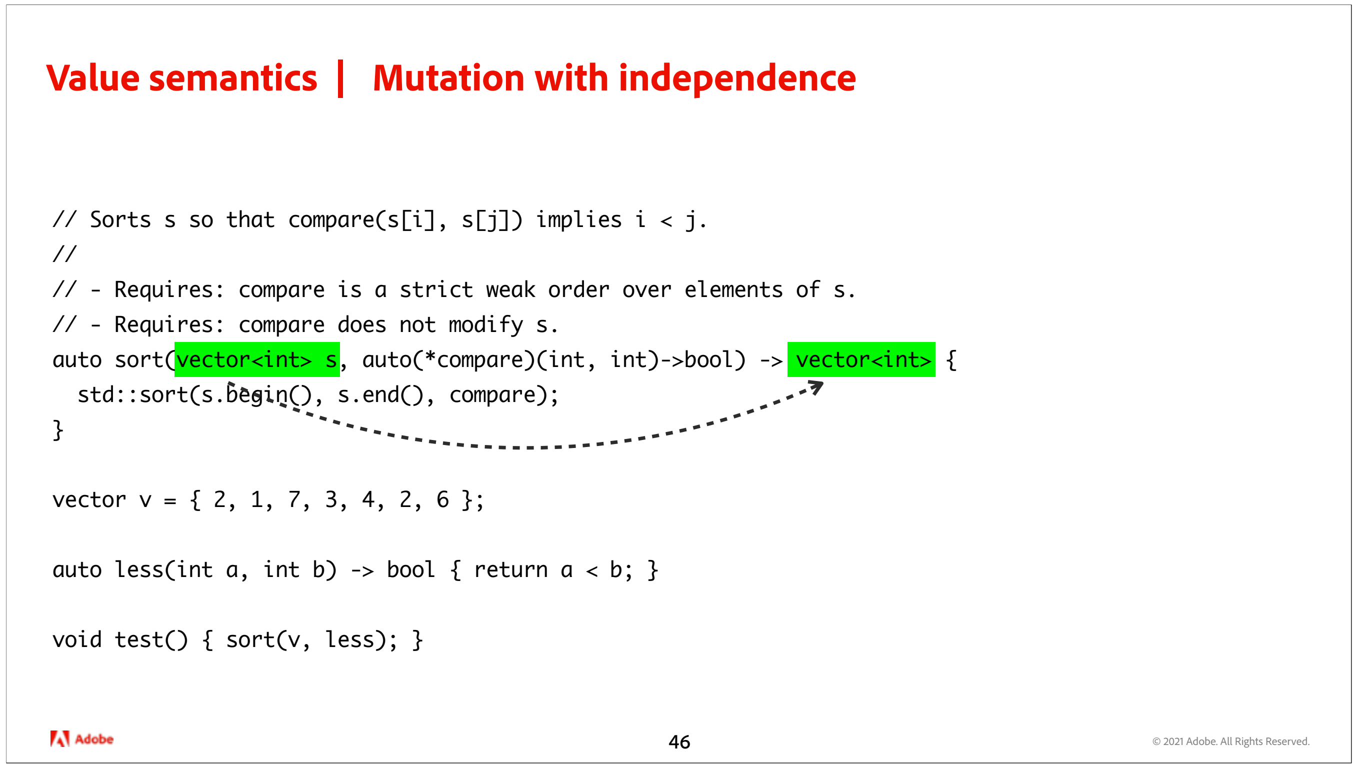
Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

But then return the same type we were mutating.

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> void {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```



Value semantics | Mutation with independence

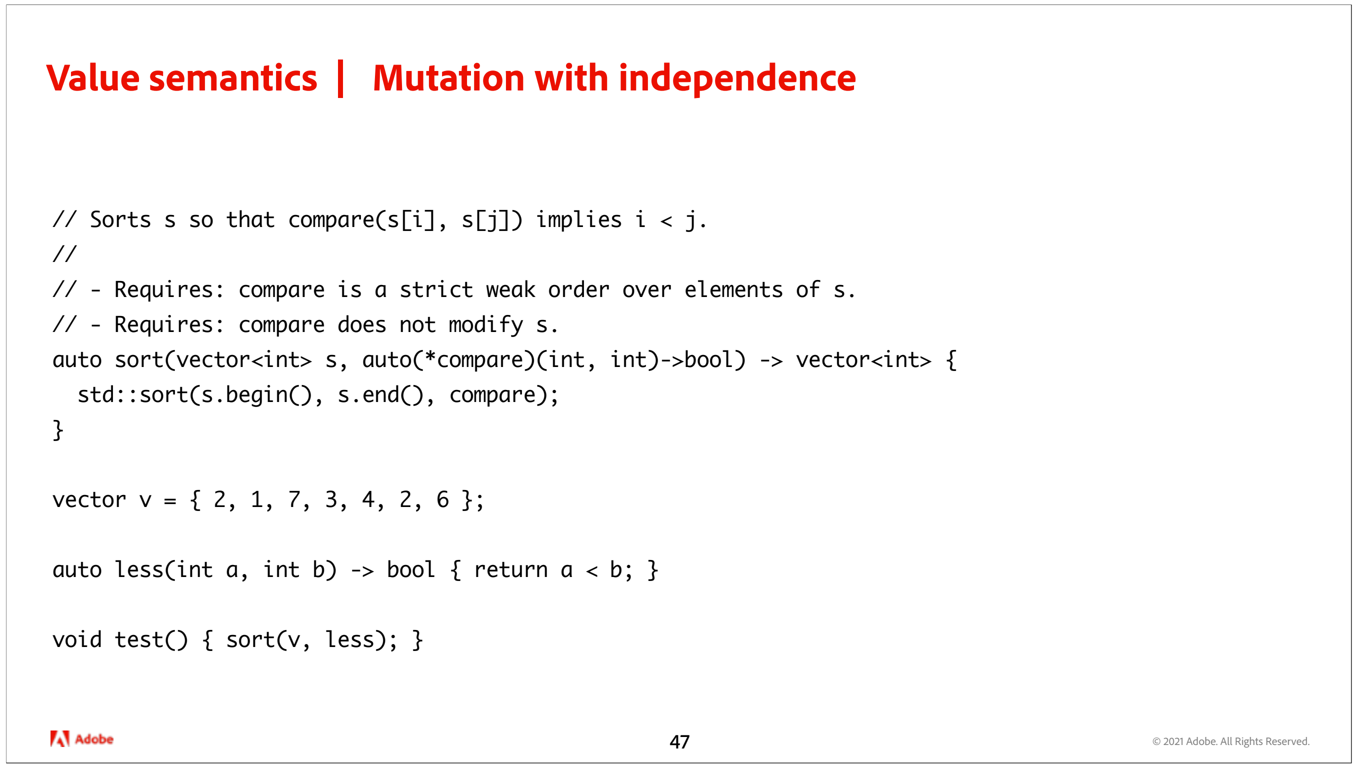
```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Adobe

If our return type hadn't been void, we could package the result into a tuple.
Now we actually have to return s

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```


Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare);
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

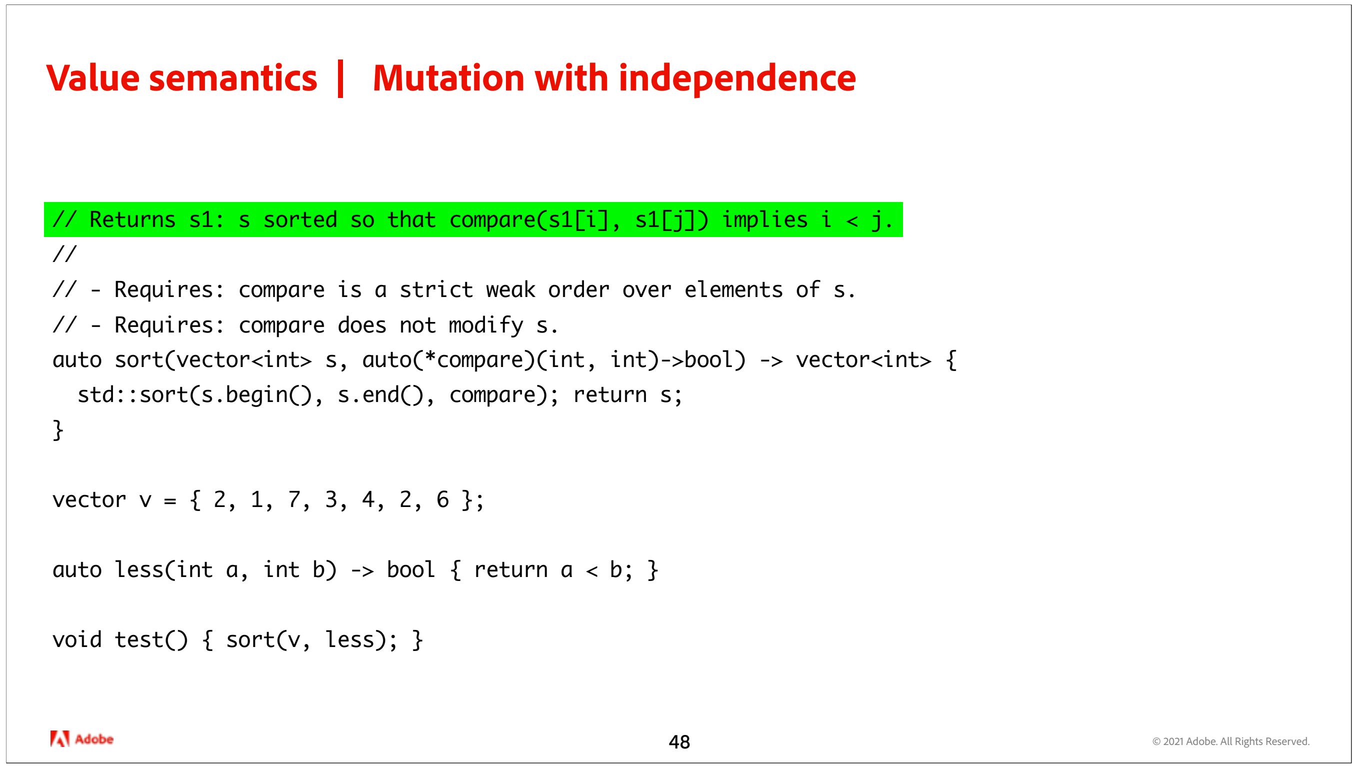
If our return type hadn't been void, we could package the result into a tuple.

Now we have to actually return something >>

Fix the documentation

Value semantics | Mutation with independence

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```


Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Adobe

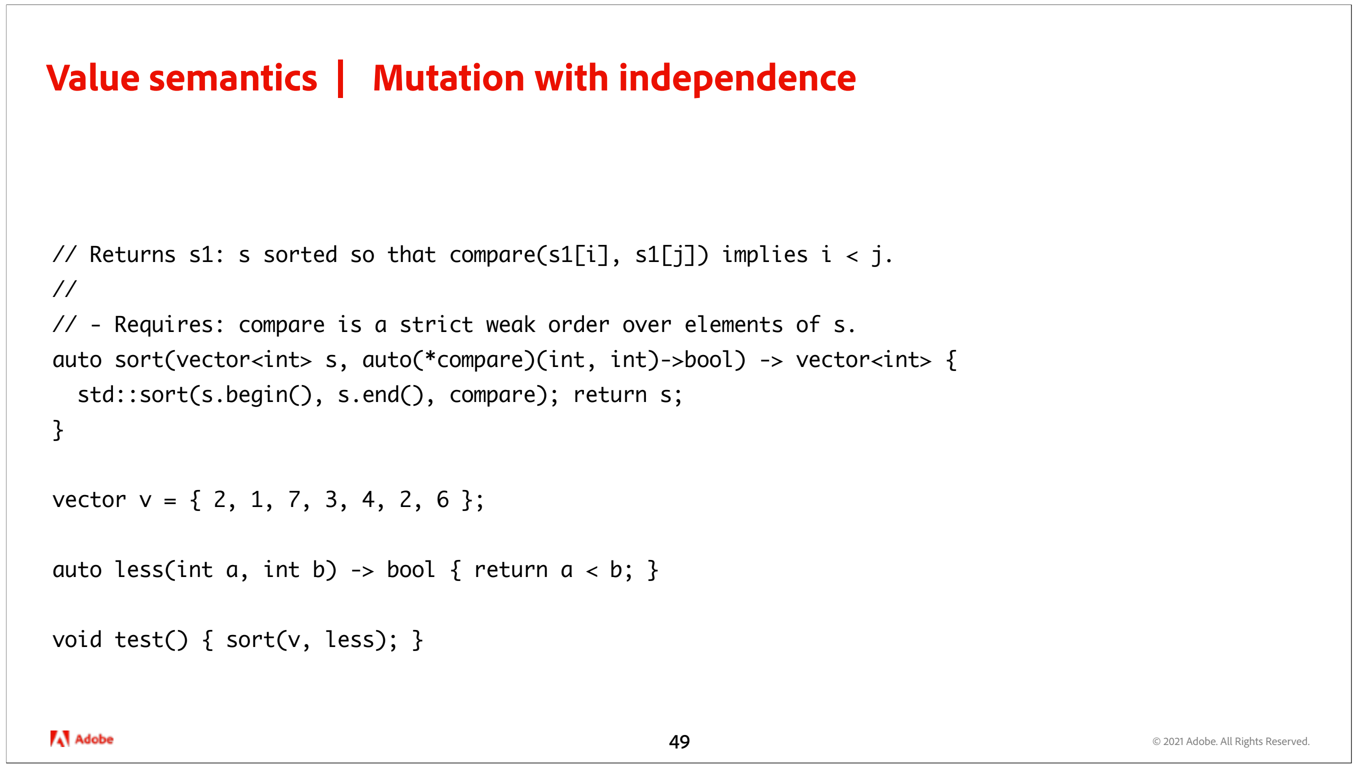
And now we can drop this requirement.

Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
// - Requires: compare does not modify s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.

auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```



Value semantics | Mutation with independence

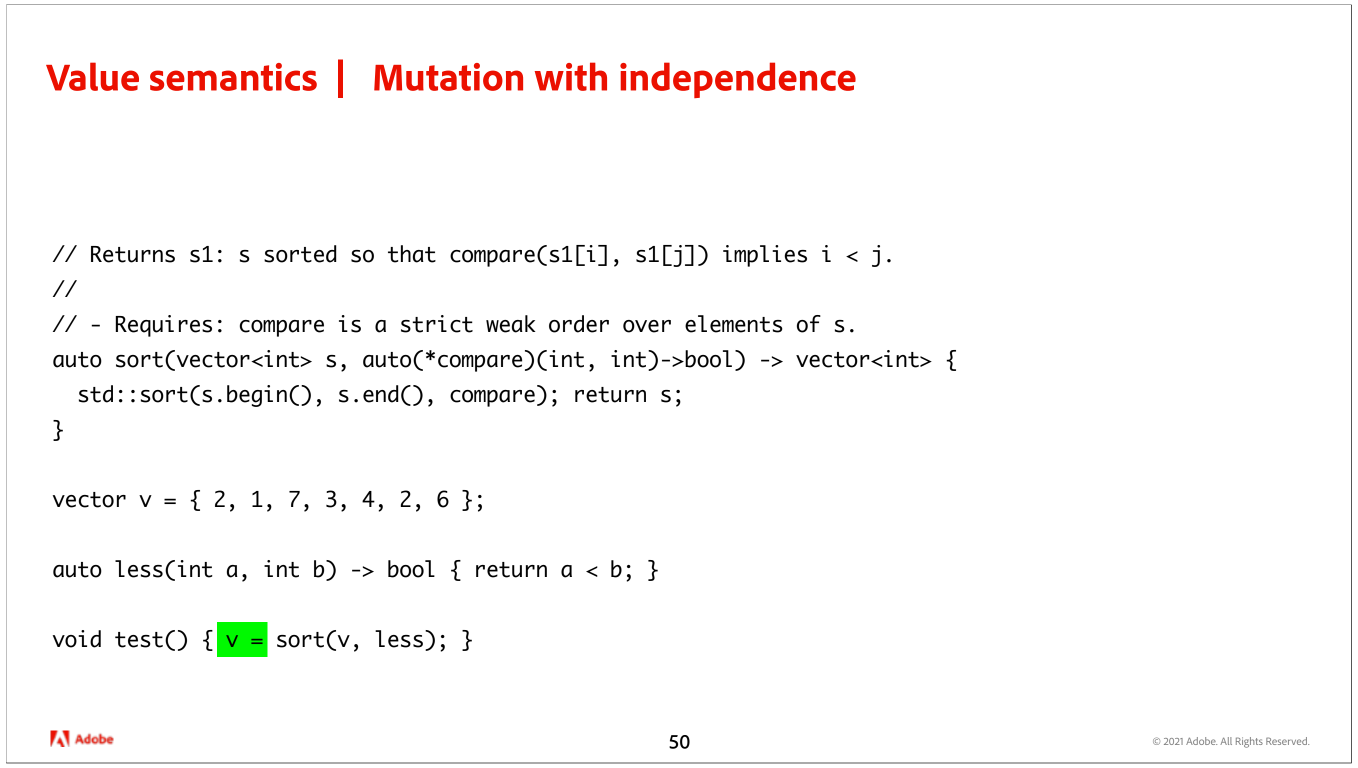
```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Adobe

and reassign v

Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```



Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { v = sort(v, less); }
```

This form is what's known as a "functional update," and there's a general transformation you can use on any mutating function.

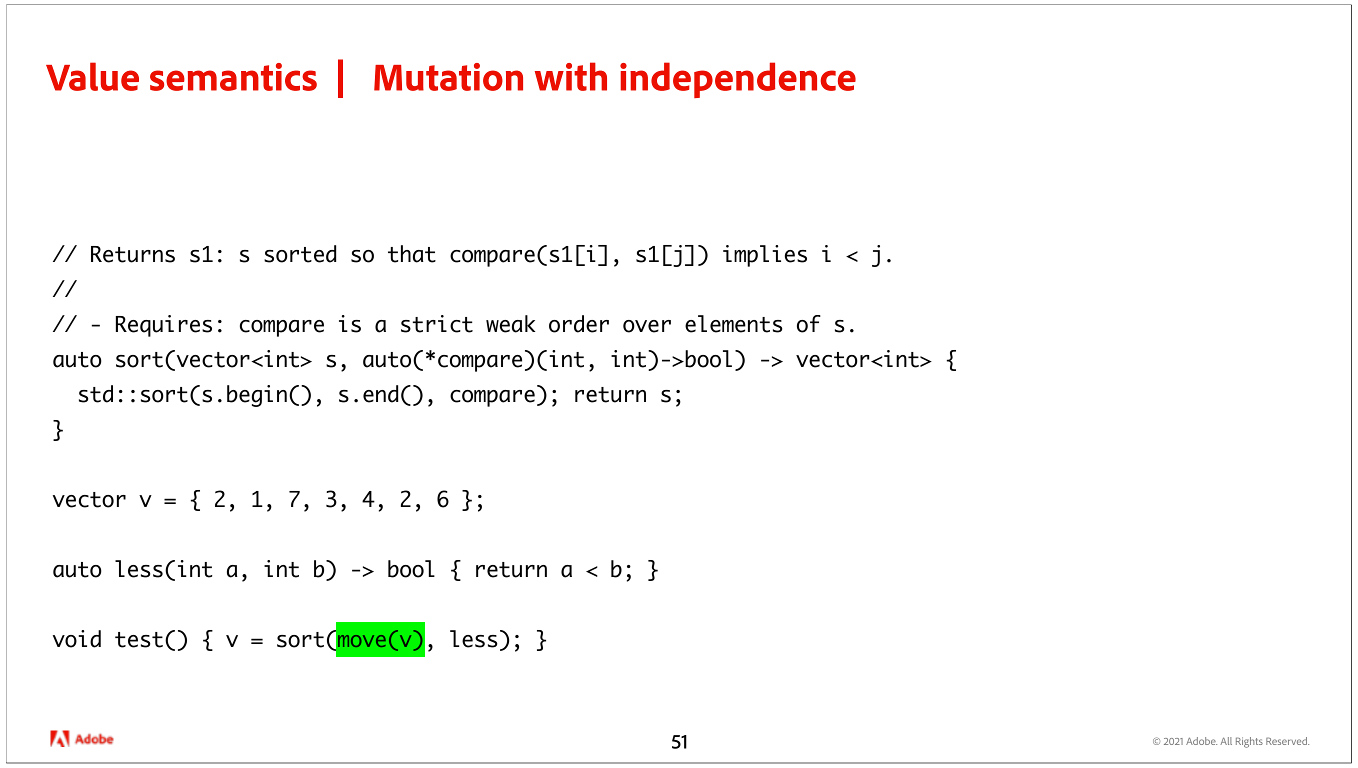
Of course because this is C++ and we need to explain everything to the compiler, so there's one more thing we have to do to make it efficient.

Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { v = sort(v, less); }
```

Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { v = sort(v, less); }
```



Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { v = sort(move(v), less); }
```

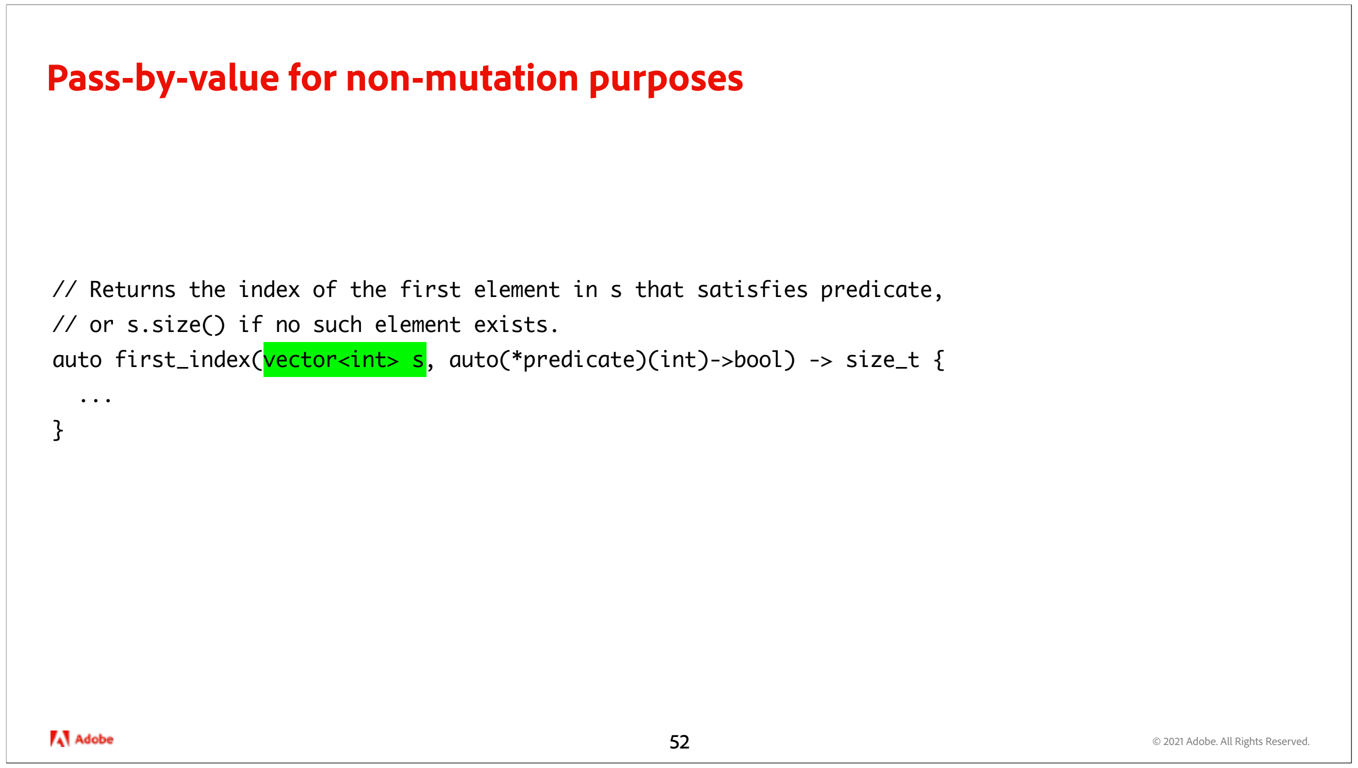
Pass-by-value is part of C++'s built-in support for value semantics, and we've leveraged it to do mutation with true independence.

Of course, nobody's gonna do all their mutations using functional update form, because it's a hassle.

Now here's something else nobody's going to do.

Value semantics | Mutation with independence

```
// Returns s1: s sorted so that compare(s1[i], s1[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> s, auto(*compare)(int, int)->bool) -> vector<int> {
  std::sort(s.begin(), s.end(), compare); return s;
}

vector v = { 2, 1, 7, 3, 4, 2, 6 };

auto less(int a, int b) -> bool { return a < b; }

void test() { v = sort(move(v), less); }
```



Pass-by-value for non-mutation purposes

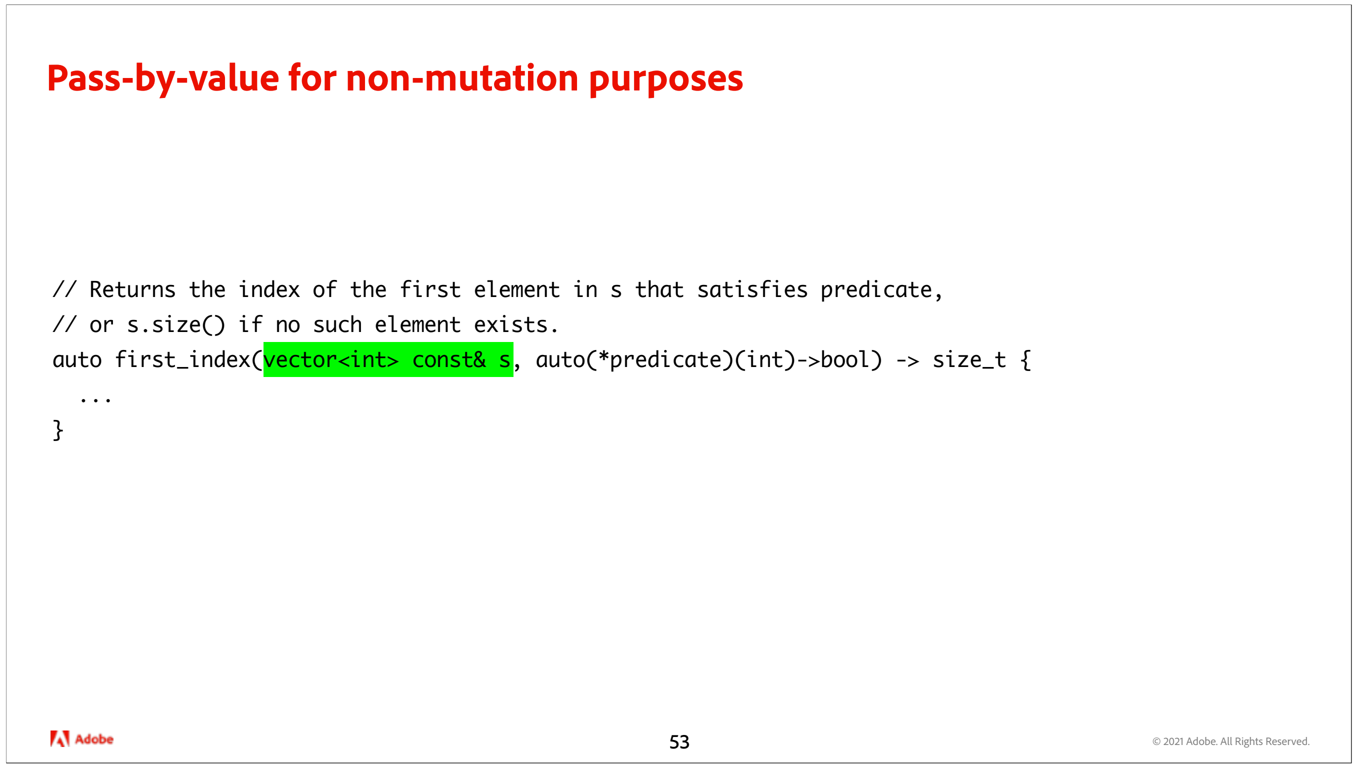
```
// Returns the index of the first element in s that satisfies predicate,
// or s.size() if no such element exists.
auto first_index(vector<int> s, auto(*predicate)(int)->bool) -> size_t {
  ...
}
```

Adobe

52

This declares a simple non-mutating function, written in the most straightforward way.

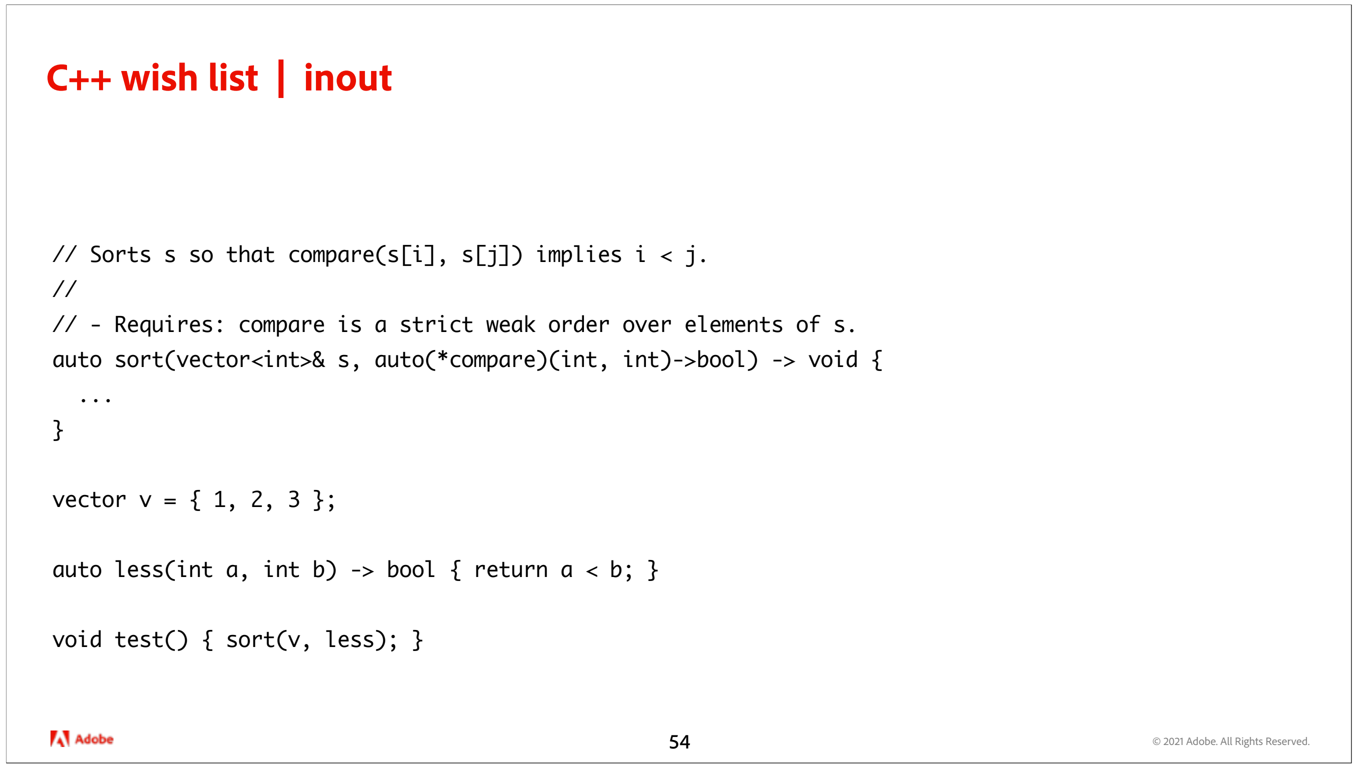
But if you take this function into a code review, someone will helpfully point out the expensive copy that's implied by pass-by-value. And they'll tell you to rewrite it using a const reference.



Pass-by-value for non-mutation purposes

```
// Returns the index of the first element in s that satisfies predicate,
// or s.size() if no such element exists.
auto first_index(vector<int> const& s, auto(*predicate)(int)->bool) -> size_t {
  ...
}
```

The irony here is that now we've given up independence. That const reference doesn't prevent the predicate—or any other code called during the execution of first_index, maybe in a different thread—from mutating the thing s refers to.



C++ wish list | inout

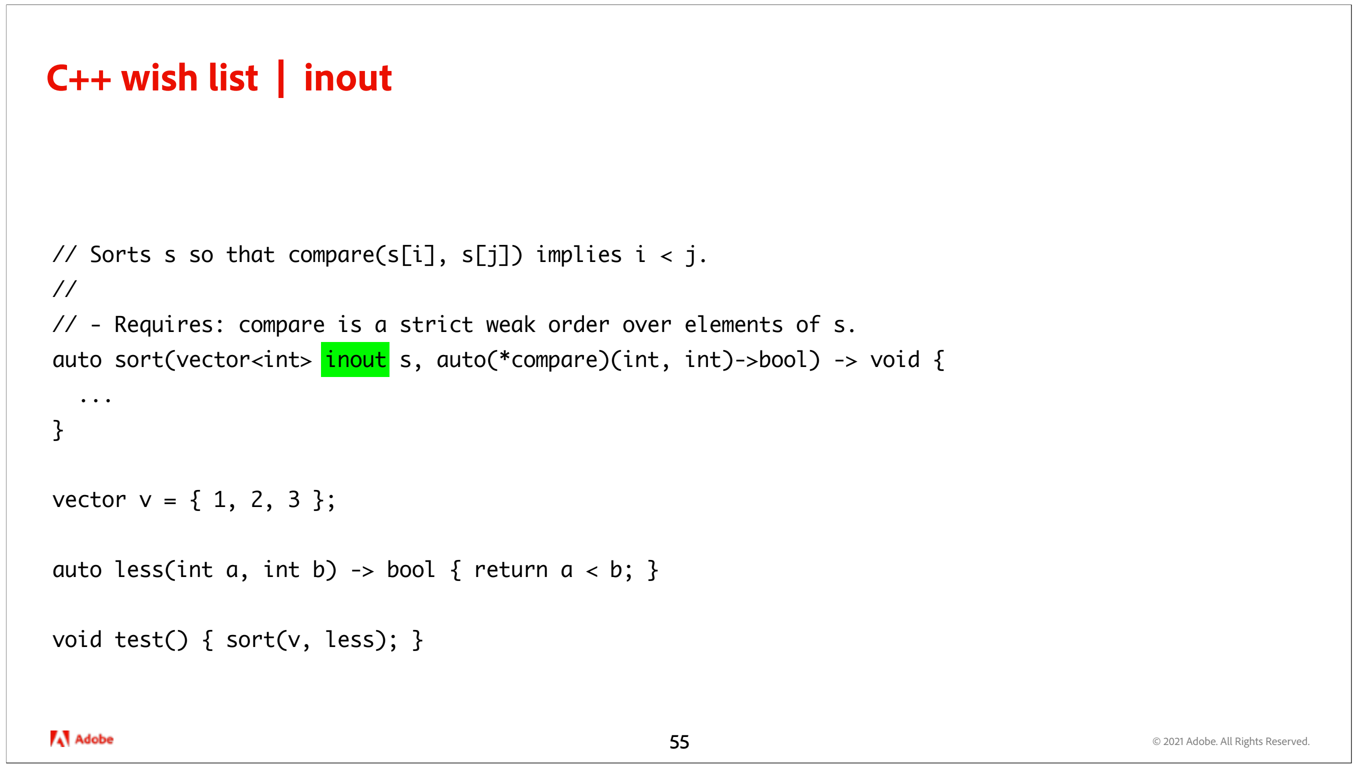
```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  ...
}

vector v = { 1, 2, 3 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

But all this rewriting is a pain. Suppose we had a language feature that would give us efficient functional update semantics, with the move and everything else, but avoid reassignment syntax?

I'd call it "inout" So let's replace that reference with my new keyword, inout

C++ wish list | inout

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int>& s, auto(*compare)(int, int)->bool) -> void {
  ...
}

vector v = { 1, 2, 3 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```



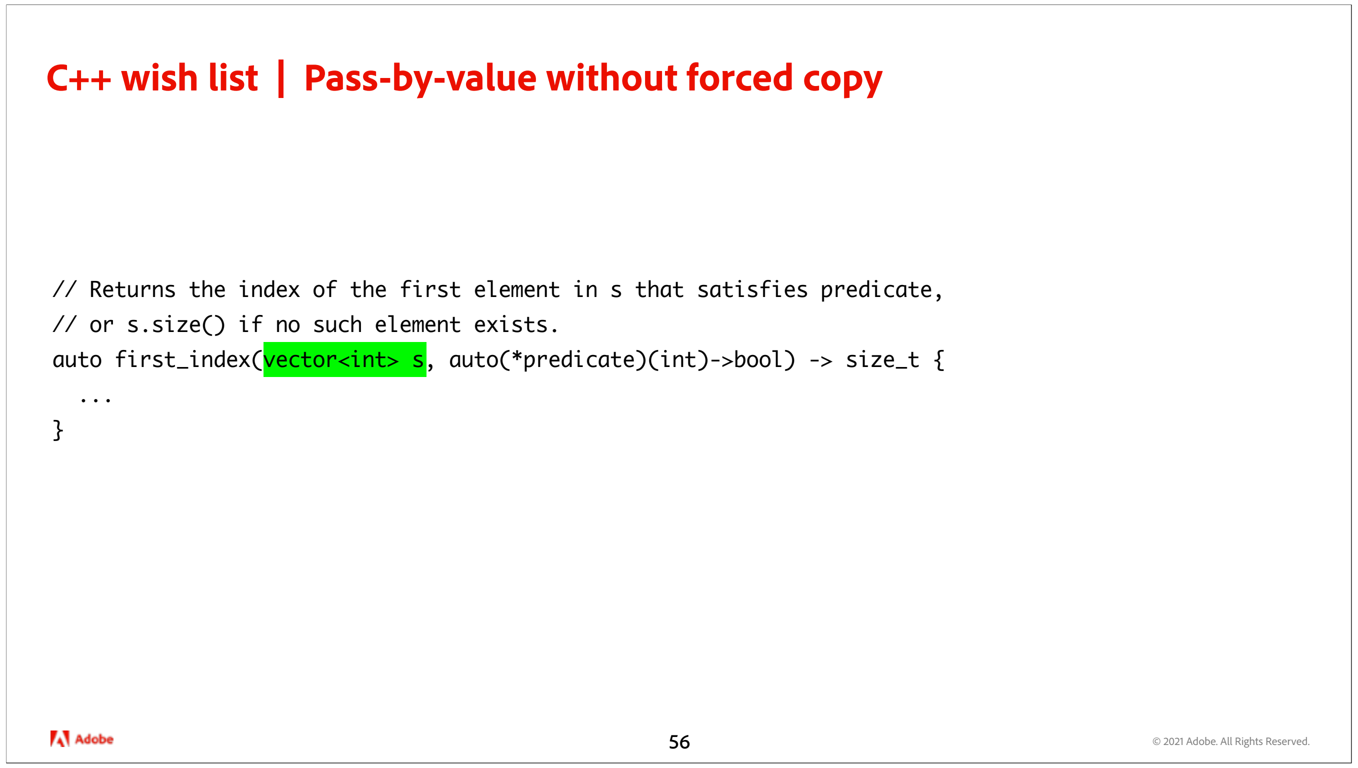
C++ wish list | inout

```
// Sorts s so that compare(s[i], s[j]) implies i < j.
//
// - Requires: compare is a strict weak order over elements of s.
auto sort(vector<int> inout s, auto(*compare)(int, int)->bool) -> void {
  ...
}

vector v = { 1, 2, 3 };

auto less(int a, int b) -> bool { return a < b; }

void test() { sort(v, less); }
```

Adobe

The compiler would rewrite my declaration of sort to use pass-by-value and return the updated result, and would rewrite my call to sort as a functional update.

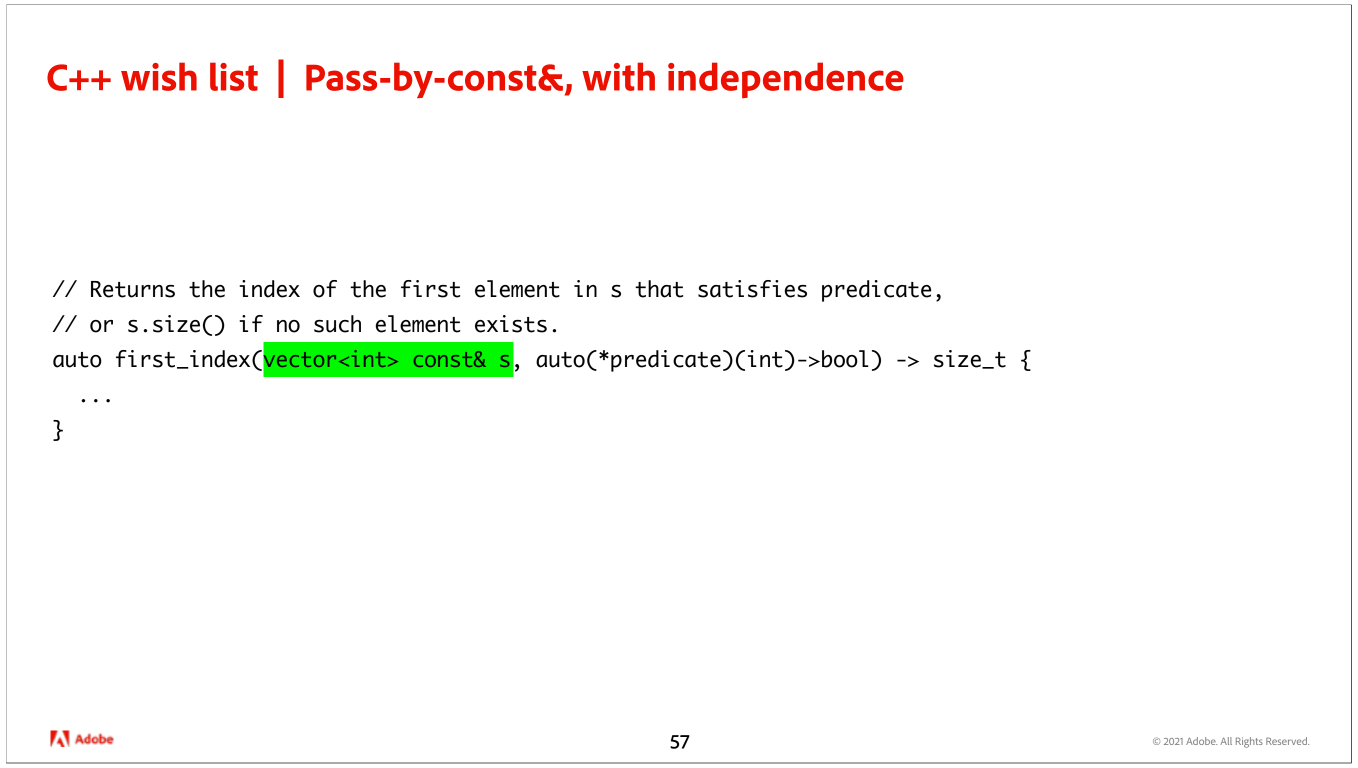
Anyway, this is basically the mutation model that Swift uses (except that there's no need to move).



C++ wish list | Pass-by-value without forced copy

```
// Returns the index of the first element in s that satisfies predicate,
// or s.size() if no such element exists.
auto first_index(vector<int> s, auto(*predicate)(int)->bool) -> size_t {
  ...
}
```

And there's a matching convention called 'in' for doing pass-by-value without a forced copy.

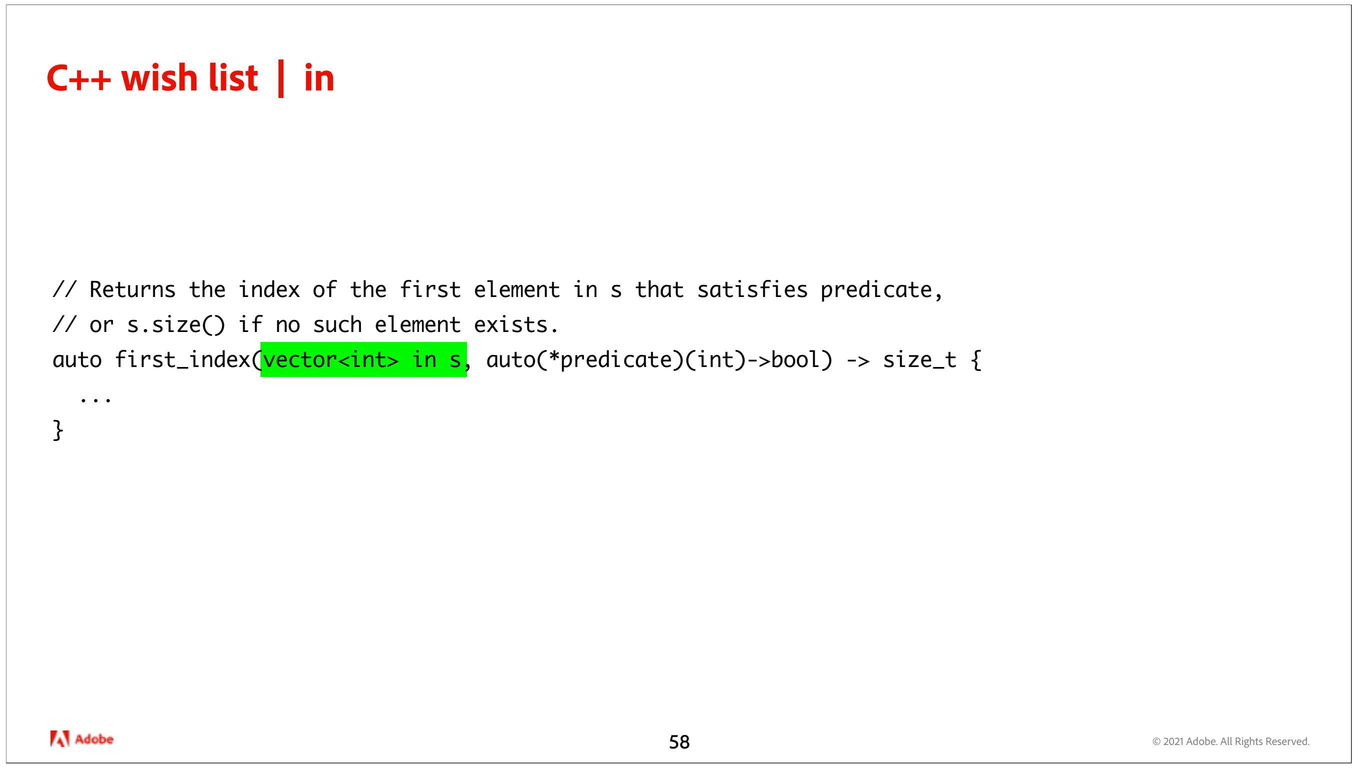


C++ wish list | Pass-by-const&, with independence

```
// Returns the index of the first element in s that satisfies predicate,
// or s.size() if no such element exists.
auto first_index(vector<int> const& s, auto(*predicate)(int)->bool) -> size_t {
  ...
}
```

Adobe

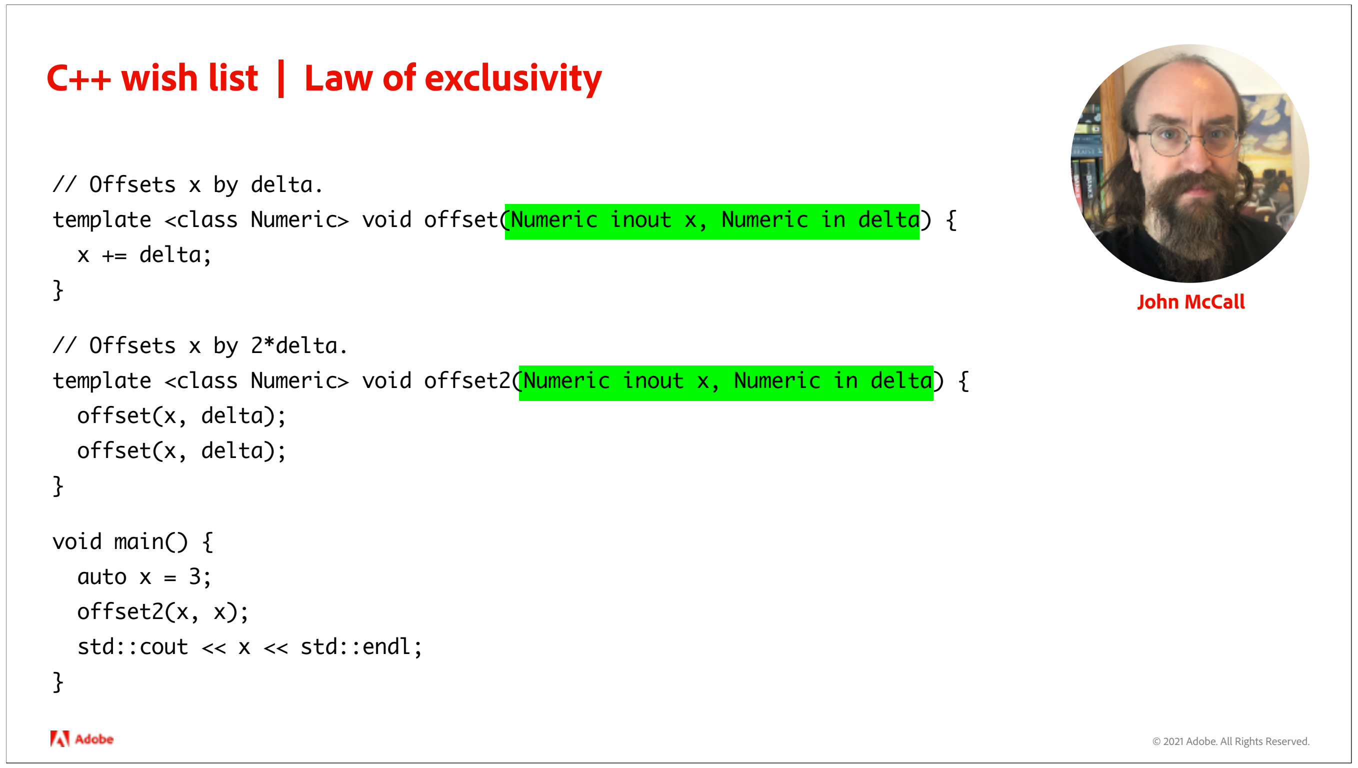
or you could think of it as pass-by-const&, but with independence.



C++ wish list | in

```
// Returns the index of the first element in s that satisfies predicate,
// or s.size() if no such element exists.
auto first_index(vector<int> in s, auto(*predicate)(int)->bool) -> size_t {
  ...
}
```

It looks like this.



C++ wish list | Law of exclusivity

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric inout x, Numeric in delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric inout x, Numeric in delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

John McCall

Adobe

The only other thing you need to make all of this work is a law of exclusivity, where the compiler doesn't let you pass to an inout argument unless it can prove it's the only access. This law is the invention of my good friend John McCall.

This is the original example, but using `inout` and `in` instead of references.

And if you use it, you get an error right where the problem originates.

C++ wish list | Law of exclusivity

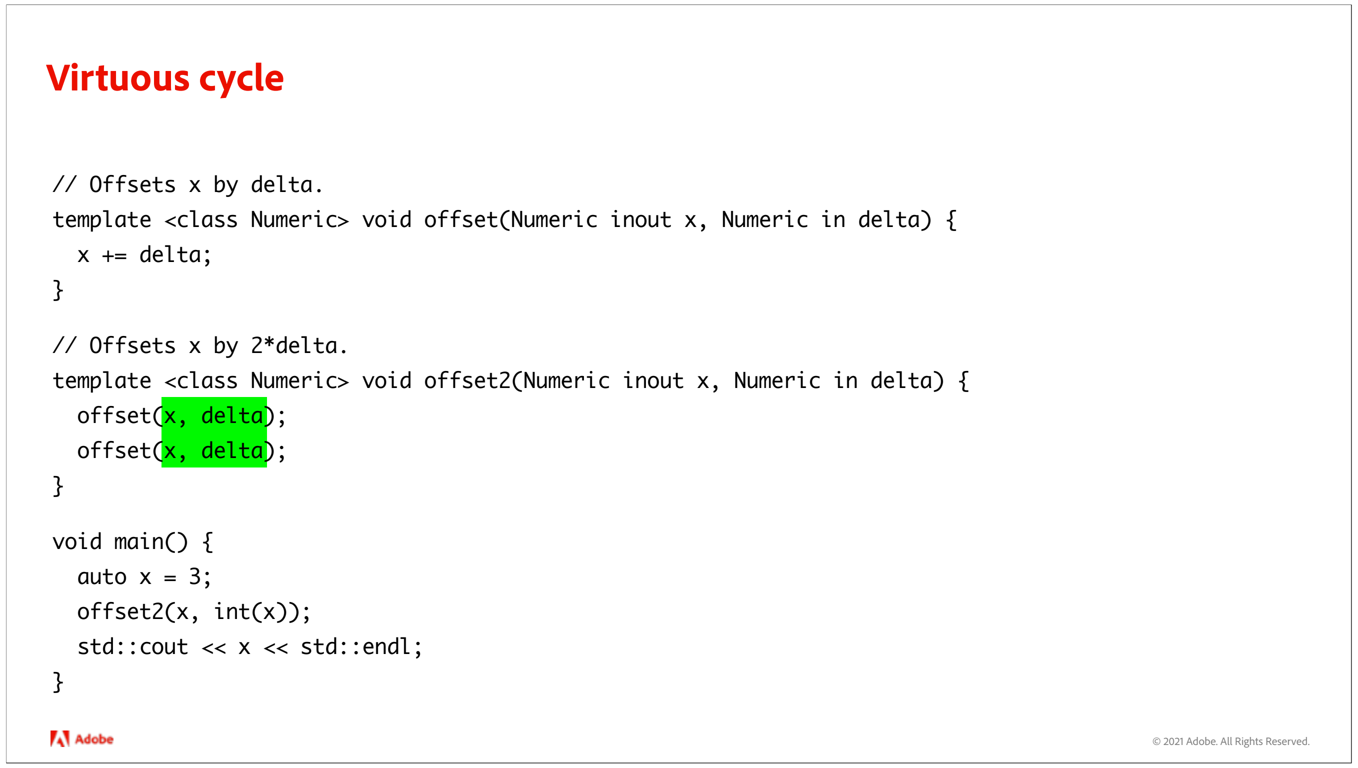
John McCall

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric inout x, Numeric in delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric inout x, Numeric in delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, x);
  std::cout << x << std::endl;
}
```

overlapping access to 'x' but modification requires exclusive access. Consider passing a copy of 'x' as the second parameter

Fix

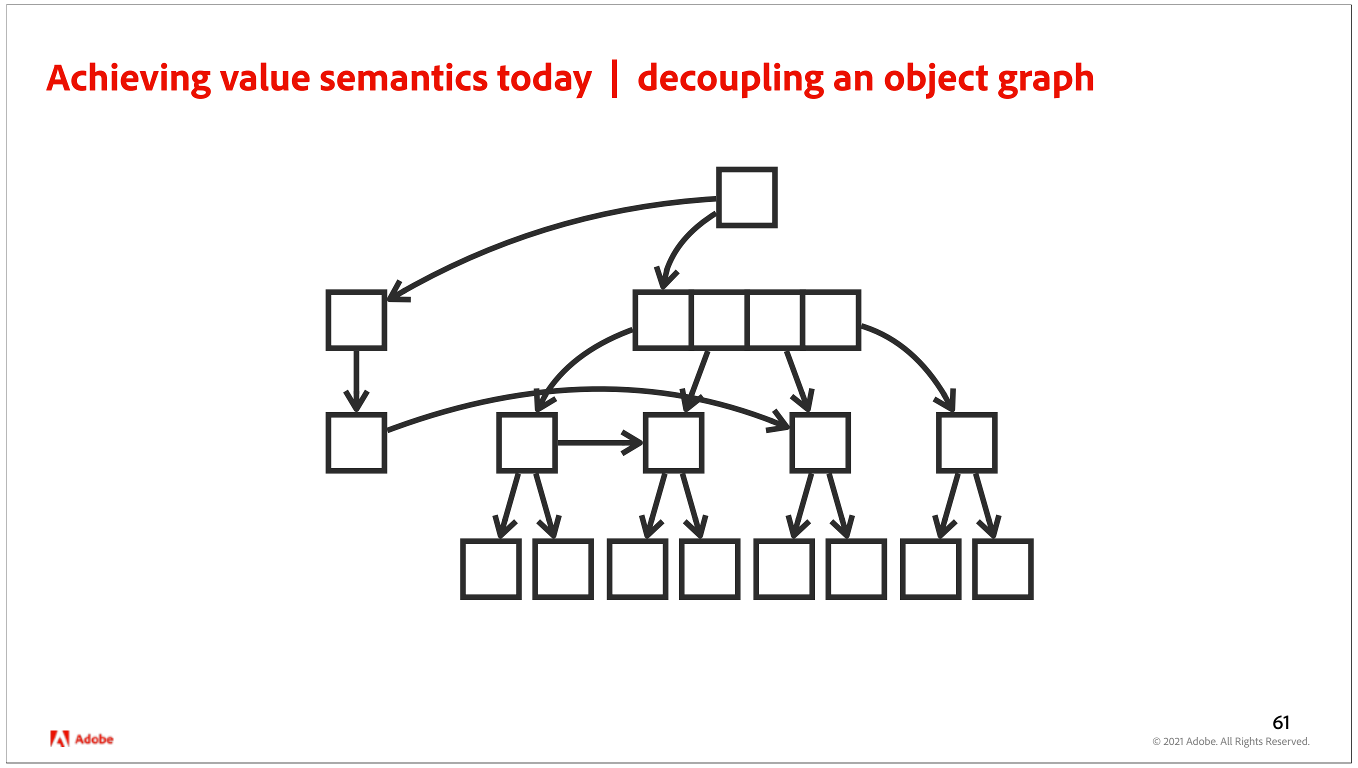
Adobe



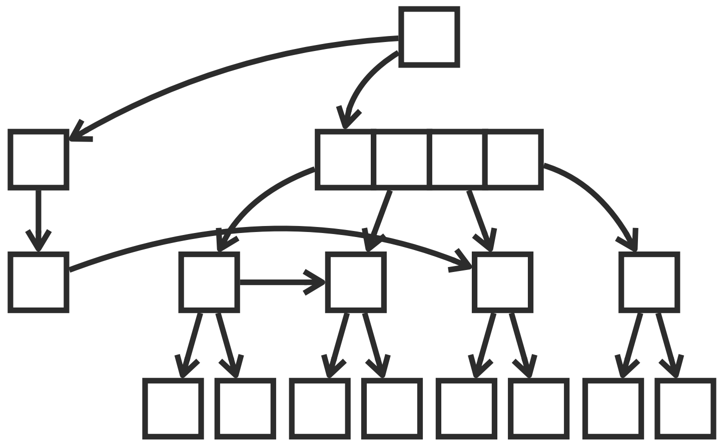
Virtuous cycle

```
// Offsets x by delta.
template <class Numeric> void offset(Numeric inout x, Numeric in delta) {
  x += delta;
}

// Offsets x by 2*delta.
template <class Numeric> void offset2(Numeric inout x, Numeric in delta) {
  offset(x, delta);
  offset(x, delta);
}

void main() {
  auto x = 3;
  offset2(x, int(x));
  std::cout << x << std::endl;
}
```

Lastly, let me point out there's a virtuous cycle at work here.
- inout s don't need to be moved to be unique and can be forwarded to inout
  or to in as references. ins can be forwarded to in. without checking.
The extra local reasoning provided means the compiler can optimize better, including eliminating more dynamic safety checks if you have them.



Achieving value semantics today | decoupling an object graph



So how can you achieve MVS today?

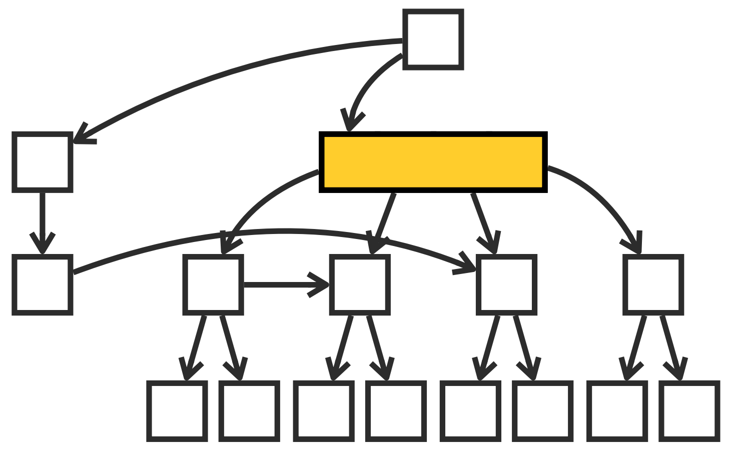
Say you have a mutable object graph like that one.

➡️ We have an array that of pointers to some data structures.
➡️ Some of the elements in that array may point to their siblings.
➡️ And we also have external references into the elements of the array.

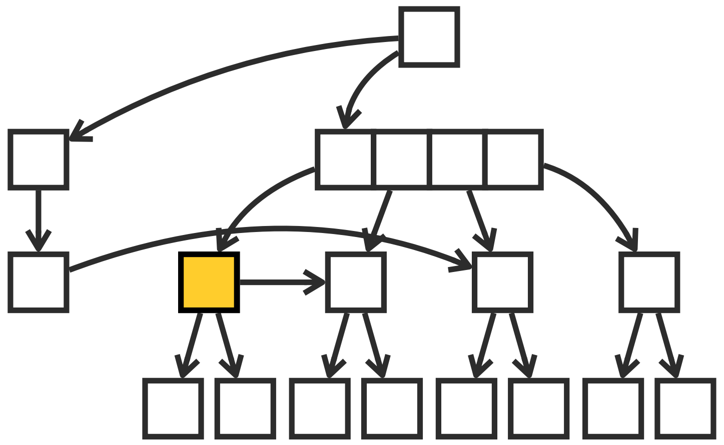
The point is, you found ourselves in some kind of messy tangled thing. So how do you start? Well, first, you identify the whole-part relationships in your graph. How do you do that exactly?

➡️

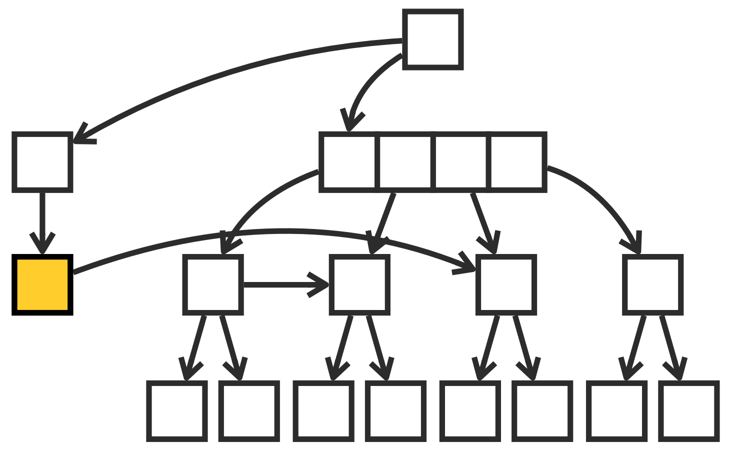
Achieving value semantics today | decoupling an object graph

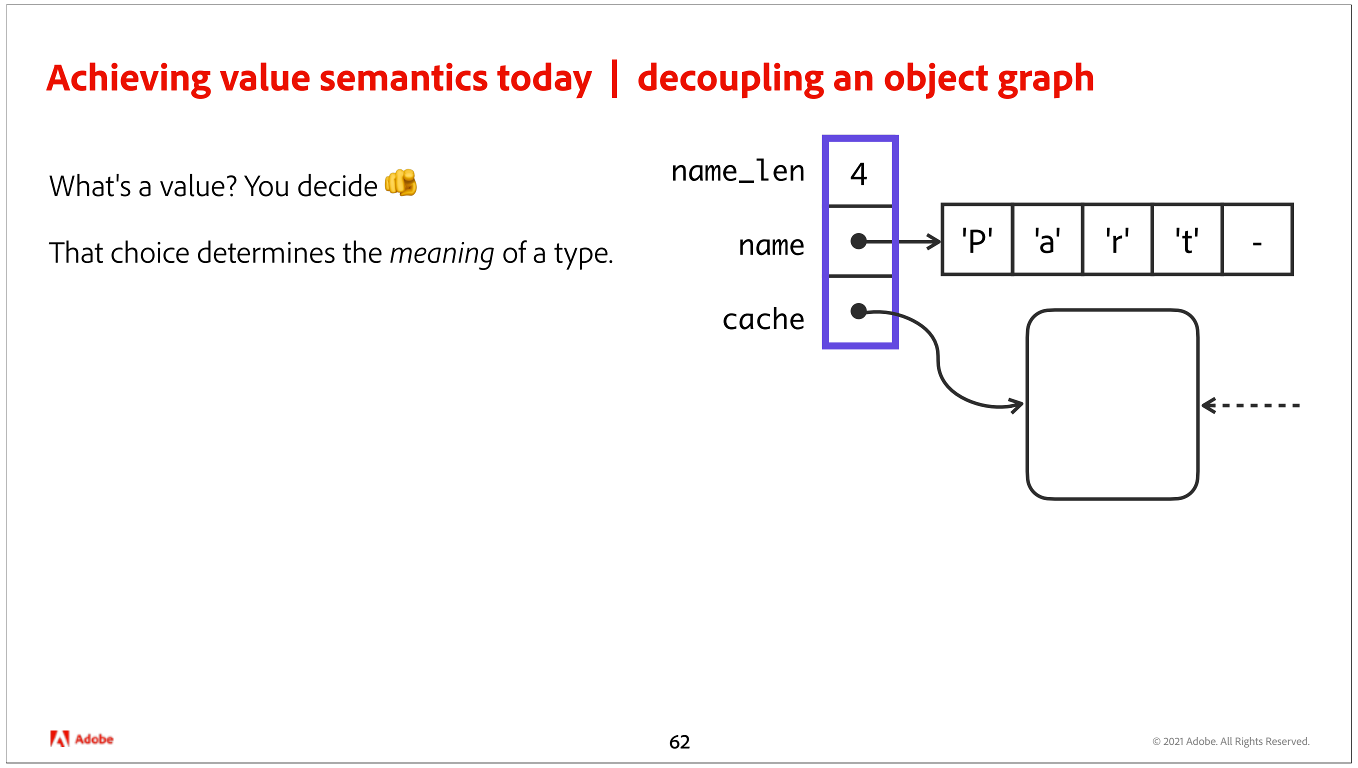


Achieving value semantics today | decoupling an object graph



Achieving value semantics today | decoupling an object graph

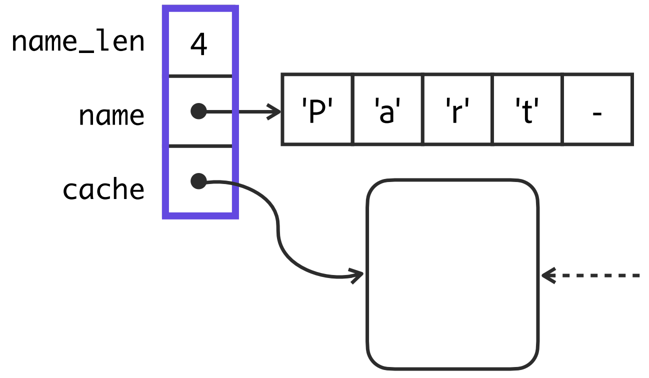




Achieving value semantics today | decoupling an object graph

What's a value? You decide 🫵

That choice determines the *meaning* of a type.

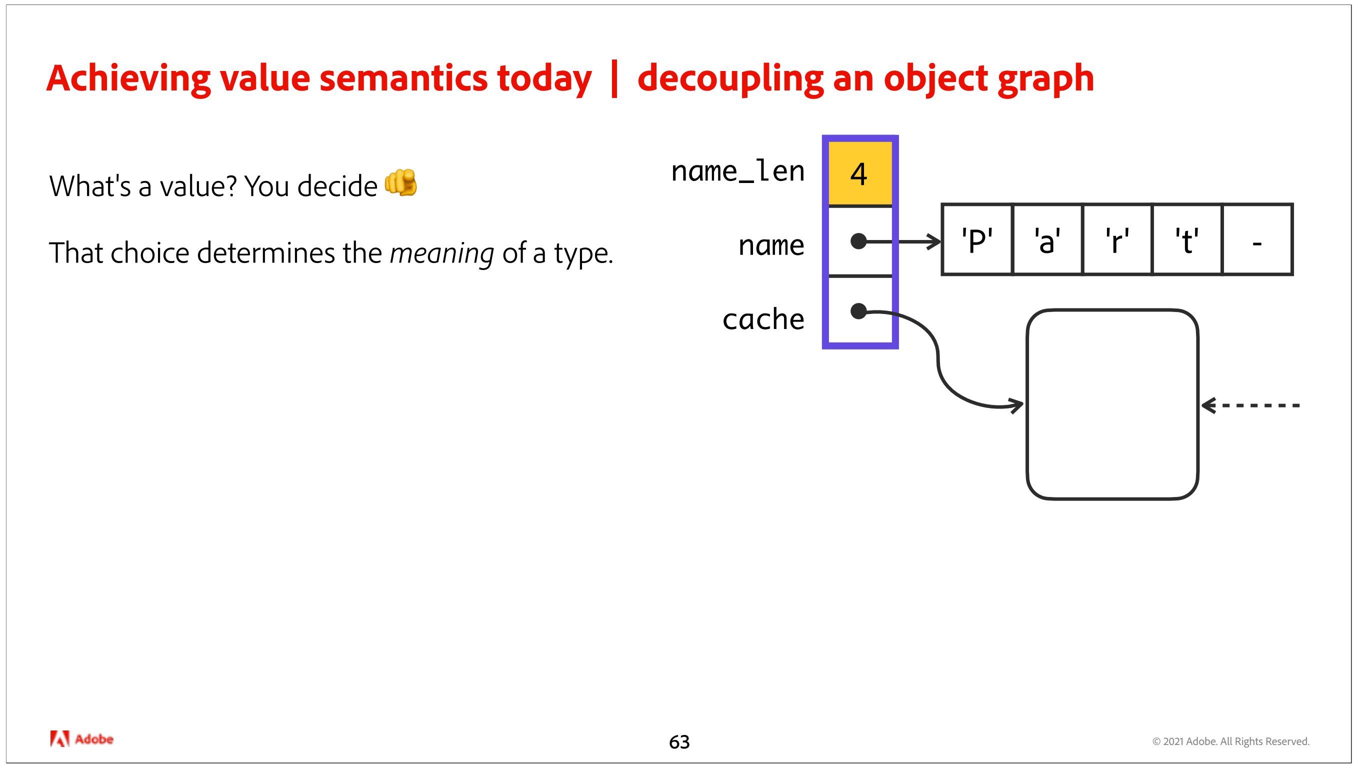


The problem is that the value of a type is in fact something that the type's author must decide. And what they decide reflects and determines what the type means.

Let's look at an example. The heavy purple box surrounds the object representation of a widget; the bytes in memory. But what is the value representation?
That depends on how widget's author understands its meaning. In this case, I'm the author, so I'm going to give you the answers, but the answers might be different for another type described by the same diagram.

First we have name_len, which determines the number of significant elements in the name array. The name is part of the value, so name_len is clearly a part.

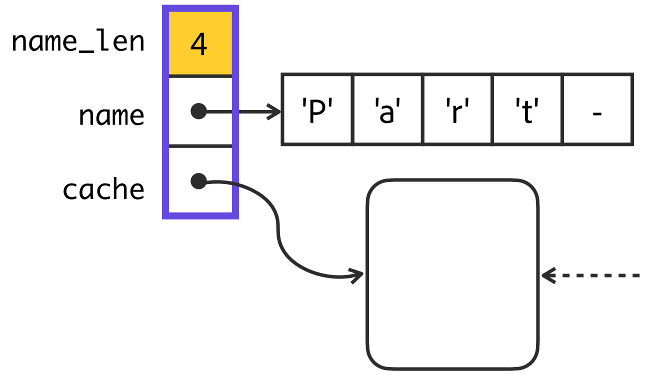
➡️



Achieving value semantics today | decoupling an object graph

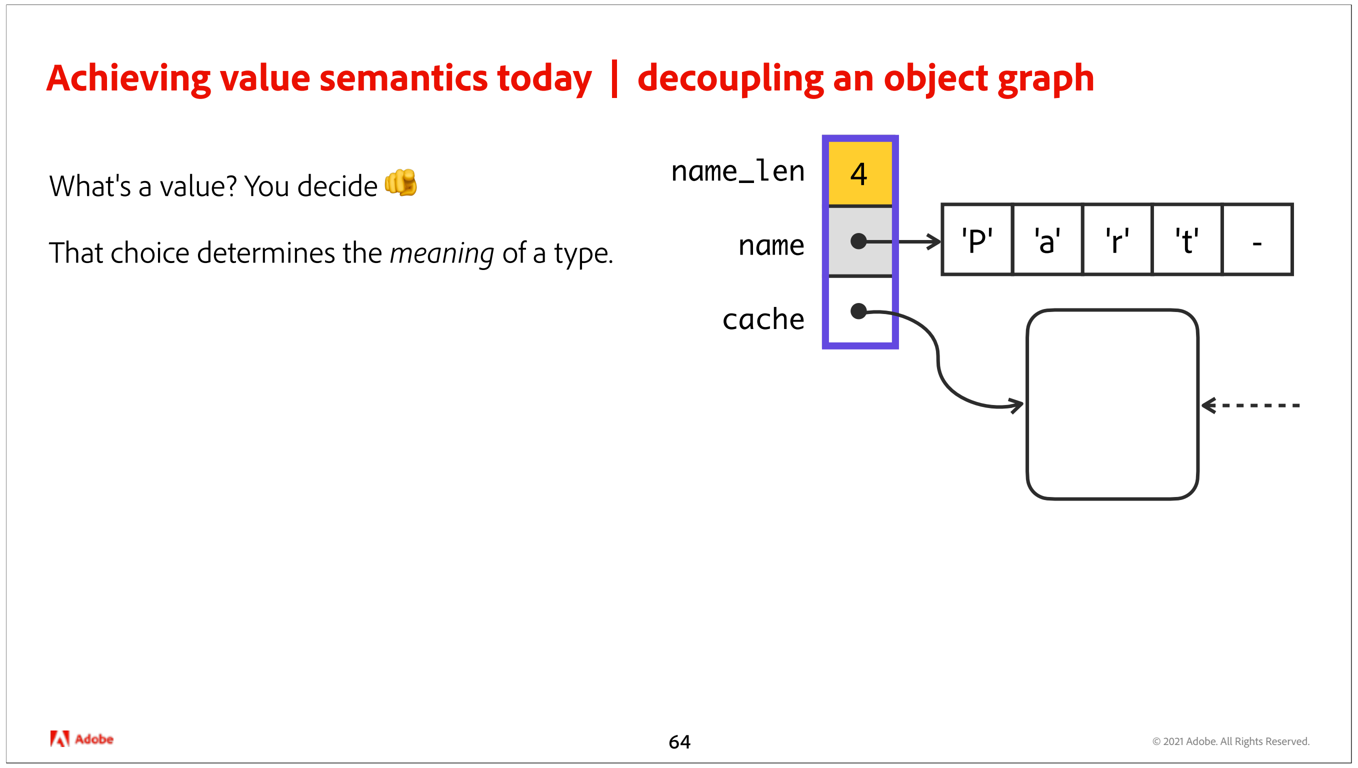
What's a value? You decide 👊

That choice determines the *meaning* of a type.



Next we have name itself. I consider two widgets with different pointers to equal name data of equal length to be equal. So name itself is not part of the value.

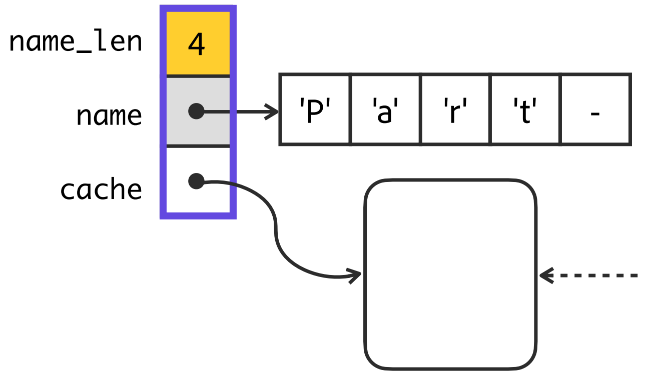
➡️



Achieving value semantics today | decoupling an object graph

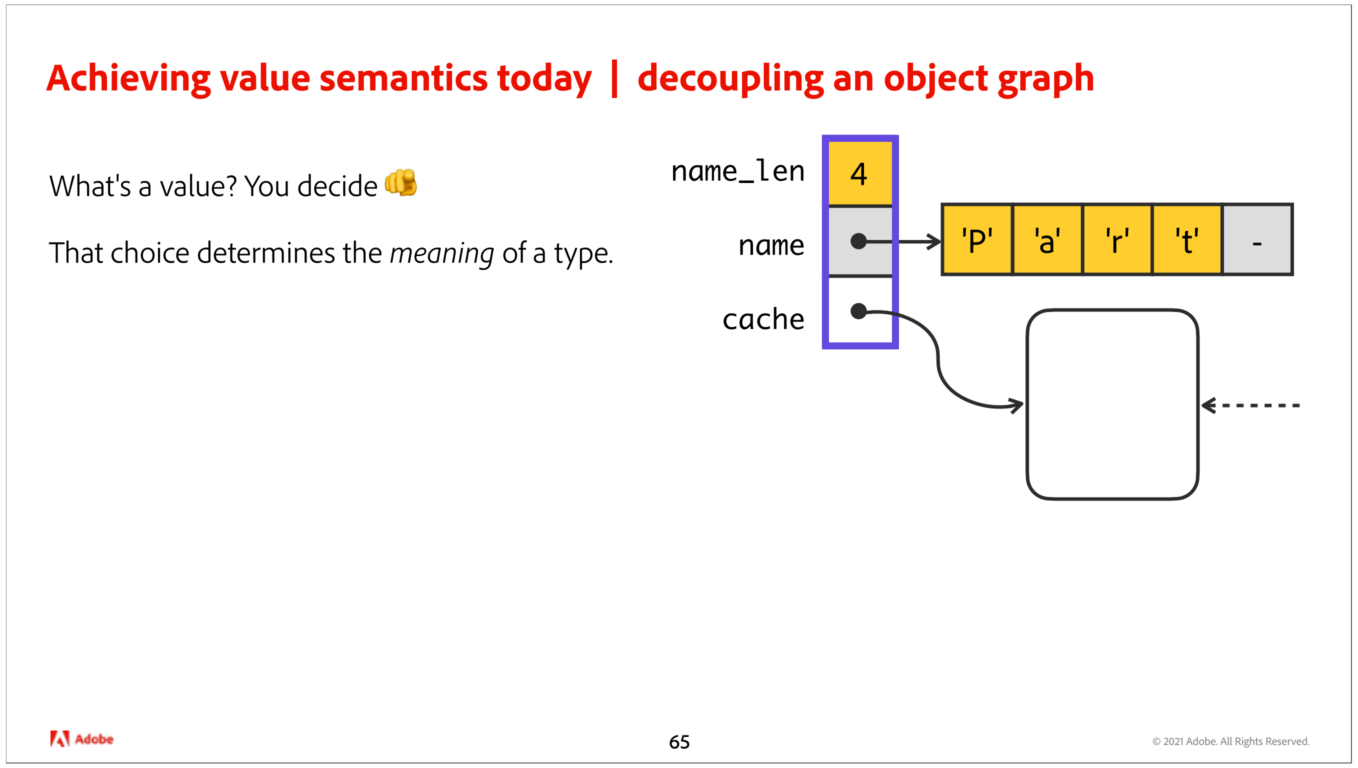
What's a value? You decide 👊

That choice determines the *meaning* of a type.



Next, there's the data pointed to by name. Exactly name_len elements of that data is part of the value.

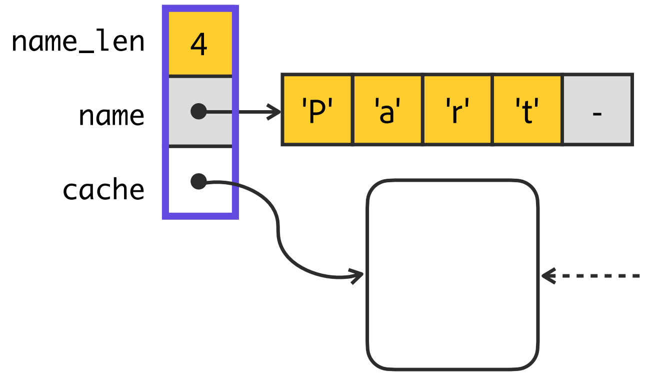
➡️



Achieving value semantics today | decoupling an object graph

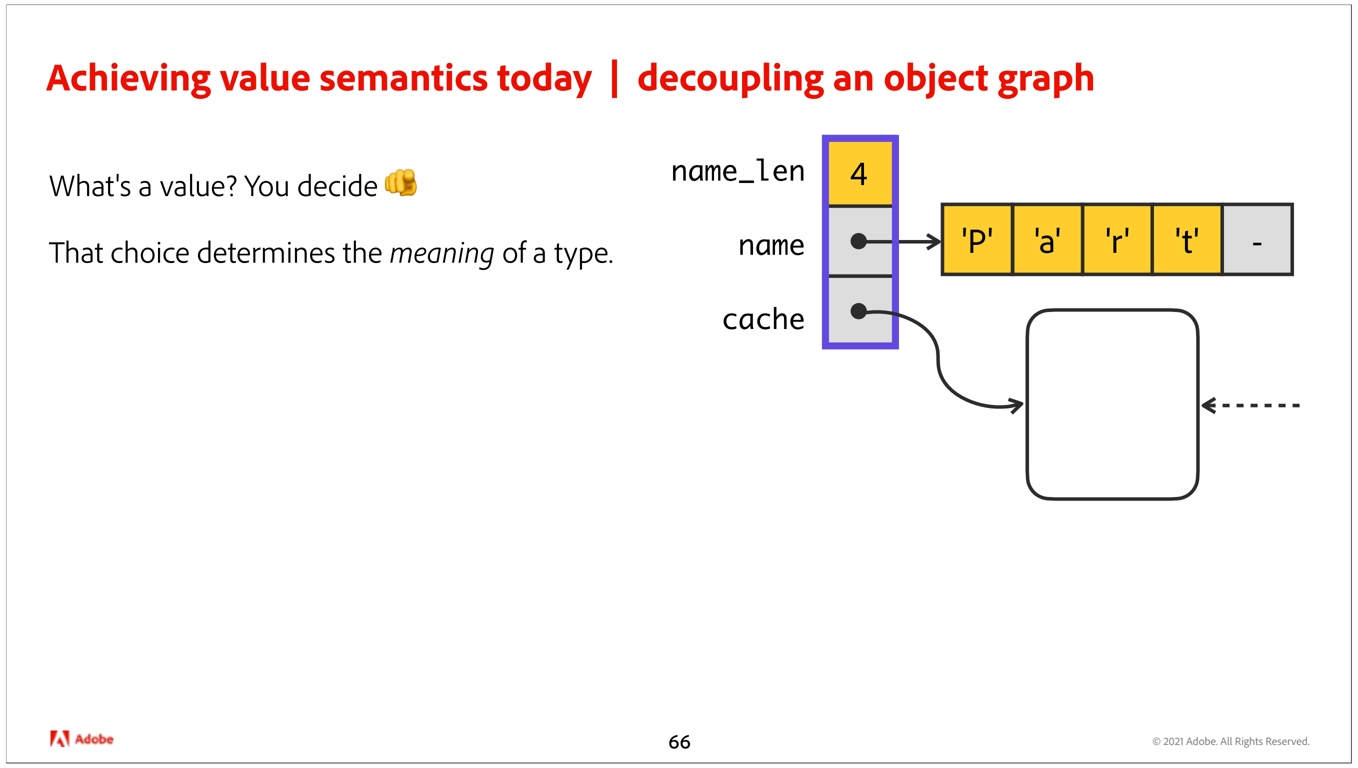
What's a value? You decide 🫵

That choice determines the *meaning* of a type.



Finally, we have a pointer to a shared cache. Widget uses the cache to respond to queries faster, but the cache doesn't otherwise affect the widget's behavior. So neither the cache nor its pointer are part of the value.

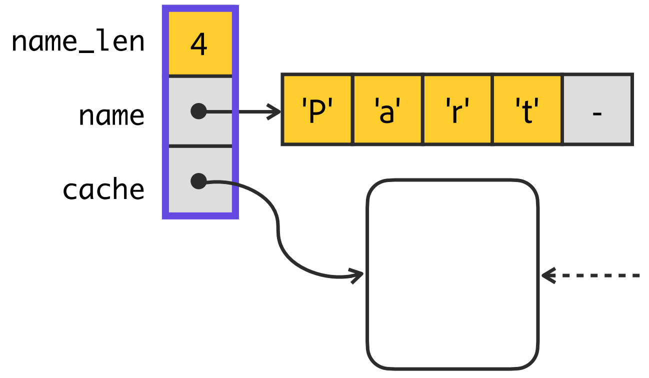
➡️



Achieving value semantics today | decoupling an object graph

What's a value? You decide 👊

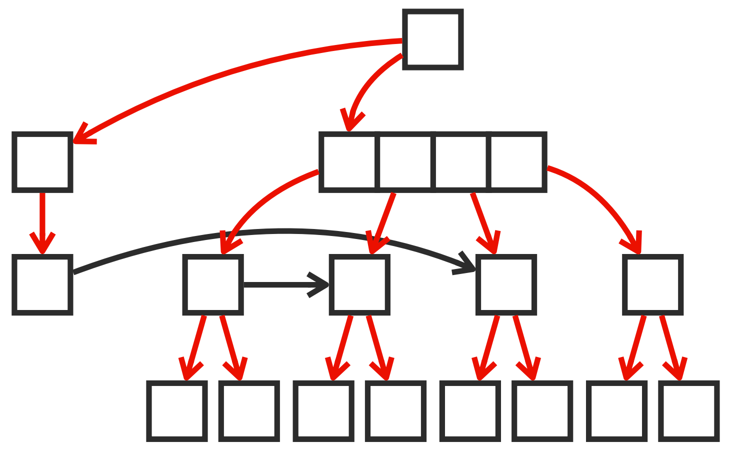
That choice determines the *meaning* of a type.



And voilà. I just determined the value of my type by identifying its whole/part relationships.

➡️

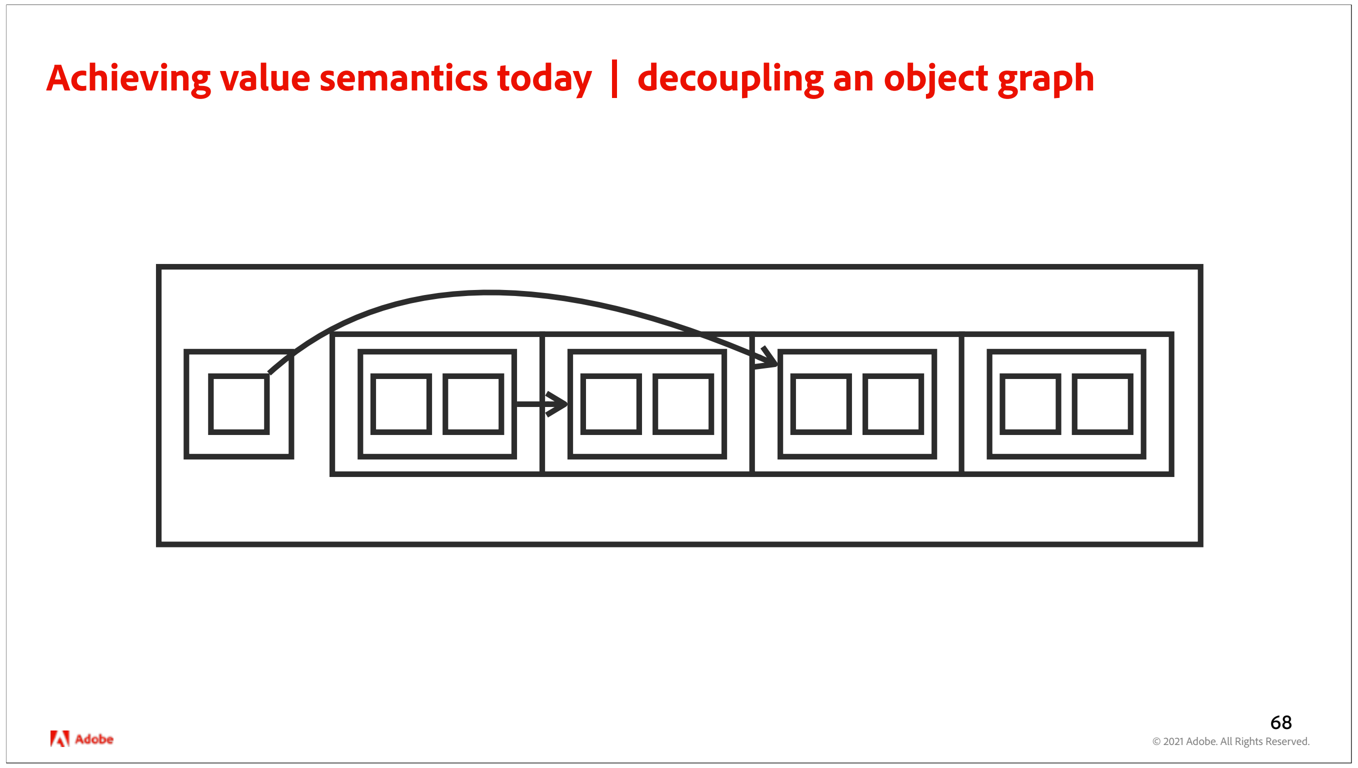
Achieving value semantics today | decoupling an object graph



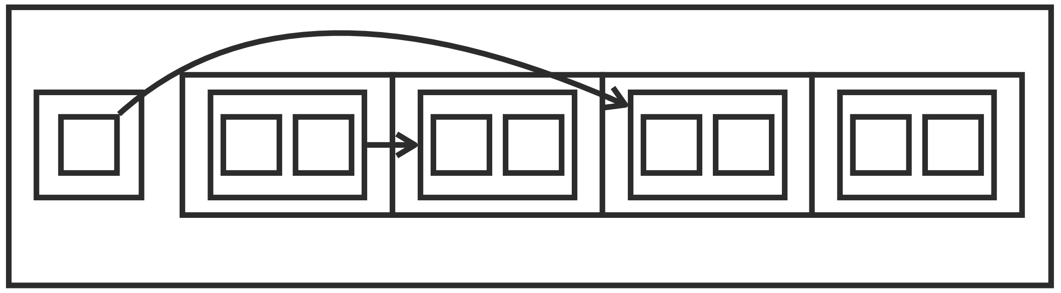
Now let's get back to our object graph. We've identified the whole/part relationships, which I have painted in red. Note that this necessarily forms a tree. Remember, whole/part relationships imply that there's a single access to a thing.

The next step is to turn that tree into nested boxes.

➡️



Achieving value semantics today | decoupling an object graph

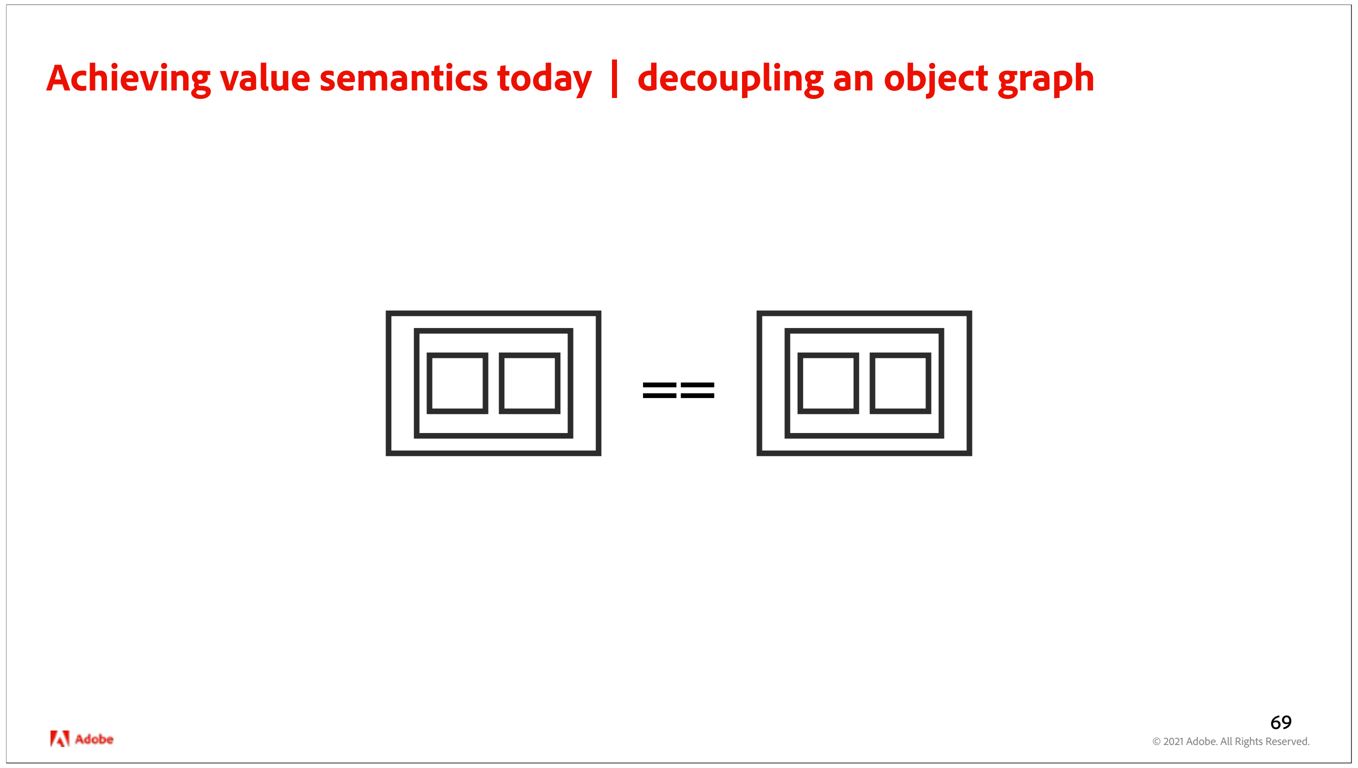


In that new mental view, the boxes are values and nesting represent whole/part relationships.

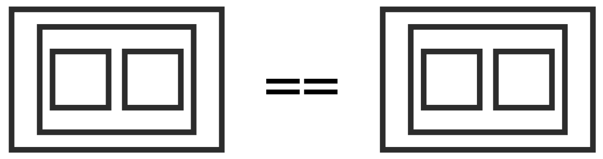
The arrows that remain denote the other kind of relationships, let's call them extrinsic relationships, which you have to represent another way.

Let's assume you stick to your pointers and references for the moment.

➡️

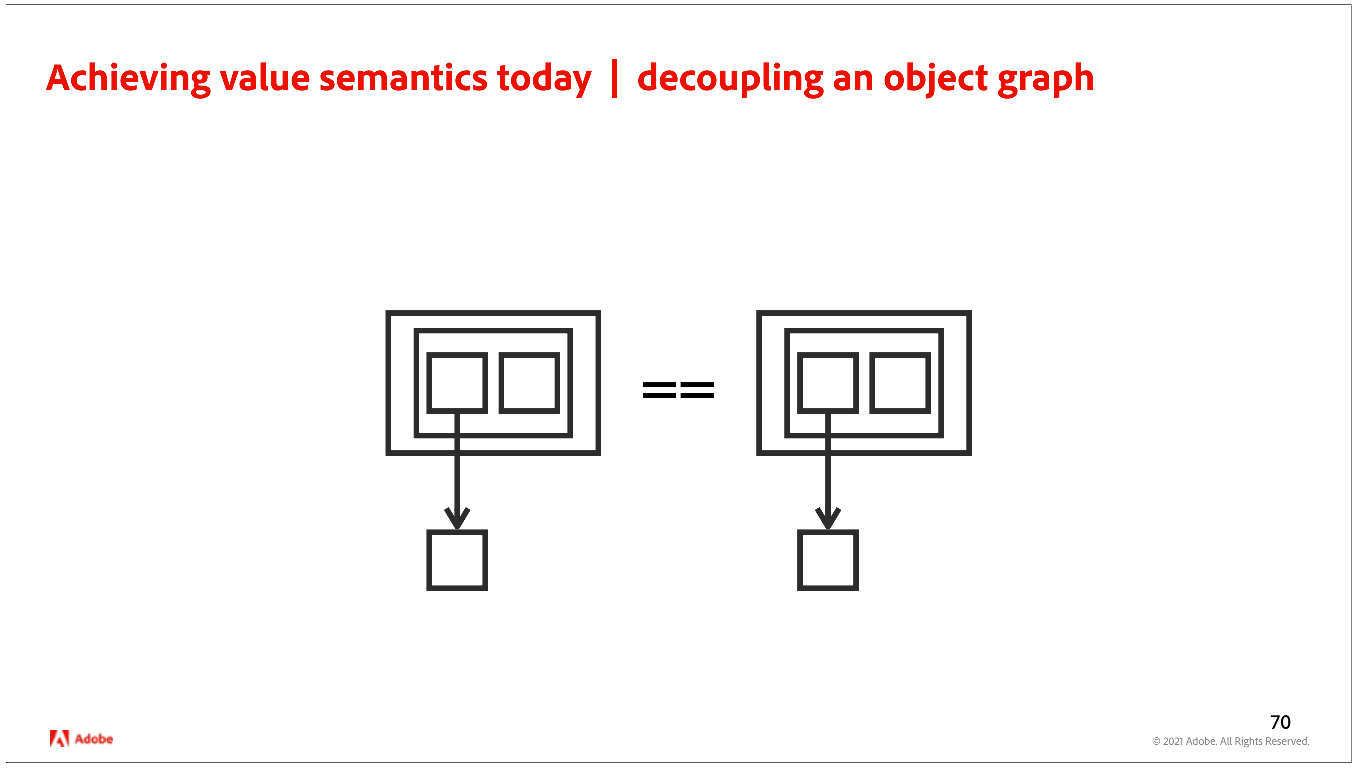


Achieving value semantics today | decoupling an object graph

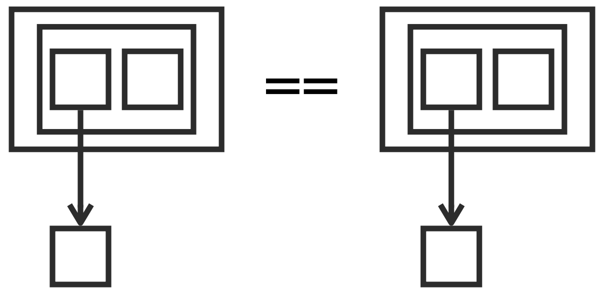


Because the boxes should represent values, you should now be able to write an equality operation between two boxes of the same type. That operation should make sense and, in the absence of extrinsic relationships, it should be synthesizable with `= default`: the equality of an aggregate is just the equality of its parts.

➡️



Achieving value semantics today | decoupling an object graph

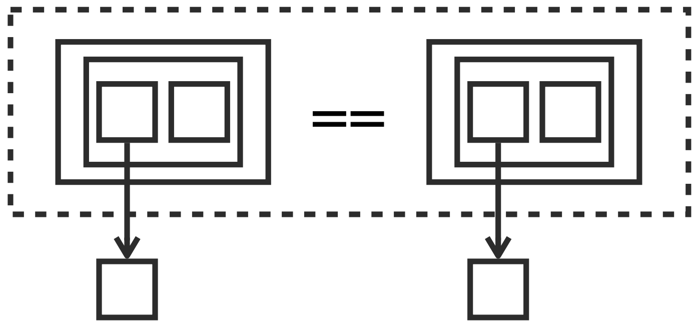


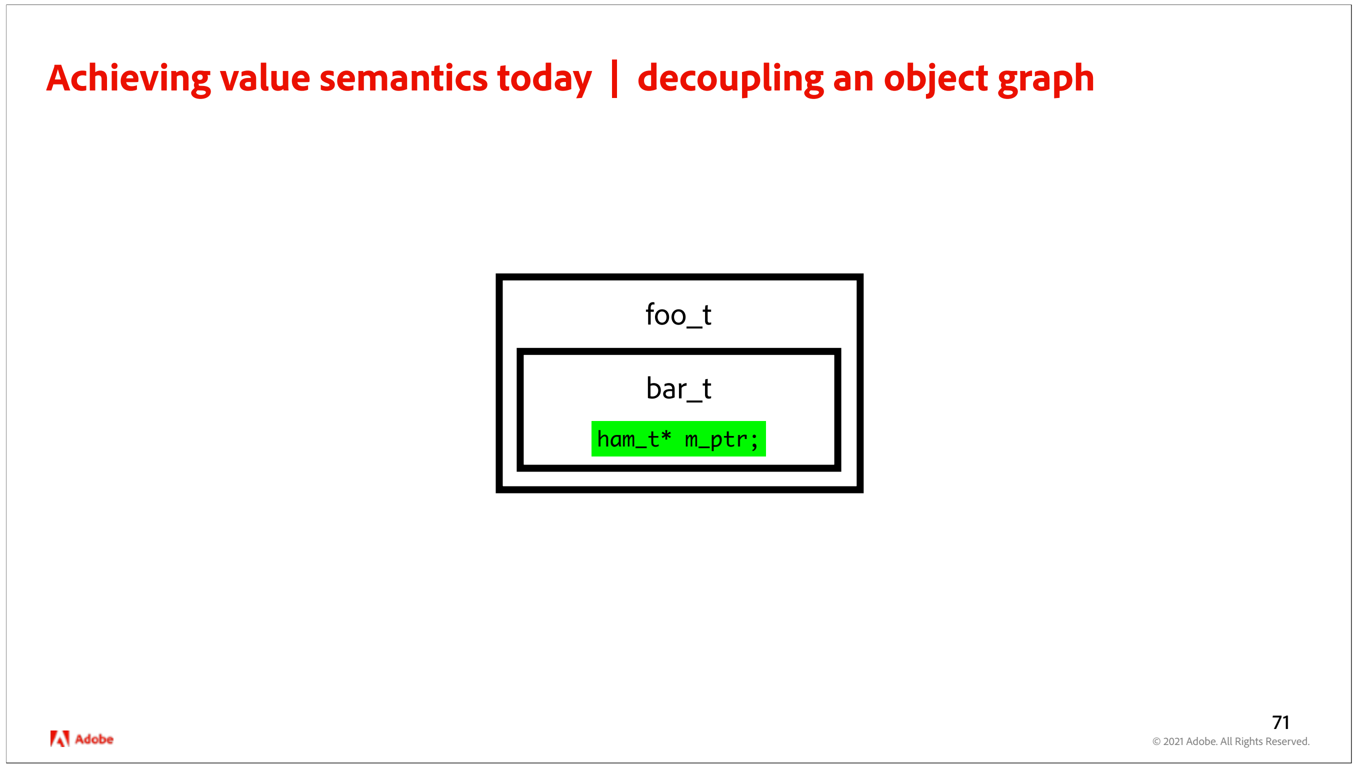
If you do have extrinsic relationships represented with pointers, then the values of the referred objects can't be part of the equality relation. ➡️ Only their identity can.

Think about it. If the value of those objects were part of the the equality relation, then they would form whole/part relationships.

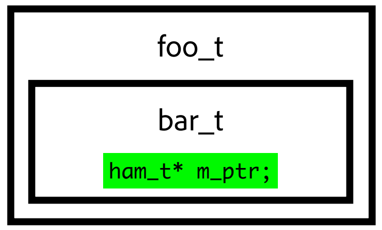
➡️

Achieving value semantics today | decoupling an object graph





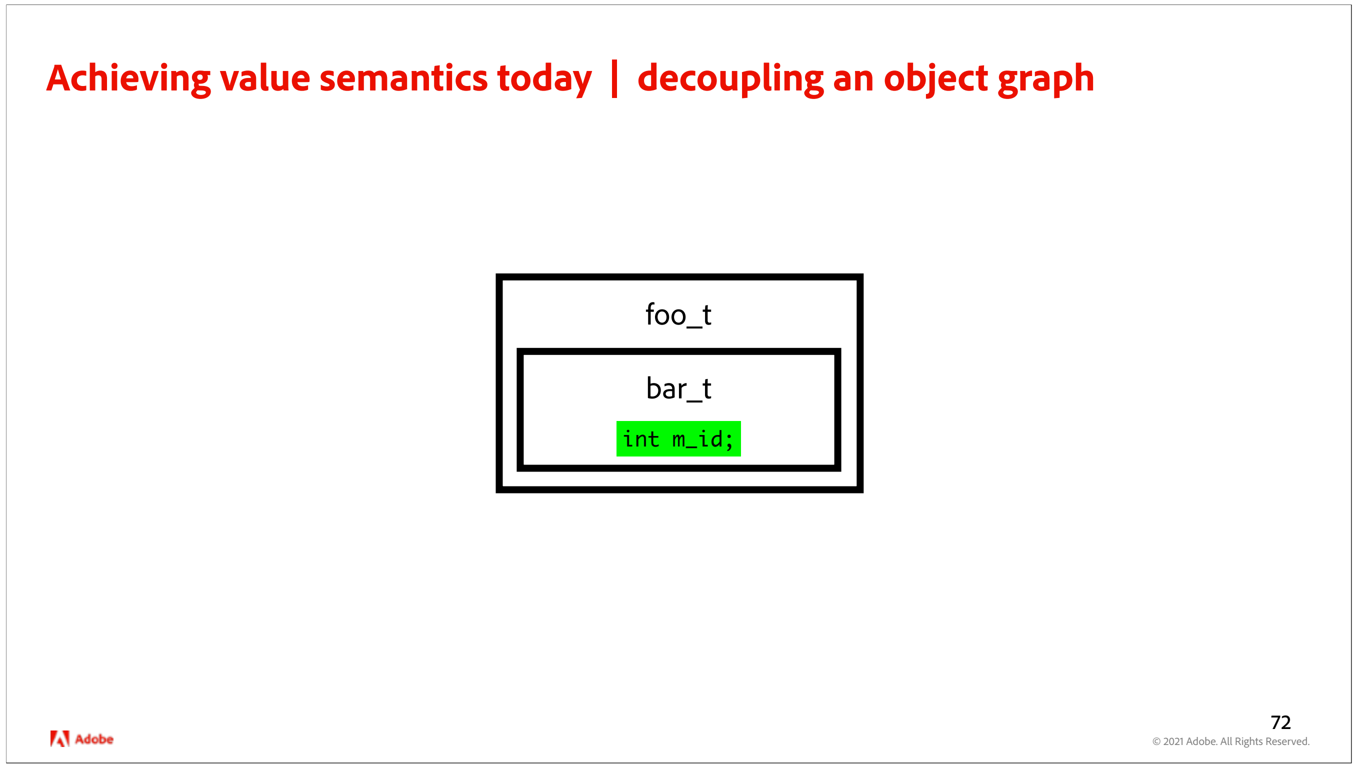
Achieving value semantics today | decoupling an object graph



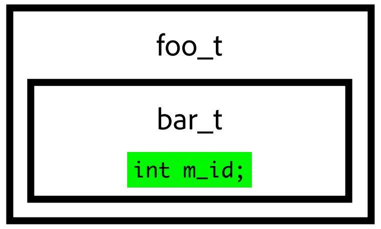
Now, the problem with pointers is that they represent a relationship while also creating another access path to non-parts. So, although you might have independent values at this point, the pointer opens a backdoor for reference semantics to sneak in.

So you need to reframe these extrinsic relationships as something that doesn't grant access.

➡️



Achieving value semantics today | decoupling an object graph



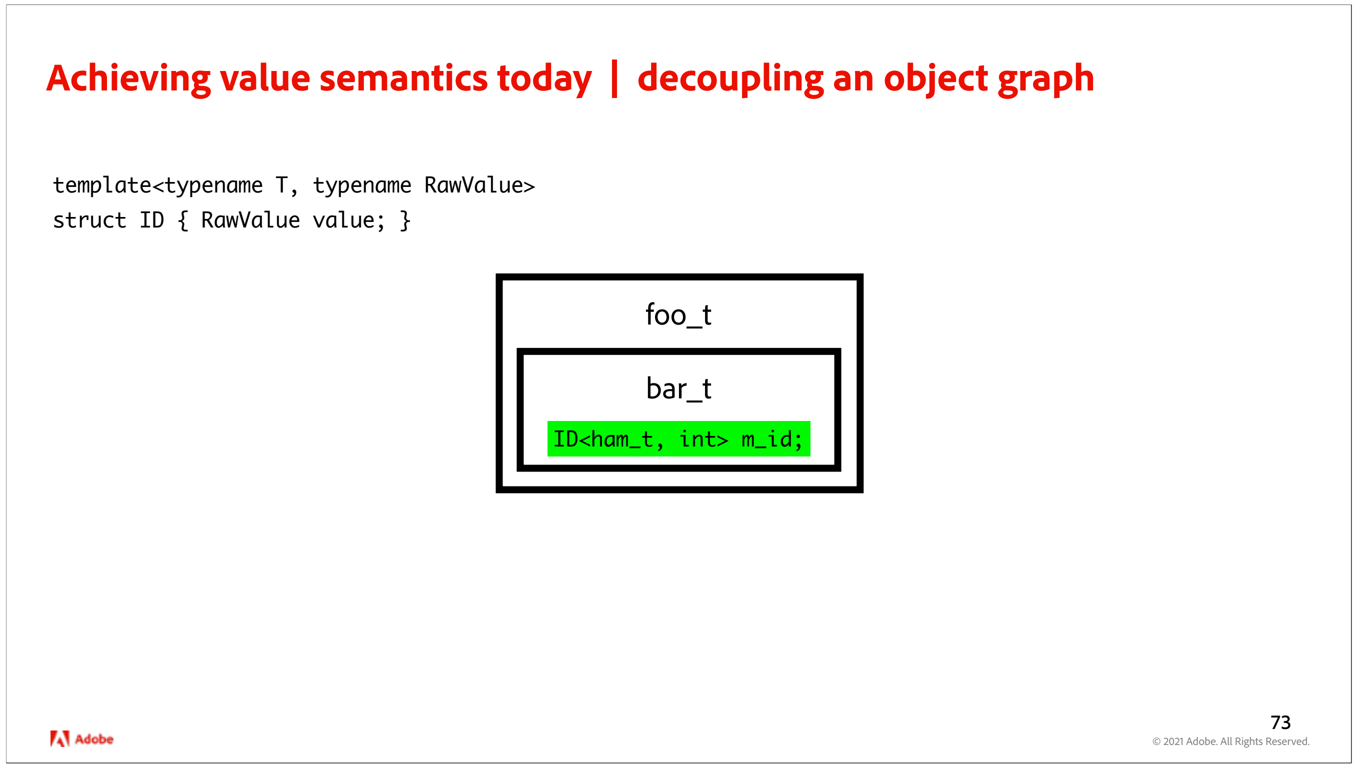
```
int m_id;
```

One simple way to do that is to give every object an identity in the form of an int and use that identity instead of a pointer. That int may, for example, represent an index in an array that contains the referred object.

That's a pretty straightforward approach, but there's one problem that will surely bother some people in this room. There's nothing that prevents us from using that identity to index into the wrong array. In other words, we lost the type information that was carried along with your pointer.

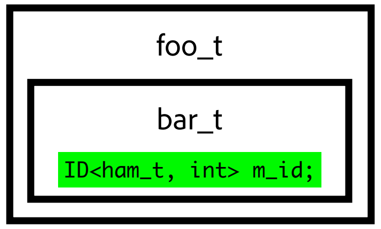
But you can easily get around that problem by tagging your identity with some type information.

➡️



Achieving value semantics today | decoupling an object graph

```
template<typename T, typename RawValue>
struct ID { RawValue value; }
```



```
ID<ham_t, int> m_id;
```

For example, I can just wrap an int in a very simple template to tell the compiler that the identity of a `ham_t` is the not same as that of a `foo_t`.

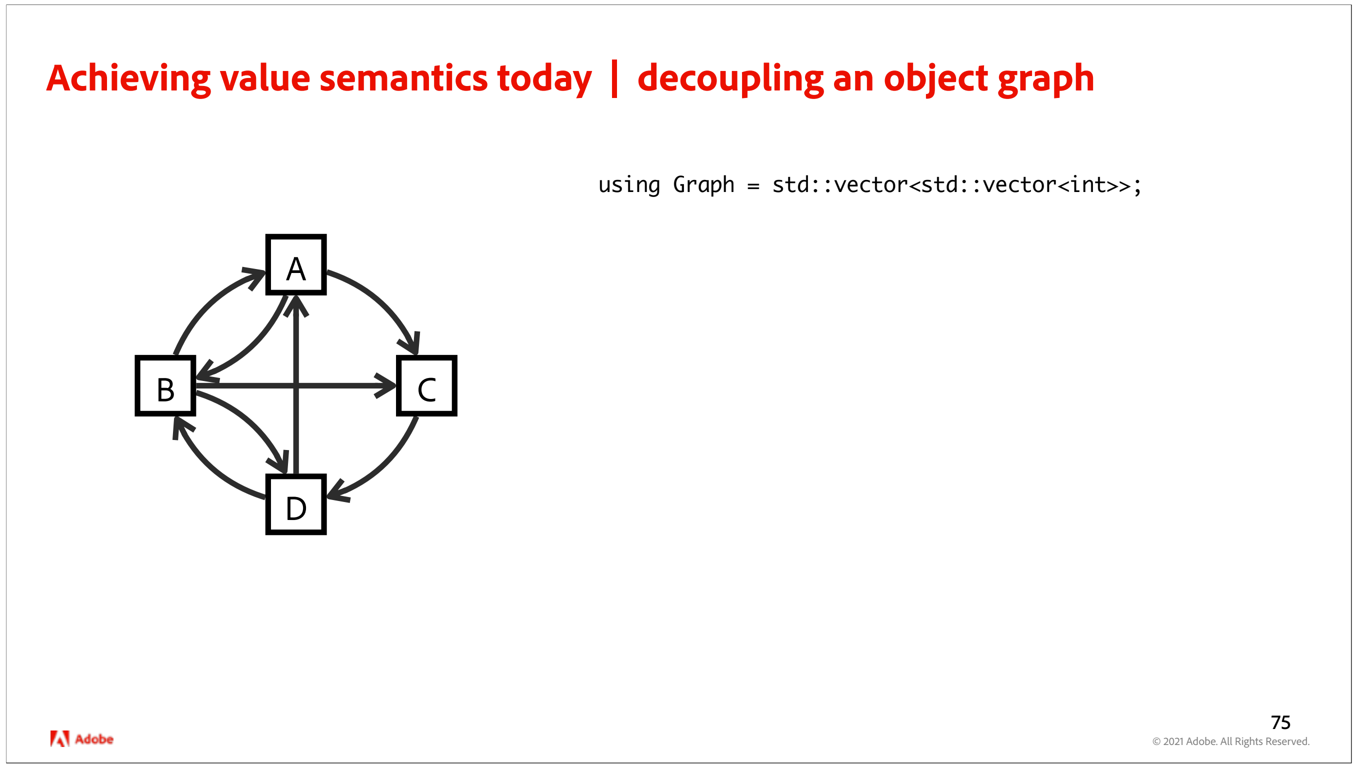
Achieving value semantics today | decoupling an object graph

$$\begin{tikzcd}
 & \boxed{A} \arrow[dl, bend right] \arrow[dr, bend left] & \\
\boxed{B} \arrow[ur, bend left] \arrow[rr] \arrow[dr, bend left] & & \boxed{C} \arrow[dl, bend left] \\
 & \boxed{D} \arrow[uu] \arrow[ul, bend left] &
\end{tikzcd}$$

Sometimes, you do have object graphs in which there are simply no whole/part relationships. How can we deal with those?

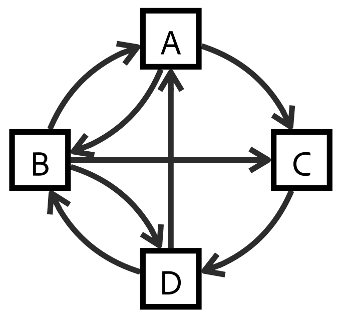
Well, you can use an adjacency list.

➡️



Achieving value semantics today | decoupling an object graph

```
using Graph = std::vector<std::vector<int>>;
```



The simplest implementation is a vector of vectors of integers.

➡️ The outer vector has one entry for every vertex in the graph. Each element of the inner vector represents an edge. For example, ➡️ the edge from A to B is represented like this ➡️ and the one from A to C like that.

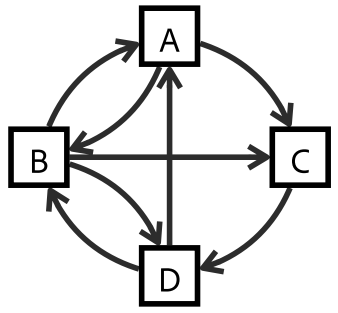
➡️ From there you can probably guess how we represent the rest of the graph.

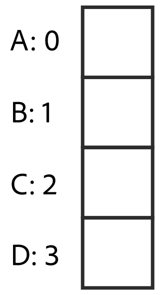
That representation may look foreign to some people. But actually it's a very common way to describe graphs. That's the representation the Boost Graph Library uses for implementing high performance graph algorithms.

➡️

Achieving value semantics today | decoupling an object graph

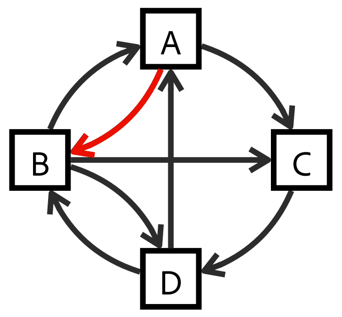
```
using Graph = std::vector<std::vector<int>>;
```

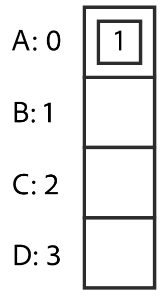




Achieving value semantics today | decoupling an object graph

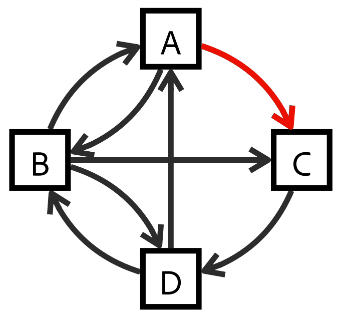
```
using Graph = std::vector<std::vector<int>>;
```

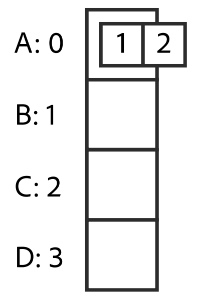




Achieving value semantics today | decoupling an object graph

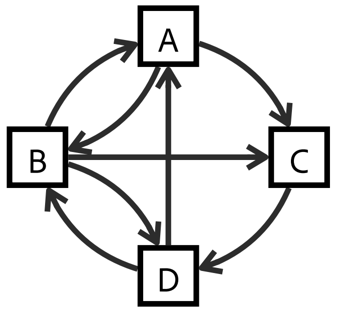
```
using Graph = std::vector<std::vector<int>>;
```

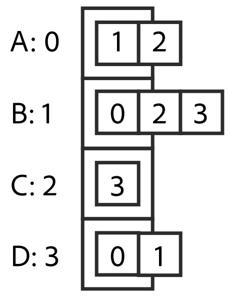


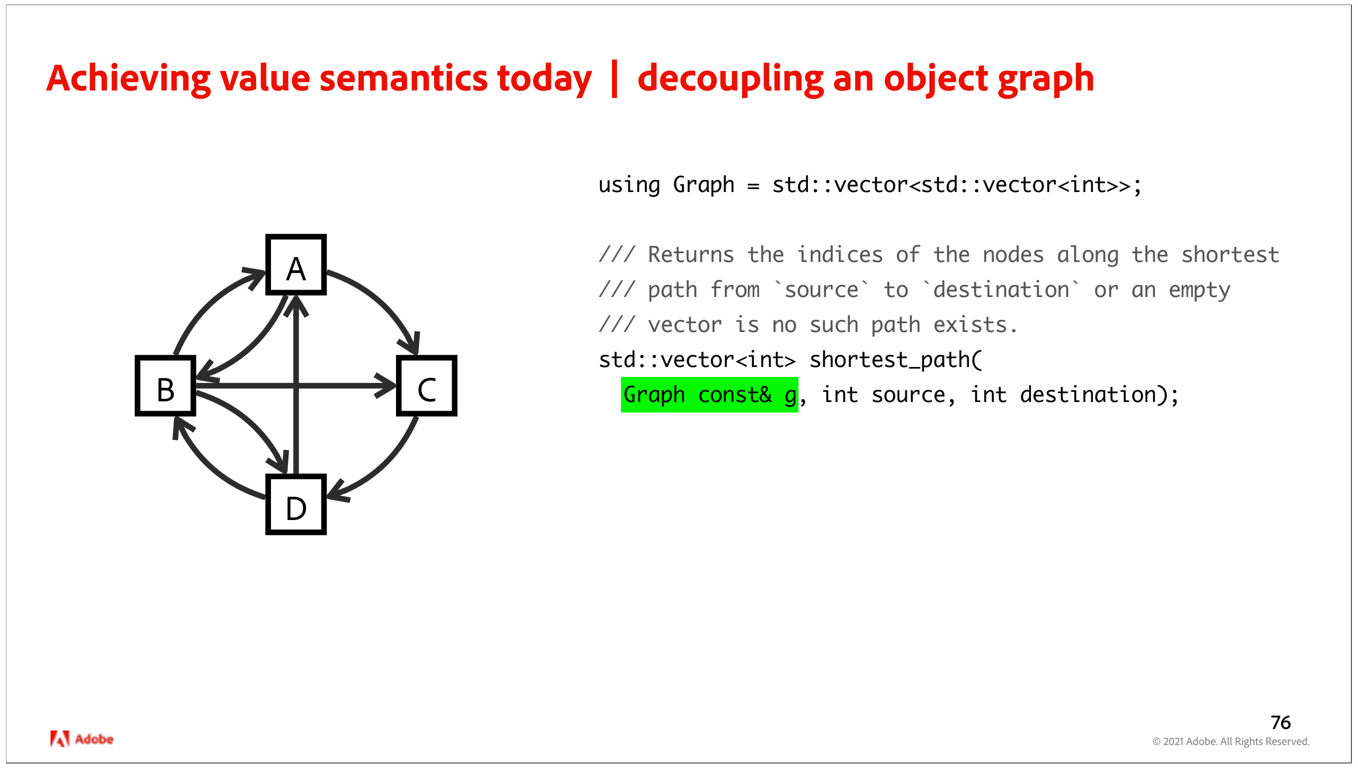


Achieving value semantics today | decoupling an object graph

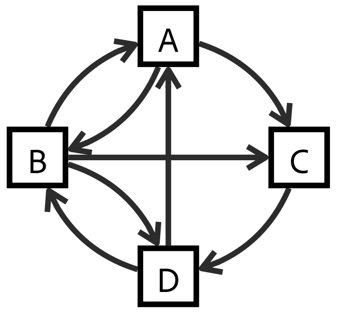
```
using Graph = std::vector<std::vector<int>>;
```







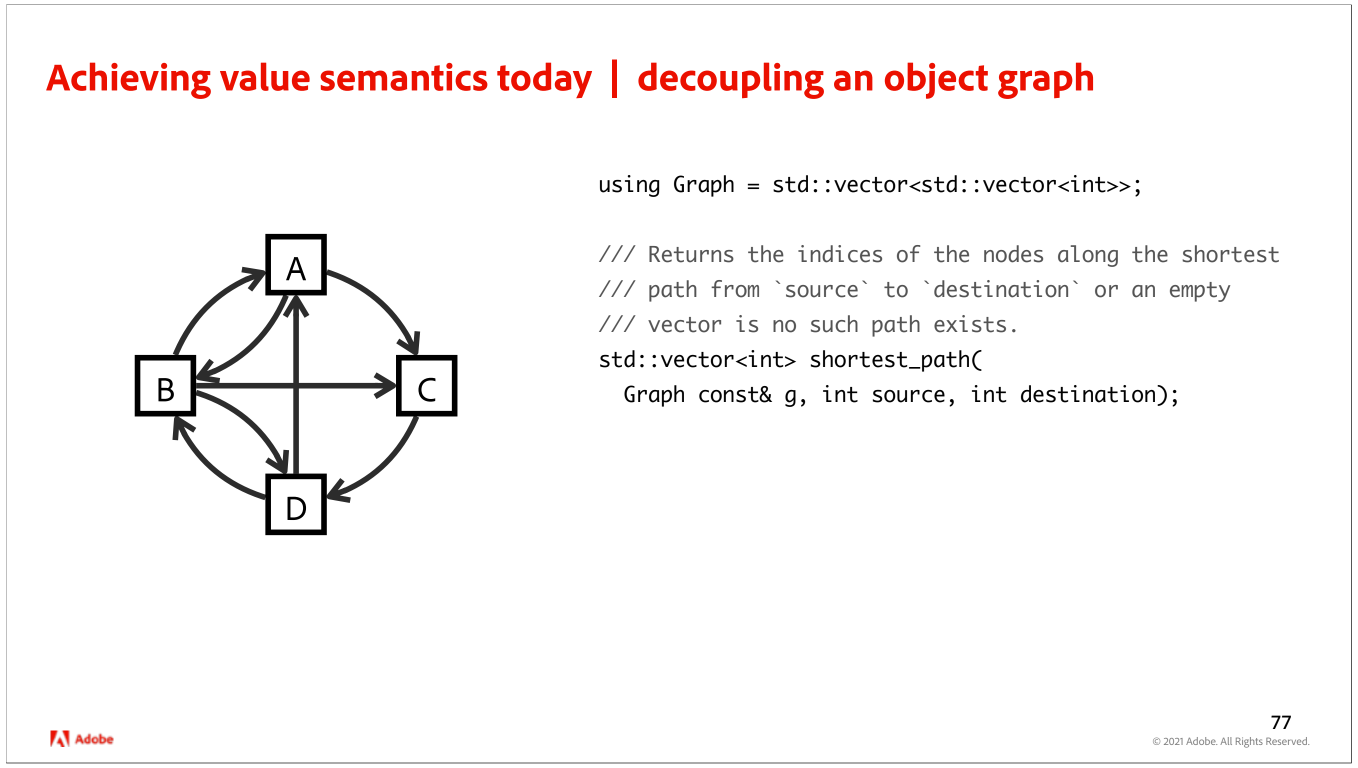
Achieving value semantics today | decoupling an object graph



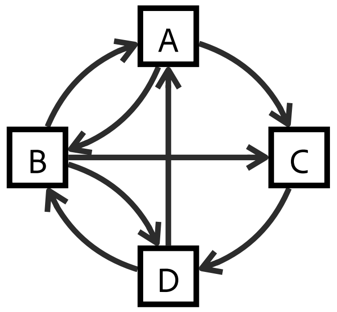
```
using Graph = std::vector<std::vector<int>>;

/// Returns the indices of the nodes along the shortest
/// path from `source` to `destination` or an empty
/// vector is no such path exists.
std::vector<int> shortest_path(
  Graph const& g, int source, int destination);
```

And of course, now we can use our graph just like any value, because our graph has value semantics. So we can reason about its mutation because it is independent, just like an int.

➡️



Achieving value semantics today | decoupling an object graph



```
using Graph = std::vector<std::vector<int>>;

/// Returns the indices of the nodes along the shortest
/// path from `source` to `destination` or an empty
/// vector is no such path exists.
std::vector<int> shortest_path(
  Graph const& g, int source, int destination);
```

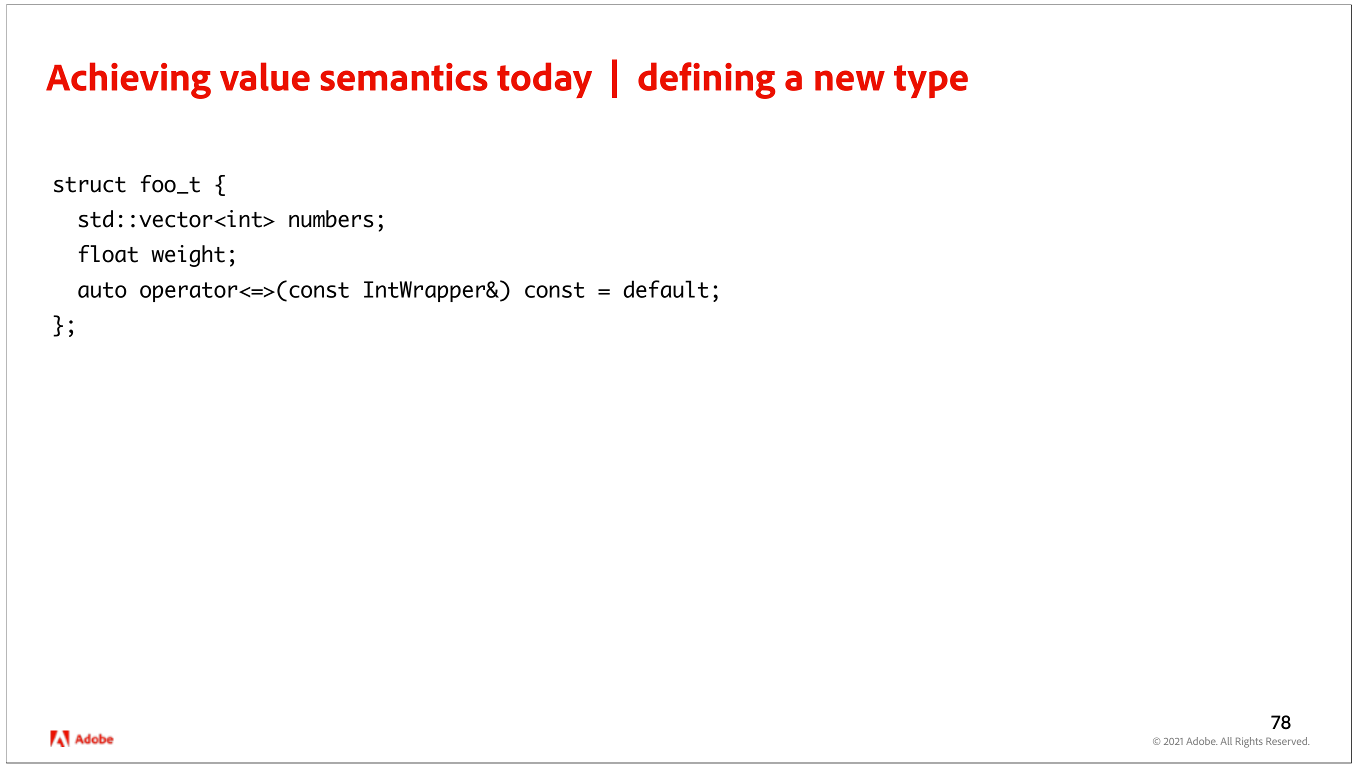
I'm sure some of you may feel uneasy at this point. I know, this is different, it doesn't look what you may be used to do.

And let's be upfront. This discipline has a real cost. You have to give up the convenience of direct access from one object to the parts of another. That does change the way we program things. But in exchange, you and your compiler get to reason about your code to make it correct and faster. That's a pretty good deal in the long run.

Reference semantics is like crack. It's sweet, it's easy, and the first one is free. But then you look at your code and you realize you can't reason about it anymore.

I can give you examples. My colleague Dimitri started writing his compiler with reference semantics. He hit the ground running and then hit all the problems. Eventually we sat down and redesigned the whole thing from the ground up with value semantics. Many of the nasty problems he had just disappeared.

➡️



Achieving value semantics today | defining a new type

```
struct foo_t {
  std::vector<int> numbers;
  float weight;
  auto operator<=>(const IntWrapper&) const = default;
};
```

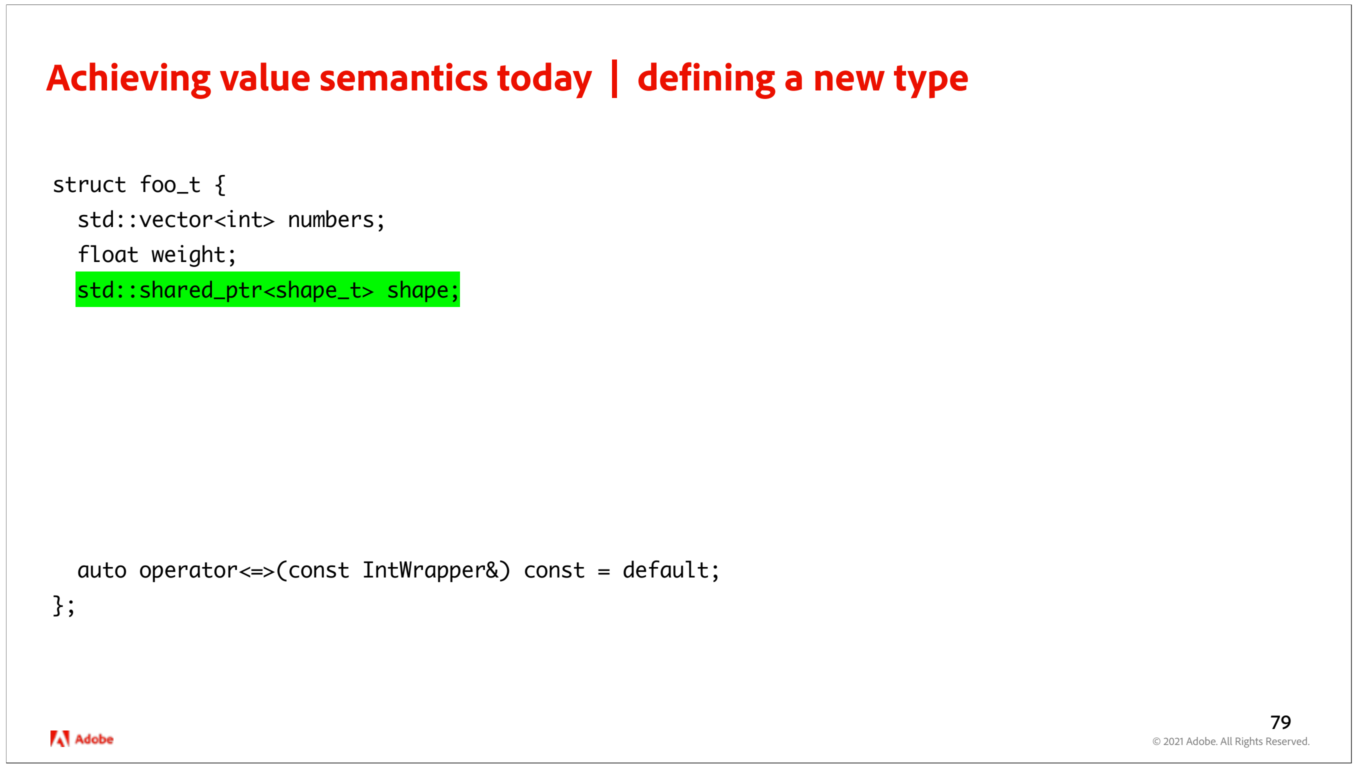
78

© 2021 Adobe. All Rights Reserved.

And what if you need to create a new type?

The easiest way is to just aggregate other value types. Remember that value semantics compose

➡️



Achieving value semantics today | defining a new type

```
struct foo_t {
  std::vector<int> numbers;
  float weight;
  std::shared_ptr<shape_t> shape;




  auto operator<=>(const IntWrapper&) const = default;
};
```

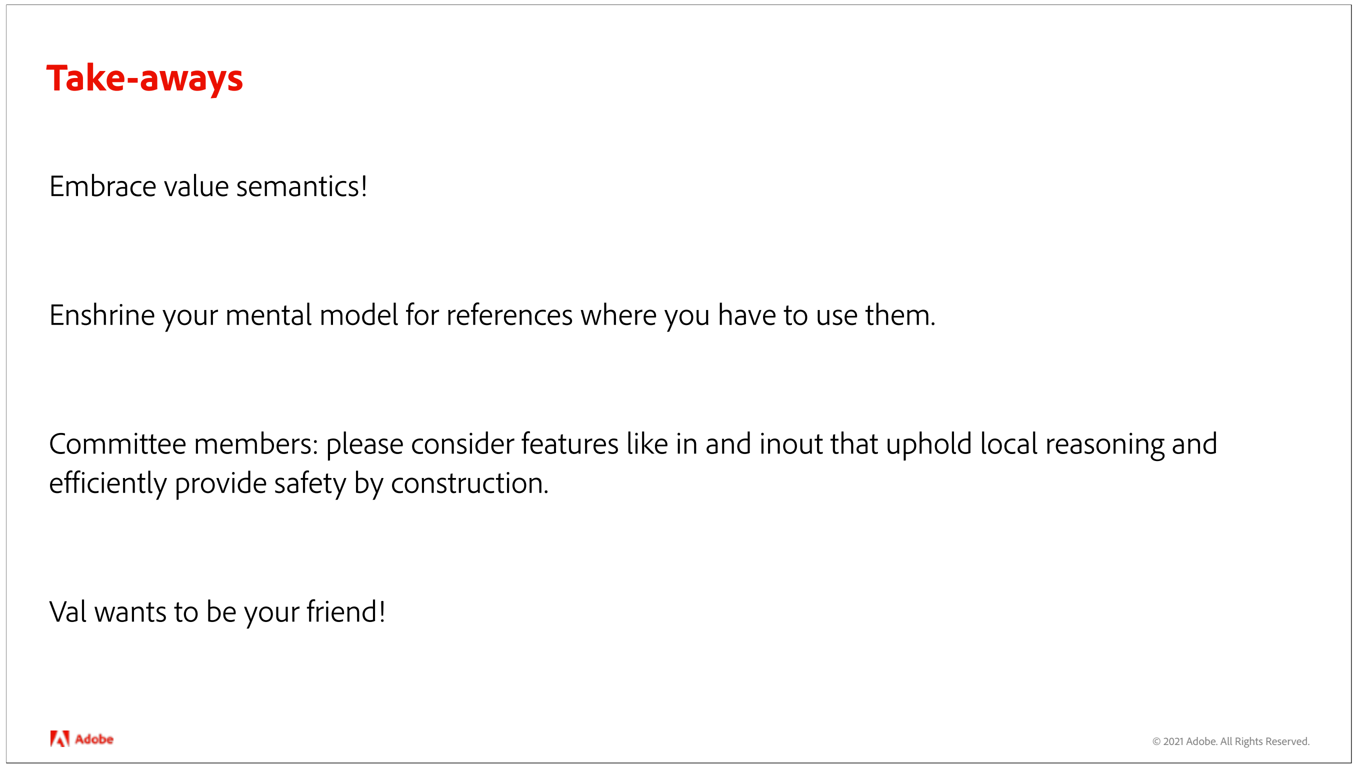
But what if *really* you want to point to data anyway?

➡️ Then you should implement the rule of 5 operations so that the copy constructors and assignment operators preserve value independence.

And if you want to mitigate copying costs, you can implement copy on write.

Achieving value semantics today | defining a new type

```
struct foo_t {
  std::vector<int> numbers;
  float weight;
  std::shared_ptr<shape_t> shape;

  foot_t();
  foo_t(foo_t const&);
  foo_t(foo_t const&&) noexcept;
  foot_t& operator=(foot_t const&);
  foot_t& operator=(foot_t&&) noexcept;

  auto operator<=>(const IntWrapper&) const = default;
};
```


Take-aways

Embrace value semantics!

Enshrine your mental model for references where you have to use them.

Committee members: please consider features like in and inout that uphold local reasoning and efficiently provide safety by construction.

Val wants to be your friend!

Adobe

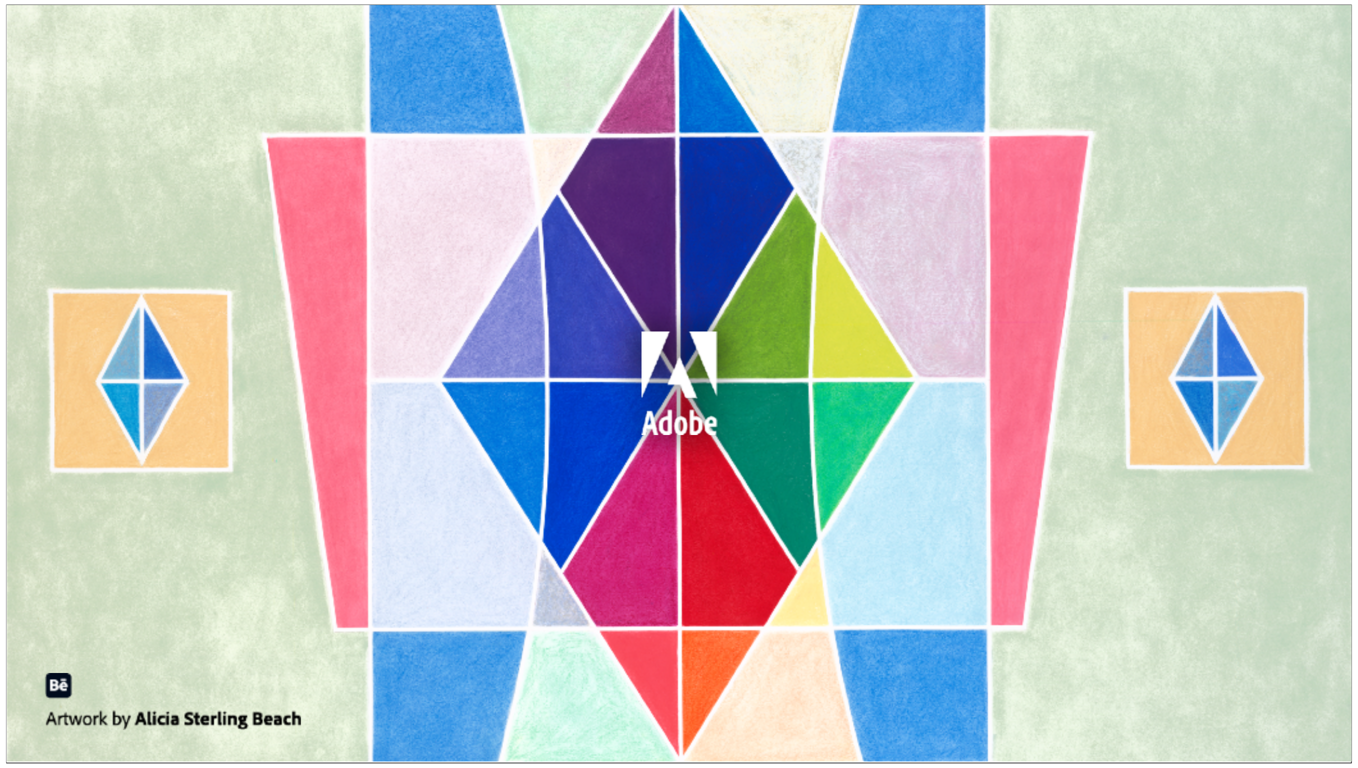
First, embrace value semantics.

Yes, I'm seriously arguing that the most important thing we can do for programming in 2023 is to propagate and uphold the properties of the integer types. The ones we inherited from K&R C in a previous century.

Value semantics is basic. It's boring. It's fundamental. But it is the future of programming.
It is the simplest path to local reasoning, one that doesn't require countless annotations and isn't alien to the C++ type system
It is the only known path to local reasoning that scales.
It is the path to safe and efficient exploitation of multiple cores and heterogeneous compute.

It's the only path to correct mutation
Done right, it's faster.
What more could you want, really?

Value semantics does *lead* somewhere exciting, if you push it to its limits, though. I've been collaborating on a language project called Val that is focused on value semantics, generic programming, and interoperability with C++. Go watch Dimitri's talk to learn more.


Adobe
Artwork by Alicia Sterling Beach