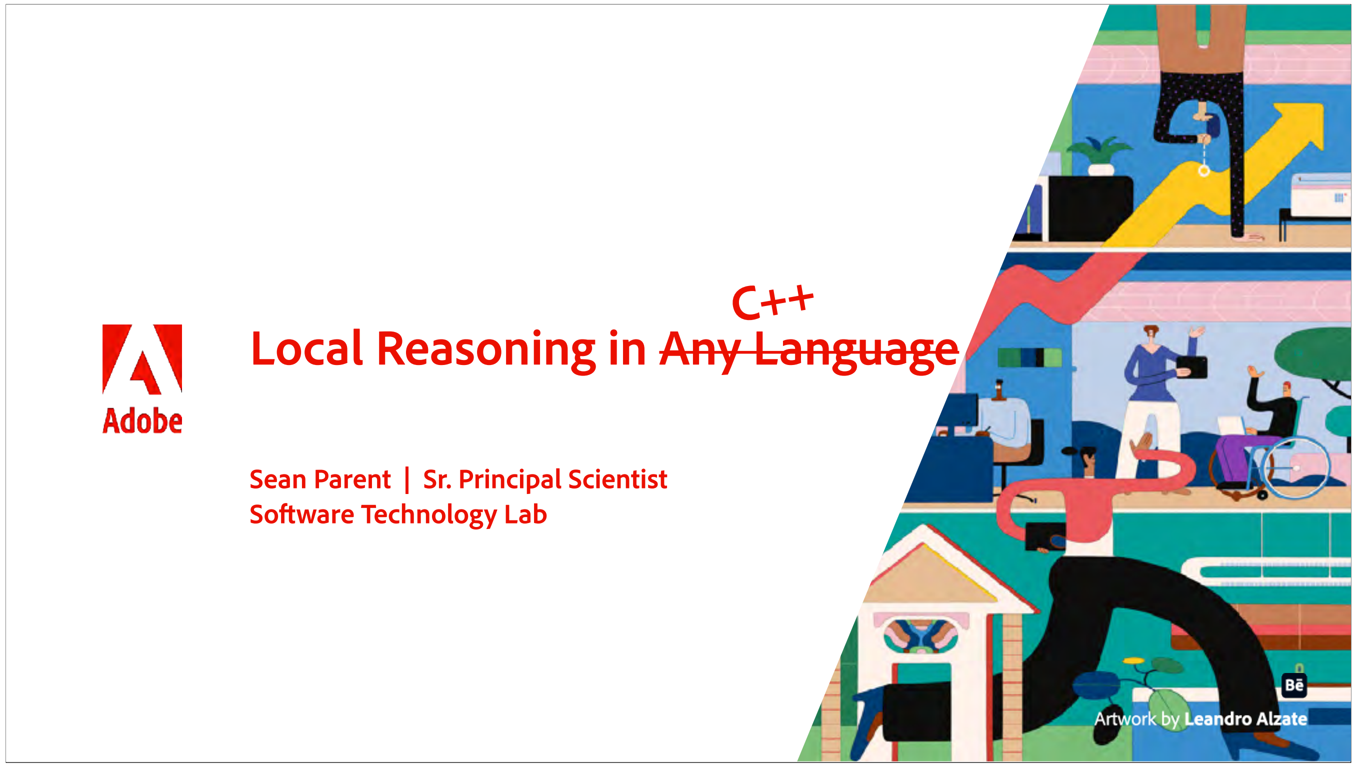


Adobe

Local Reasoning in ~~Any Language~~ C++

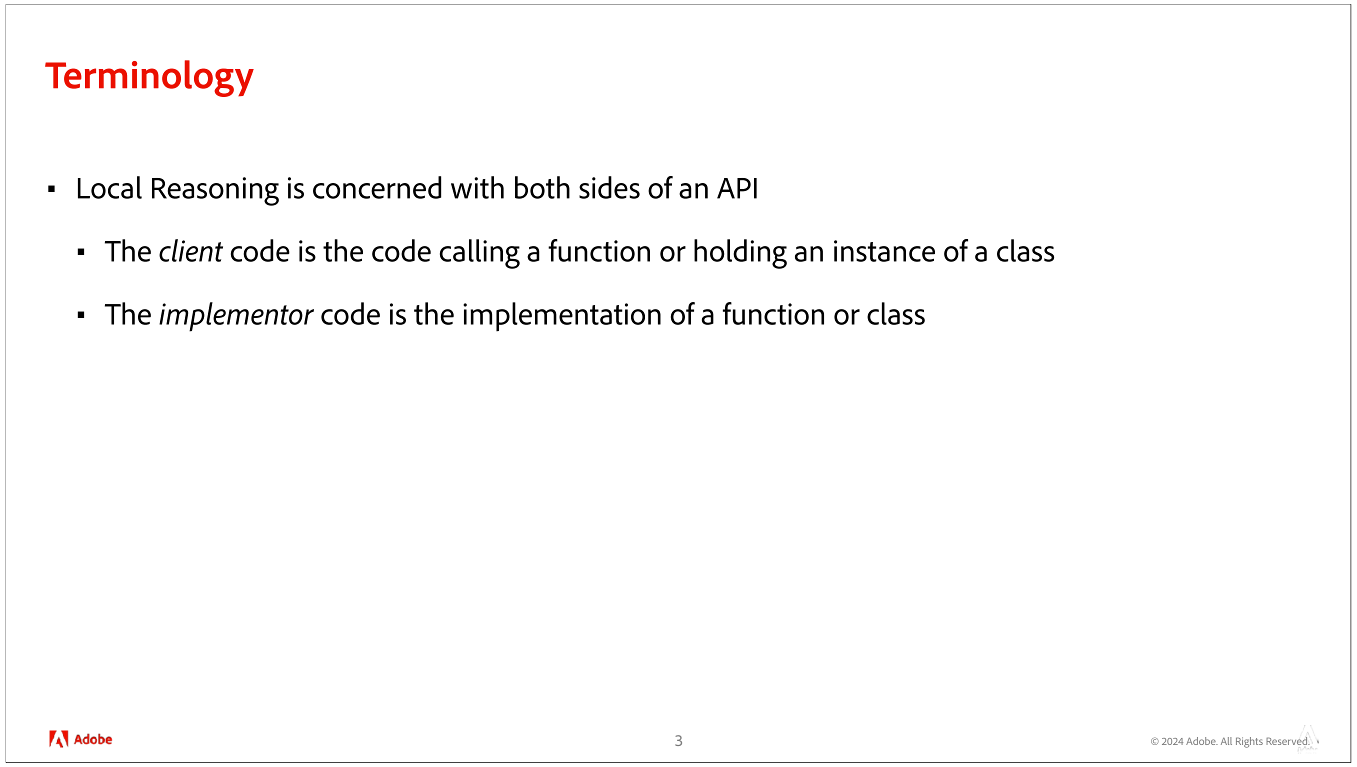
Sean Parent | Sr. Principal Scientist
Software Technology Lab

Artwork by Leandro Alzate

This is the first draft of this talk. The original idea was to do "Local Reasoning in Any Language" to document how I think about code, regardless of the language I'm programming in. The talk gets mired in the C++ details, so doing a set of languages seemed too much. Except for the details, the rules in this talk apply to all languages. I present here how I map these ideas into C++; it isn't the only mapping for C++, and if you program in a different language, figure out a set of conventions to map the ideas into that language.

Local Reasoning

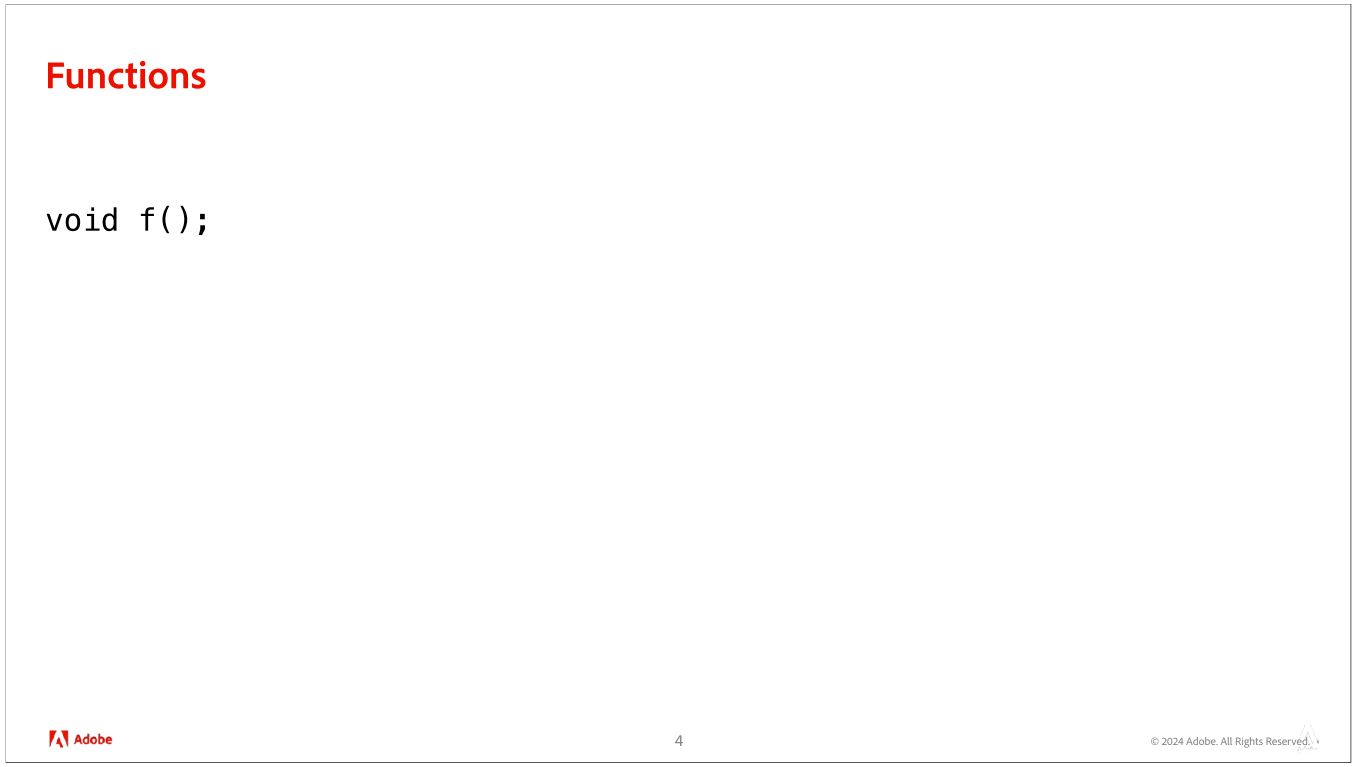
- *Local Reasoning* is the ability to reason about a defined unit of code and verify its correctness without understanding all the contexts in which it is used or the implementations upon which it relies.
- The two units of code this talk is concerned with are:
  - Functions
  - Classes



Terminology

- Local Reasoning is concerned with both sides of an API
  - The *client* code is the code calling a function or holding an instance of a class
  - The *implementor* code is the implementation of a function or class

I'll sometime use _caller_ and _callee_ when discussing functions, but client and implementor generalize to classes.

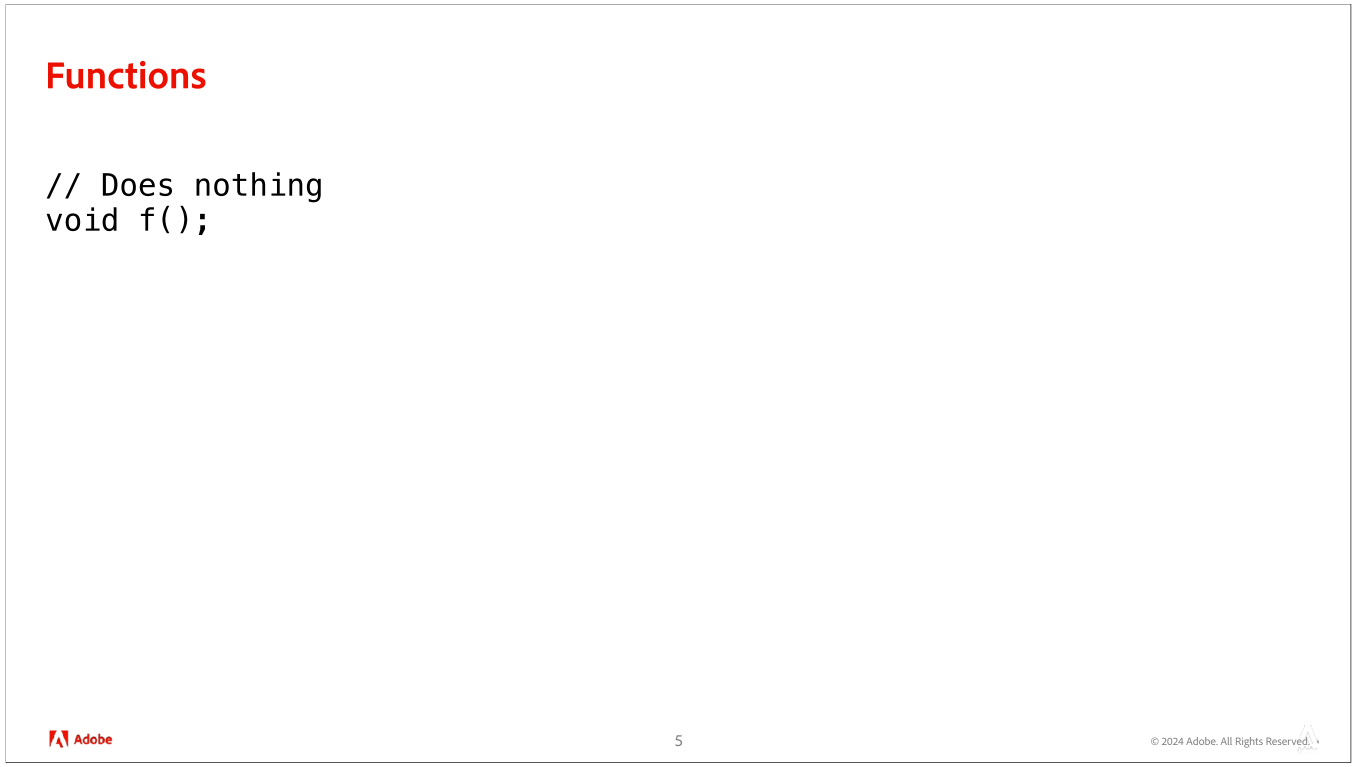


Functions

```
void f();
```

Adobe

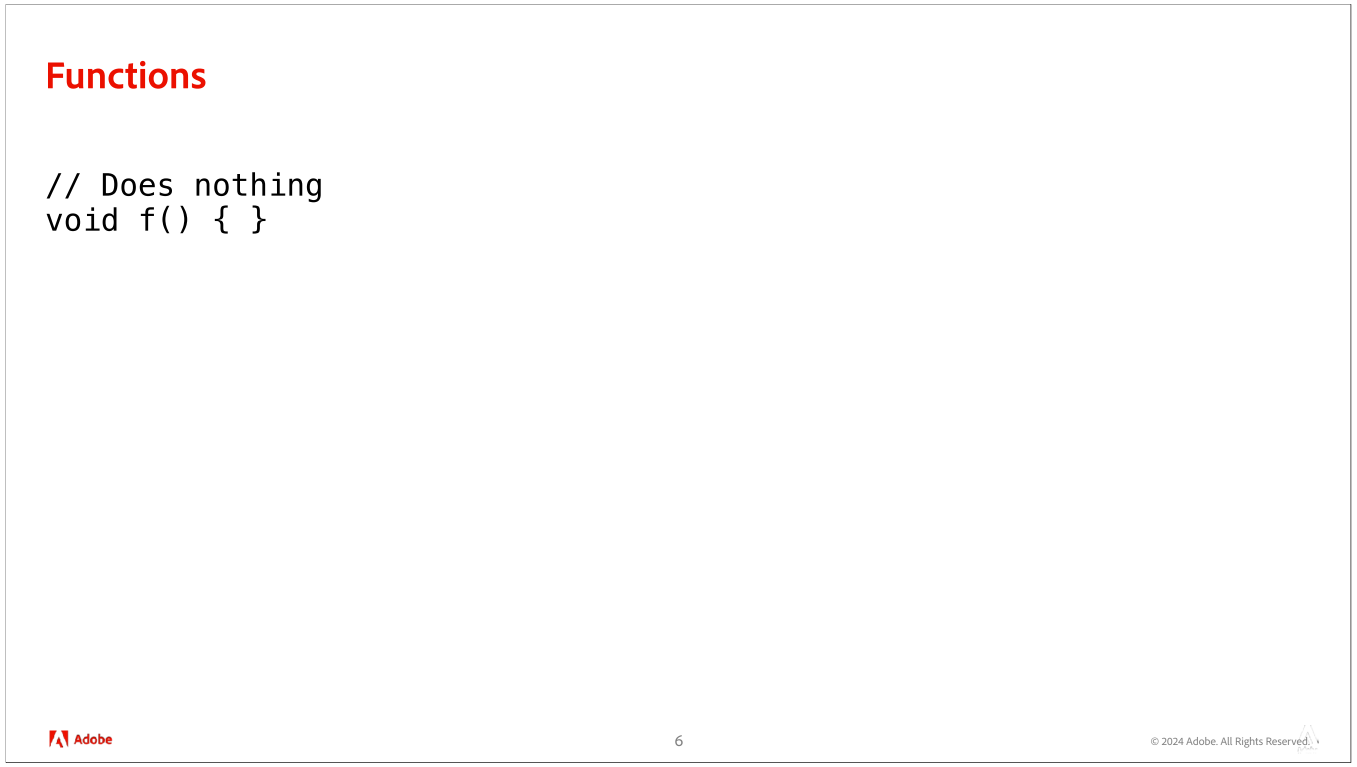
Let's start with a simple function signature. <click>
Either this function does nothing, or whatever it does is entirely through side effects. Either way, we should document it. <click>
Now we can implement `f` <click>



Functions

```
// Does nothing
void f();
```

Let's start with a simple function signature. <click>
Either this function does nothing, or whatever it does is entirely through side effects. Either way, we should document it. <click>
Now we can implement `f` <click>



Functions

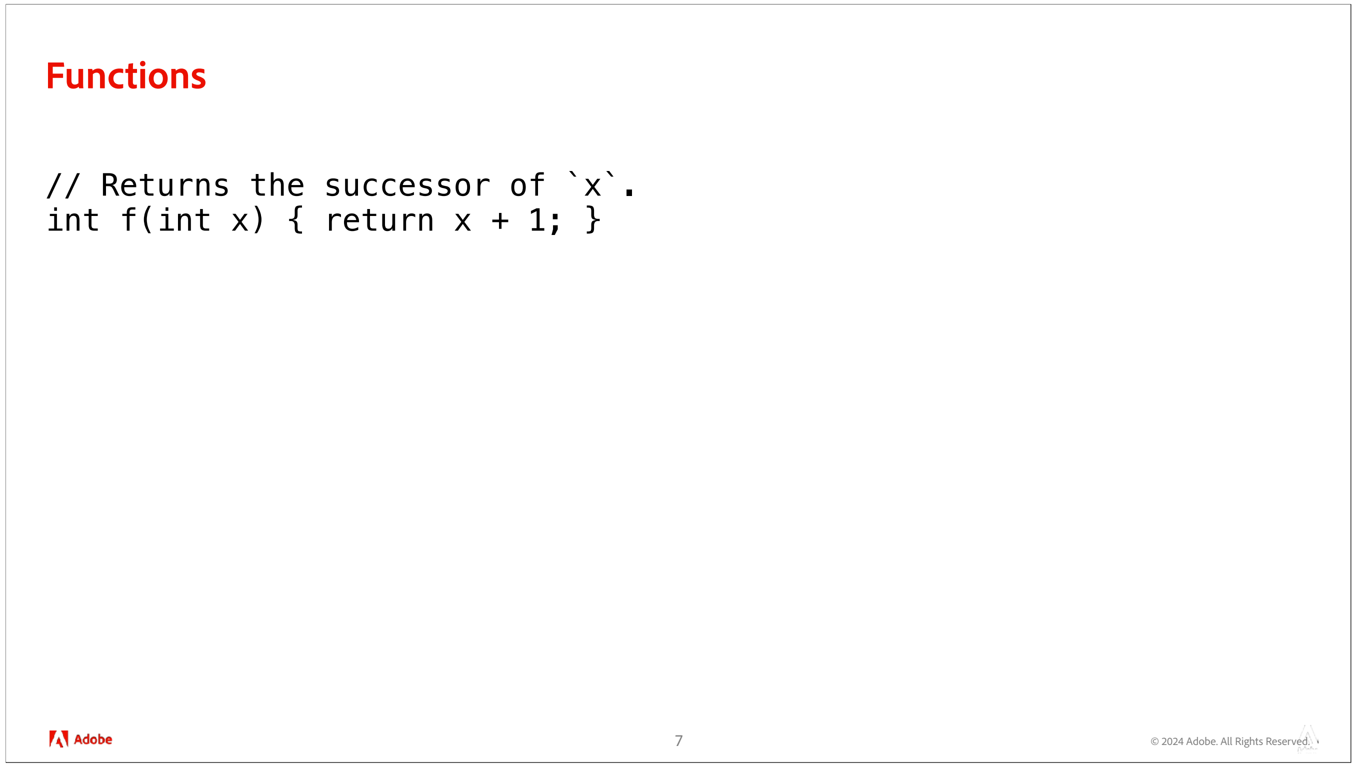
```
// Does nothing
void f() { }
```

Adobe

I hope everyone is convinced that `f()` is implemented correctly. A requirement for local reasoning is a specification, a contract.

Suppose a piece of code has a contract, and everything it invokes also has one. In that case, we can read the implementation and prove (or disprove) that the function body is correct and fulfills the contract. But this isn't another talk about contracts; instead, it is about general principles for constructing code that is _simple_ to reason about and that we can prove correct.

Let's make our function a little more complicated<click>

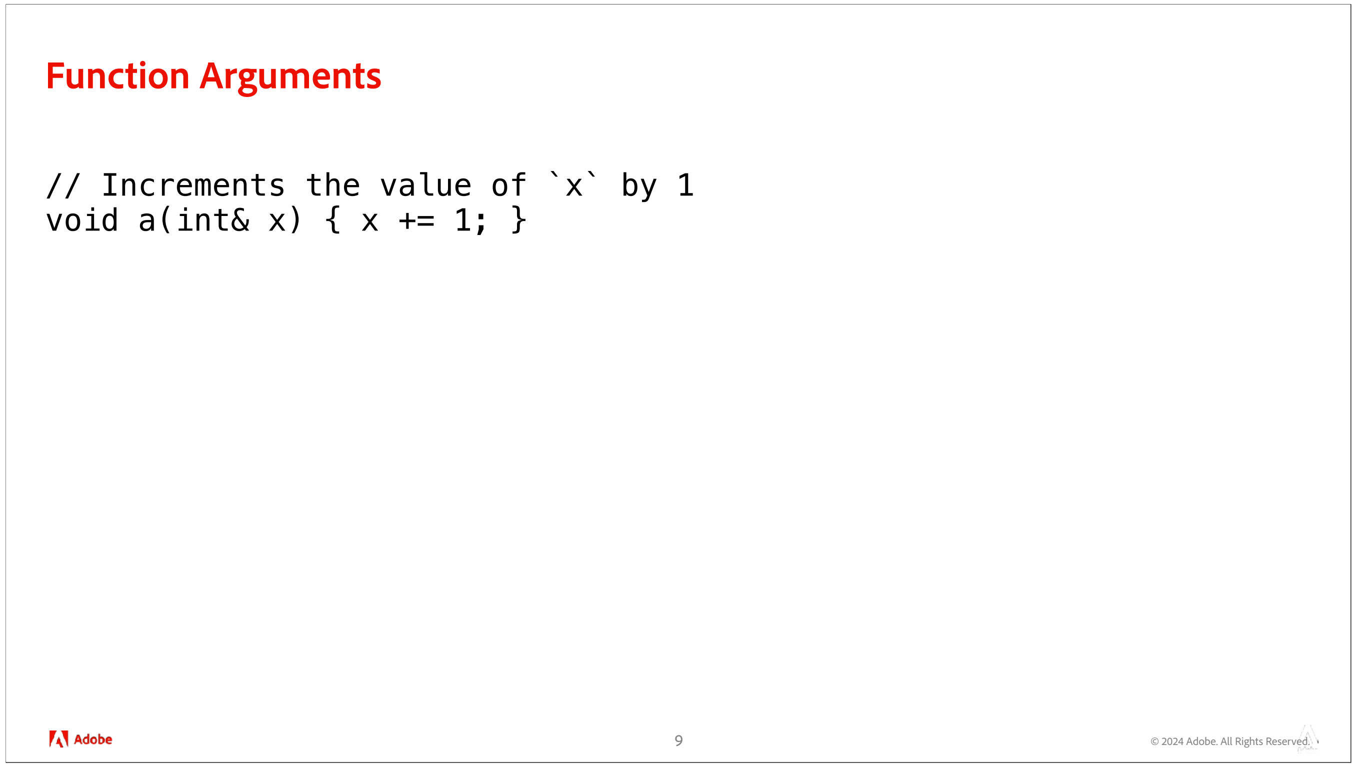


Functions

```
// Returns the successor of `x`.
int f(int x) { return x + 1; }
```

Still very simple, this code is easy to understand at a glance. It doesn't have a great name—we'll get to that—but it does what the specification says. Let's add a little more complexity <click>

Function Arguments



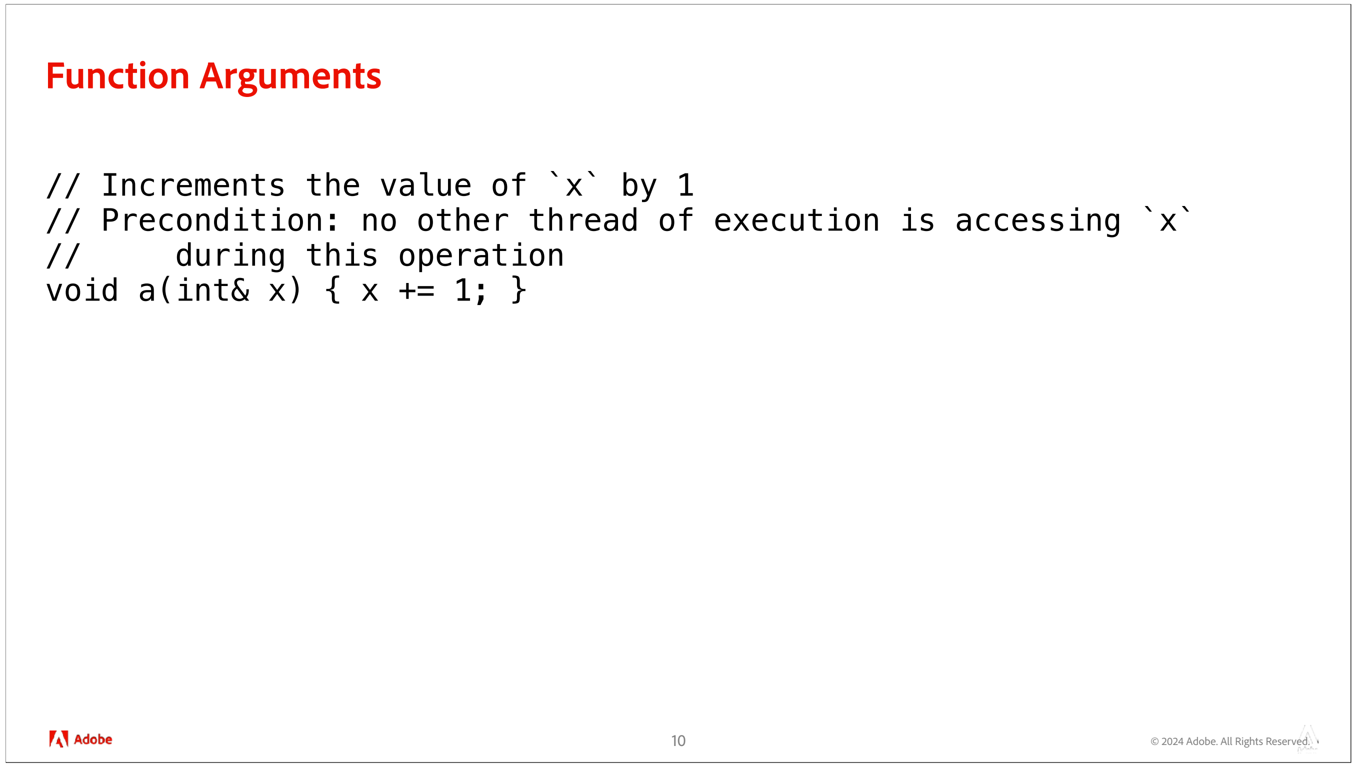
Function Arguments

```
// Increments the value of `x` by 1
void a(int& x) { x += 1; }
```

Adobe

This function is still simple; is it correct? We introduced a precondition. What is it?

What if another thread is readying `x` when we update it? That would be a data race. There is an implicit precondition here <click>

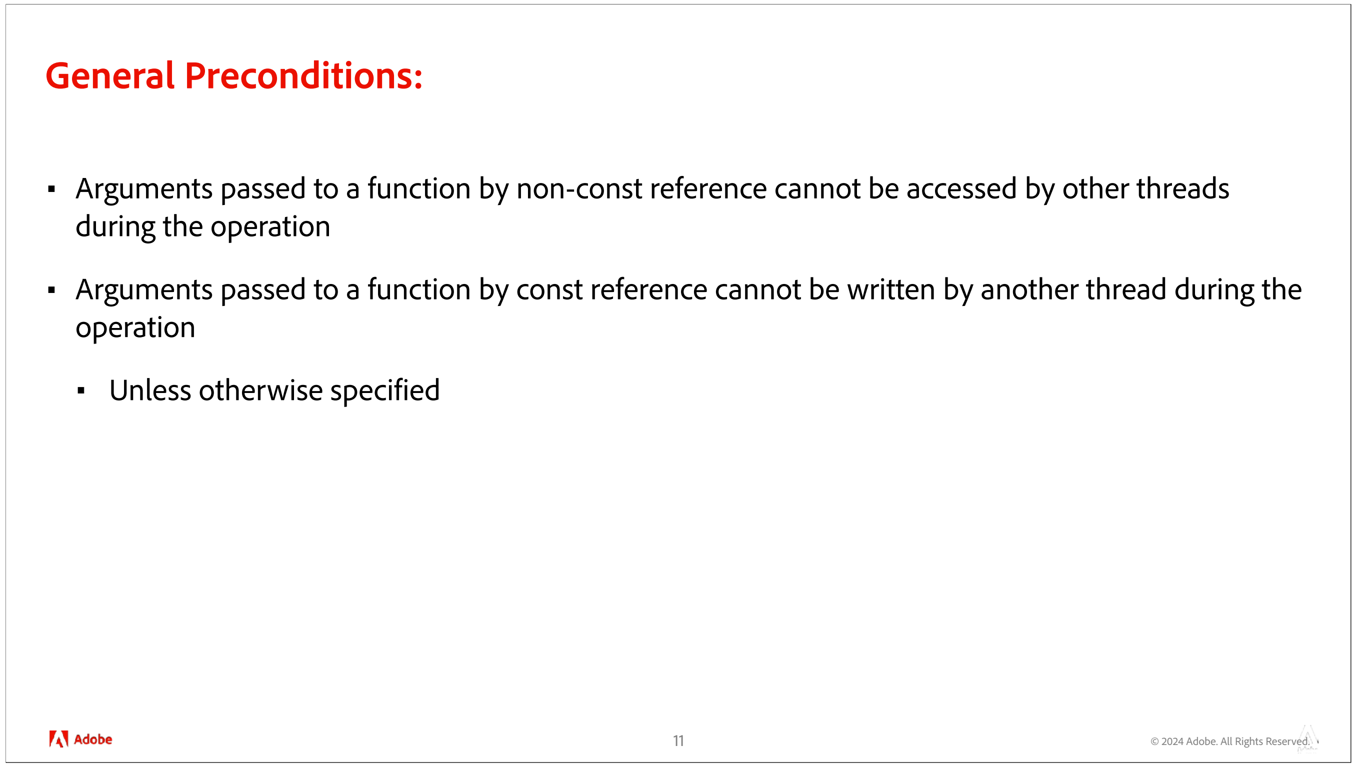


Function Arguments

```
// Increments the value of `x` by 1
// Precondition: no other thread of execution is accessing `x`
//     during this operation
void a(int& x) { x += 1; }
```

This precondition cannot be tested or verified by `a()`. The client must ensure it. By introducing indirection (passing the argument by reference), we raise the prospect of _aliasing_ in the interface, having more than one way to access an object. The rest of this talk is about techniques to control aliasing and confine the effect of an operation so it can be reasoned about locally.

We certainly don't want to write preconditions like this with every function. So, instead, we're going to develop a set of general preconditions that must be upheld for all operations unless otherwise specified.

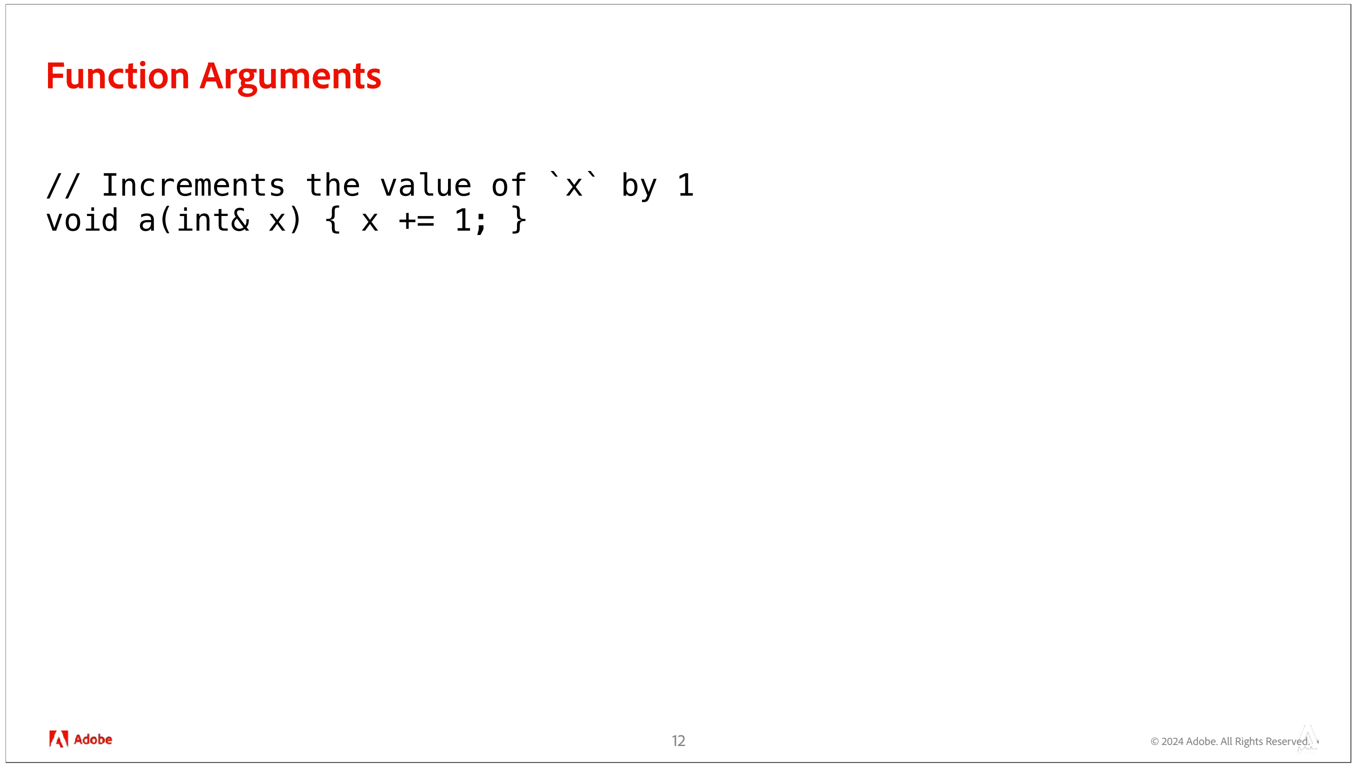


General Preconditions:

- Arguments passed to a function by non-const reference cannot be accessed by other threads during the operation
- Arguments passed to a function by const reference cannot be written by another thread during the operation
    - Unless otherwise specified

Adobe

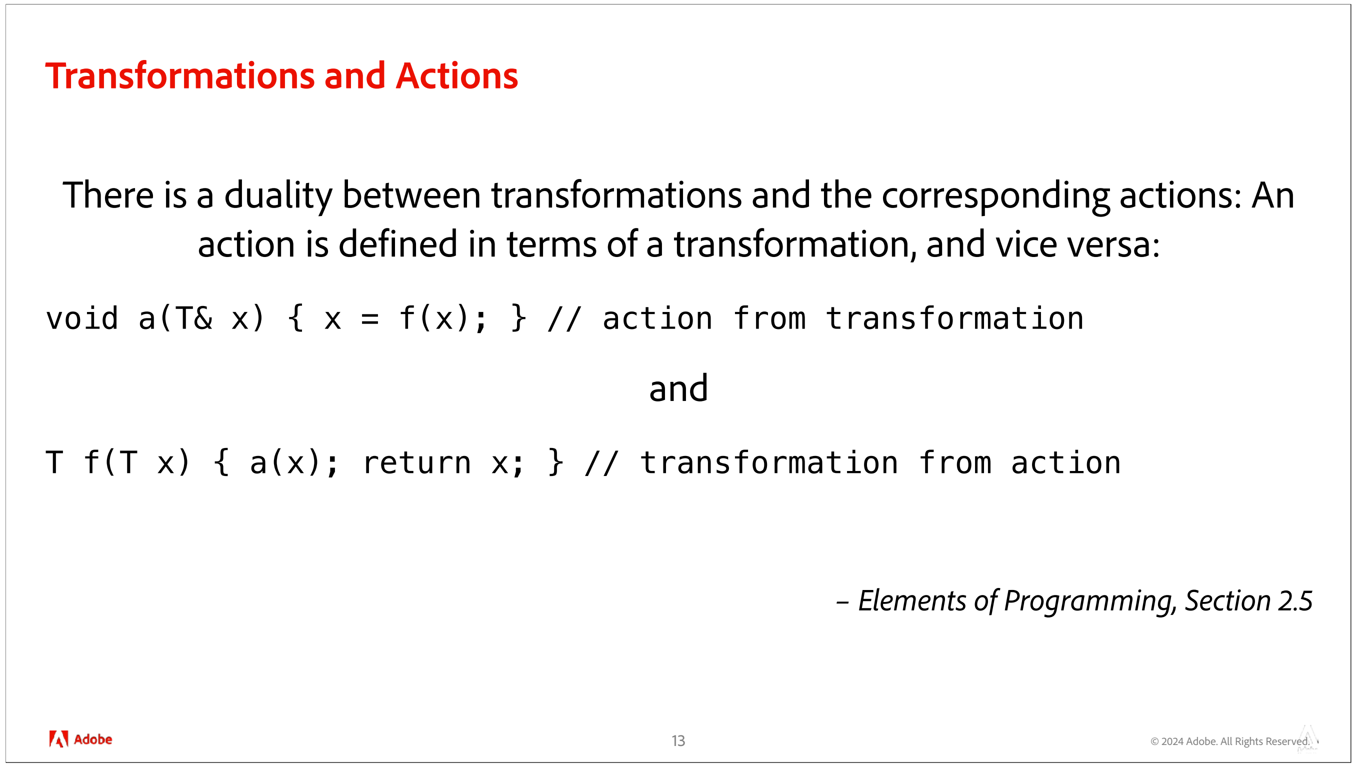
And now we can remove our precondition.



Function Arguments

```
// Increments the value of `x` by 1
void a(int& x) { x += 1; }
```

We don't normally pass an `int` by reference; we pass an object by reference as an optimization to avoid unnecessary copies. But for types where the cost of taking the reference is as much as passing the value, we pass the value. By convention, arithmetic types and pointers are passed by value. In generic code, iterators and invocable (function objects) are passed by value because they are likely small and trivial.



Transformations and Actions

There is a duality between transformations and the corresponding actions: An action is defined in terms of a transformation, and vice versa:

```
void a(T& x) { x = f(x); } // action from transformation
```

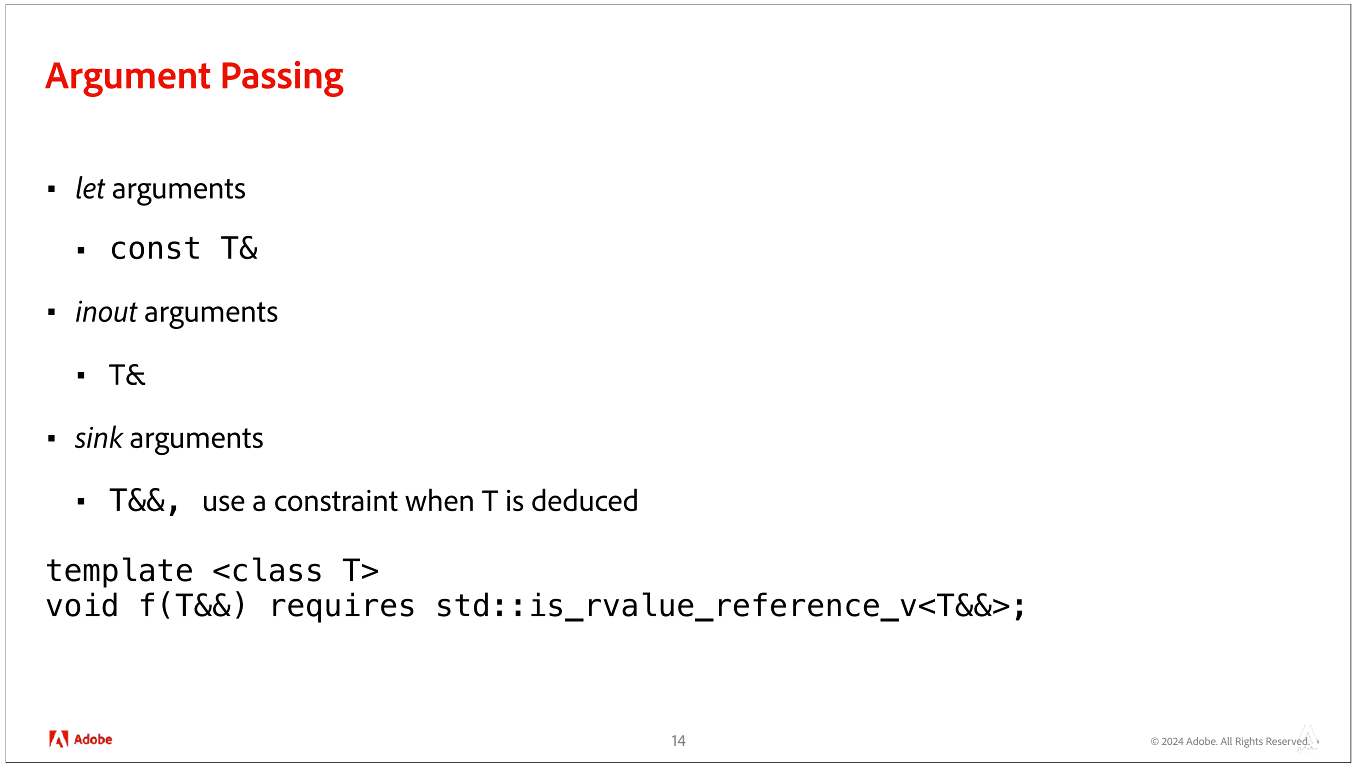
and

```
T f(T x) { a(x); return x; } // transformation from action
```

– Elements of Programming, Section 2.5

For a given instance, either an action or transformation may have a more efficient implementation. All other things being equal prefer transformations.

The transformation form here is taking the argument by value, but we need to say a little more about passing arguments <click>



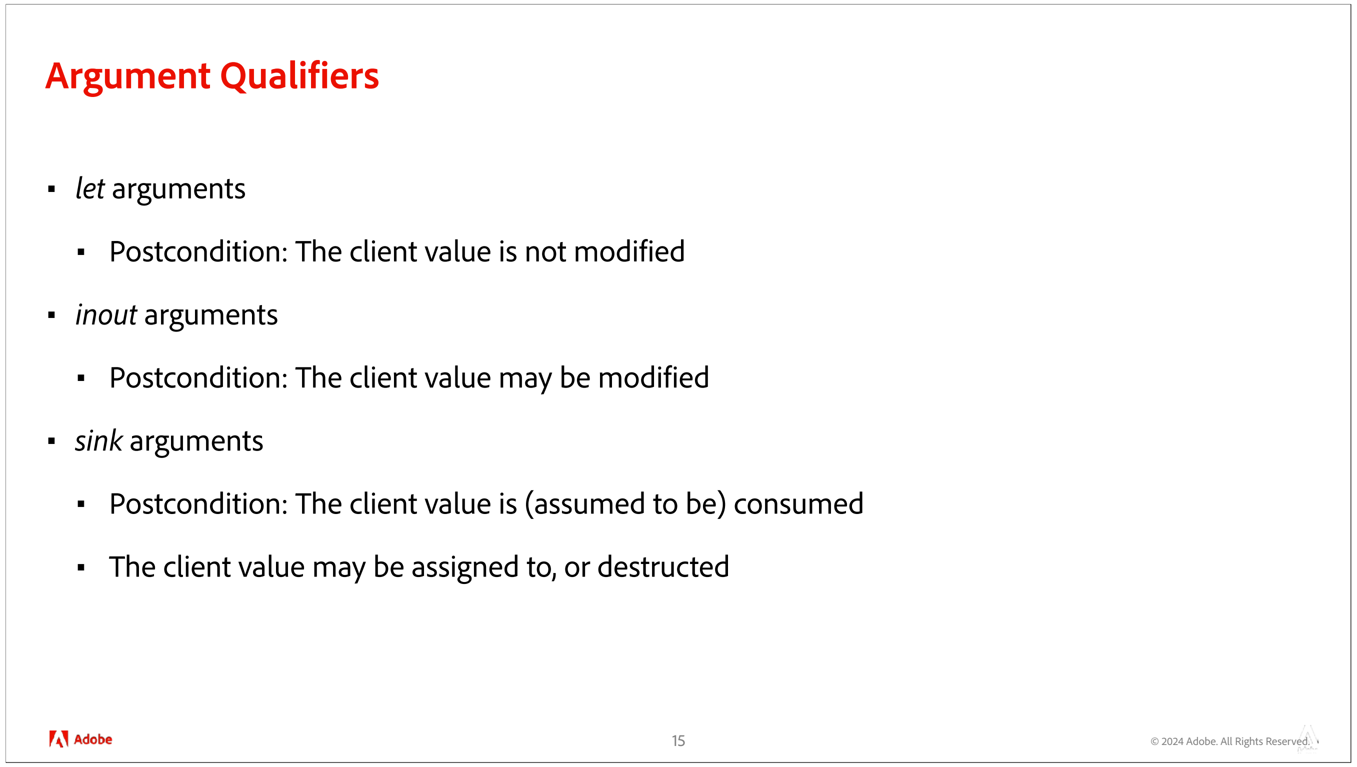
Argument Passing

- *let* arguments
  - `const T&`
- *inout* arguments
  - `T&`
- *sink* arguments
  - `T&&,` use a constraint when T is deduced

```
template <class T>
void f(T&&) requires std::is_rvalue_reference_v<T&&>;
```

We want the sink to be a non_const rvalue reference, I leave it as an exercise to write is_sink_v constraint. Unfortunately, I don't see a way to do it as a concept where you could say `auto sink a`

```
template <class T>
inline constexpr bool is_sink_v{std::is_const_v<std::remove_reference_t<T>> && std::is_rvalue_reference_v<T>};
```



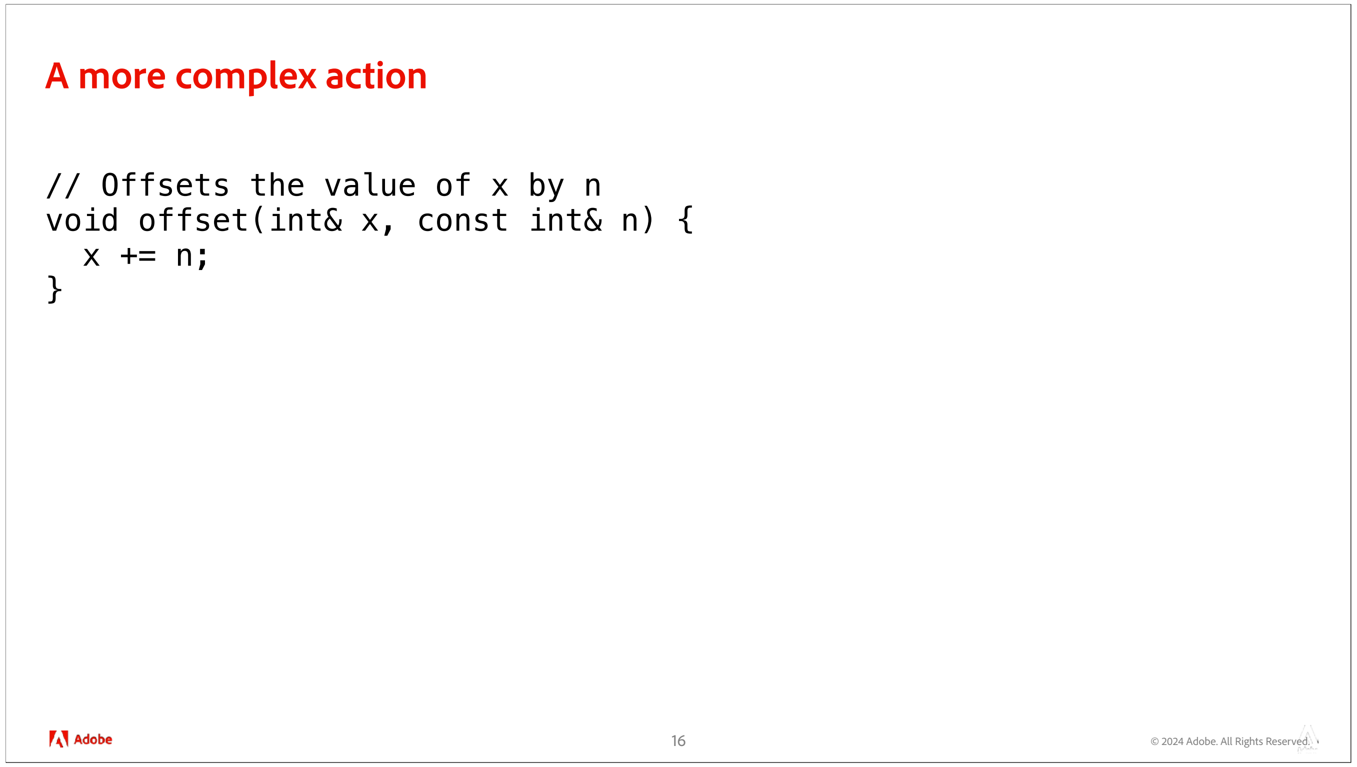
Argument Qualifiers

- *let* arguments
  - Postcondition: The client value is not modified
- *inout* arguments
  - Postcondition: The client value may be modified
- *sink* arguments
  - Postcondition: The client value is (assumed to be) consumed
  - The client value may be assigned to, or destructed

We want each of these to behave like the corresponding transformation form was used. We already found we cannot alias the value across threads. Are there other preconditions?

sink arguments are used when the argument is escaped - either stored or returned, possibly with modification.

Pass by value is a let argument from the caller side, and consumable (sink) by the implementor. For small (<= sizeof(void*)) basic types (move and copy are equivalent) pass by value is used.



A more complex action

```
// Offsets the value of x by n
void offset(int& x, const int& n) {
  x += n;
}
```

Adobe

Will this print `4`, or `2`, or something else? We can see from the implementation that the answer is `4`. This is breaking the client contract that the second argument is not modified. The postconditions conflict - a contradiction. But maybe this is what we "expected." But what if offset was implemented this way <click>

A more complex action

```
// Offsets the value of x by n
void offset(int& x, const int& n) {
  x += n;
}
```

- What if this is called as:

```
int x{2};
offset(x, x);

println("{}", x);
```

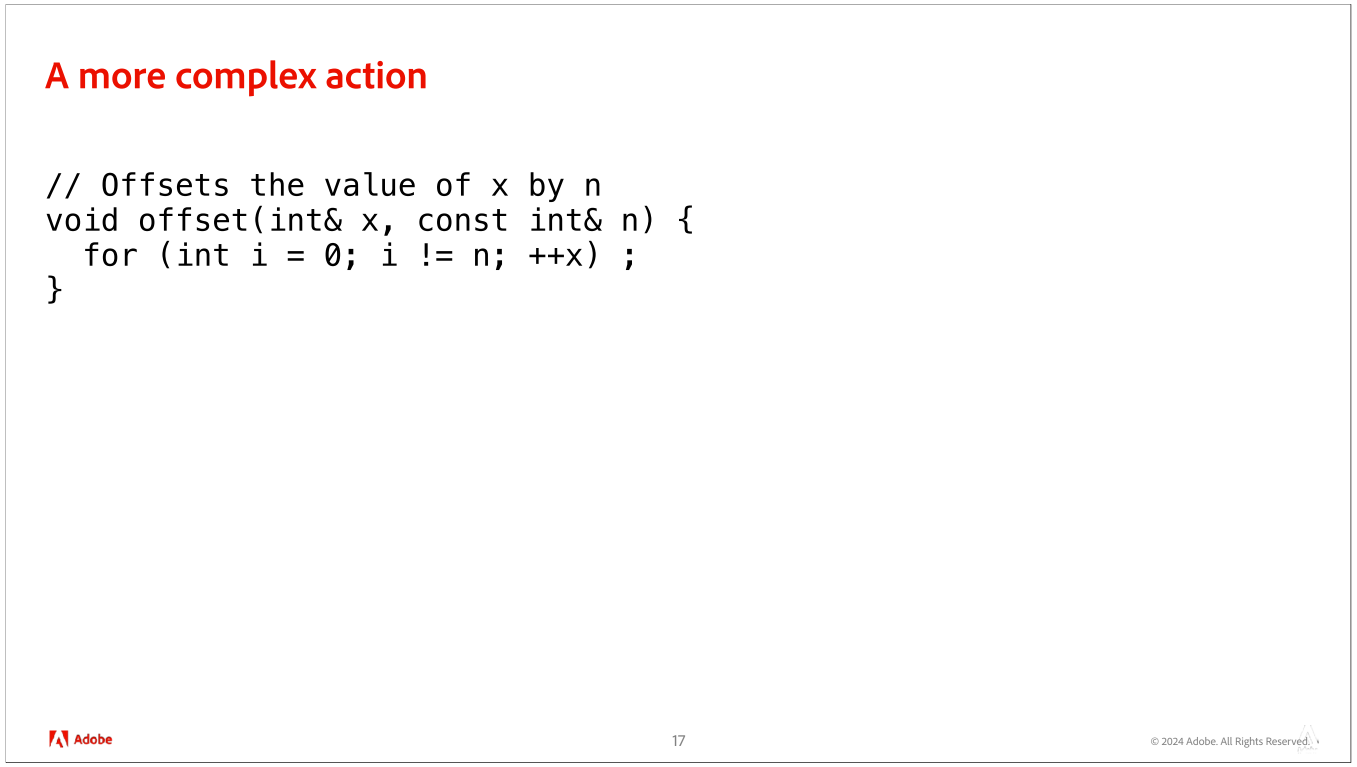
A more complex action

```
// Offsets the value of x by n
void offset(int& x, const int& n) {
  x += n;
}
```

- What if this is called as:

```
int x{2};
offset(x, x);

println("{}", x);
```

4



A more complex action

```
// Offsets the value of x by n
void offset(int& x, const int& n) {
  for (int i = 0; i != n; ++x) ;
}
```

the print statement is never reached. Because of the aliasing between arguments, where one is under mutation, the implementation cannot satisfy either postcondition.

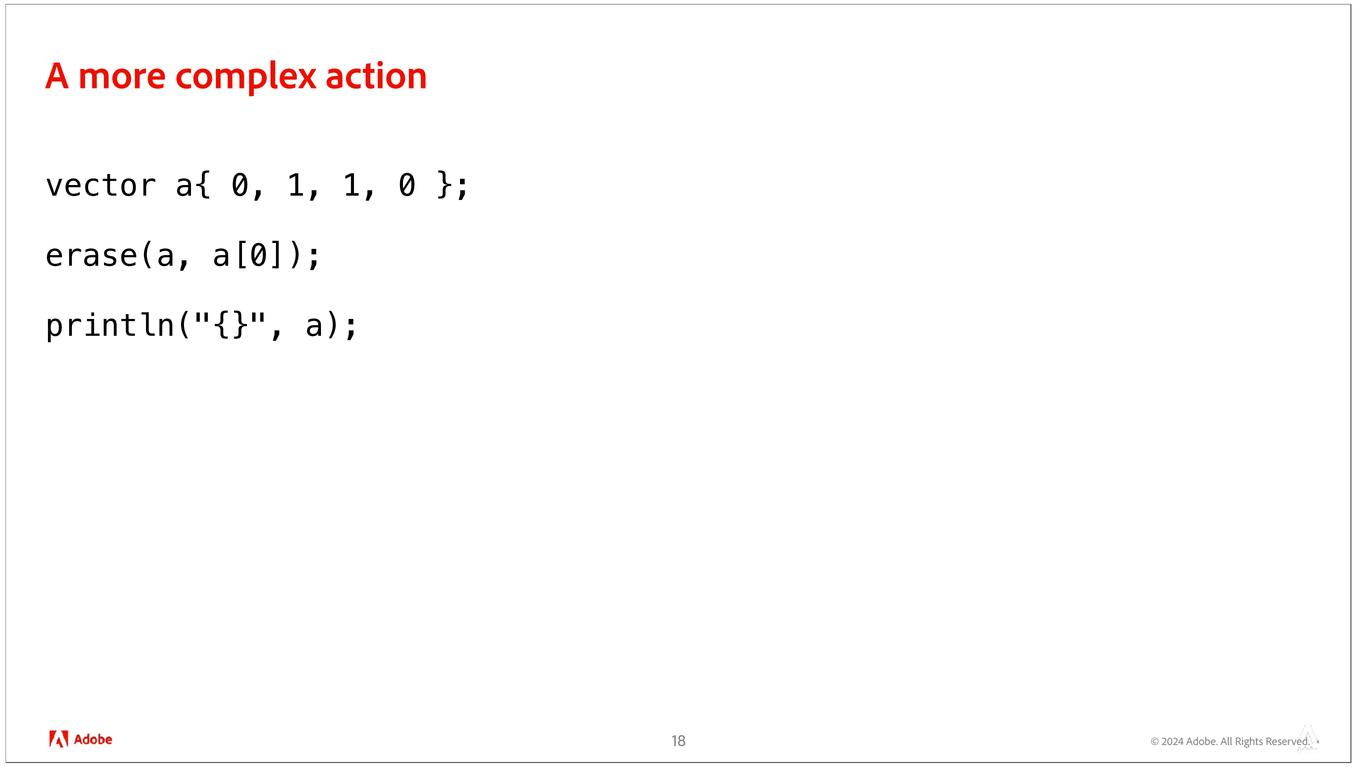
This may seem like a contrived example. But here is a real one <click>

A more complex action

```
// Offsets the value of x by n
void offset(int& x, const int& n) {
  for (int i = 0; i != n; ++x) ;
}
```

- What will this print?

```
int x{2};
offset(x, x);

println("{}", x);
```



A more complex action

```
vector a{ 0, 1, 1, 0 };

erase(a, a[0]);

println("{}", a);
```

What will this print… It depends on the implementation but here is one answer<click>

Why? After the code removes the first element matching a[0] (0), a[0] holds a 1, so the remaining 1 is removed, leaving the trailing 0. According to the standard, the answer is unspecified.

If arguments are aliased with mutation, local reasoning is broken for both the client and implementor.

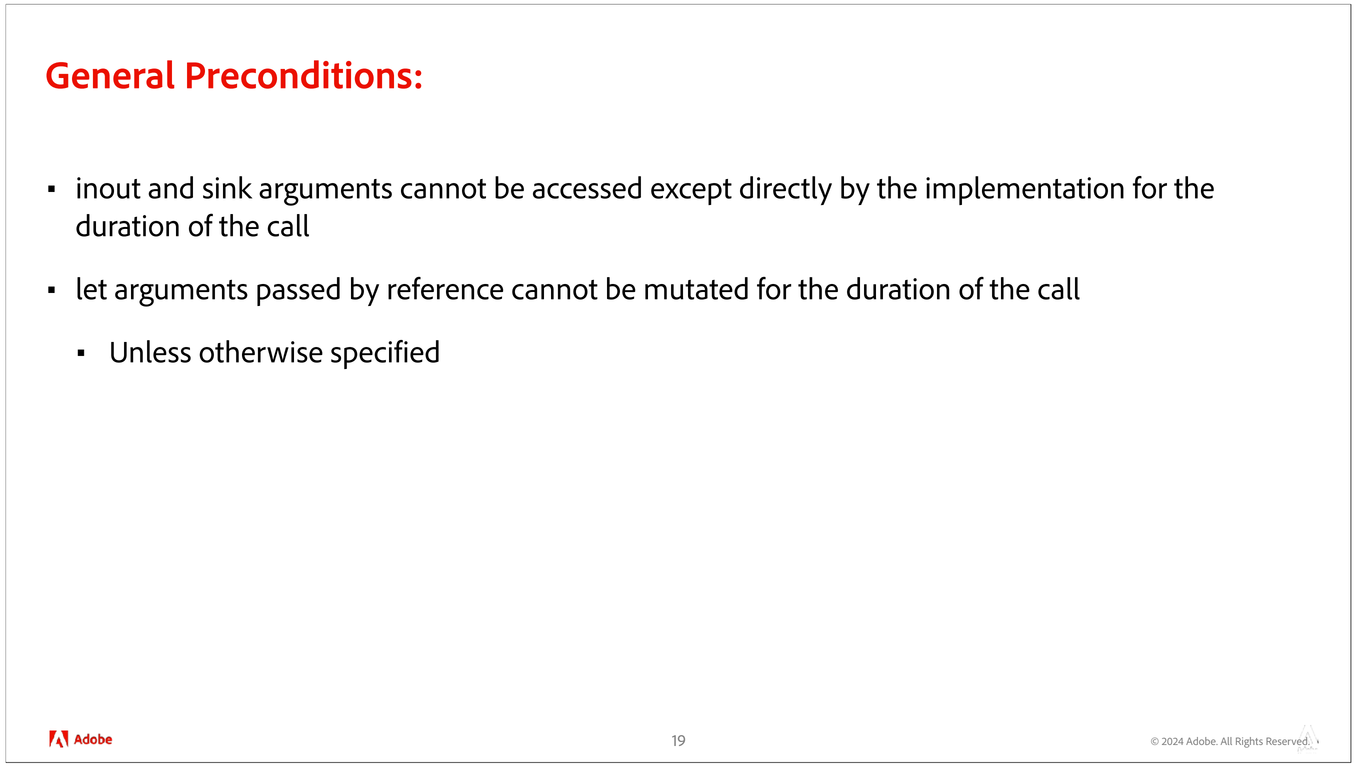
A more complex action

```
vector a{ 0, 1, 1, 0 };

erase(a, a[0]);

println("{}", a);
```

- What will this print?

```
[1, 0]
```

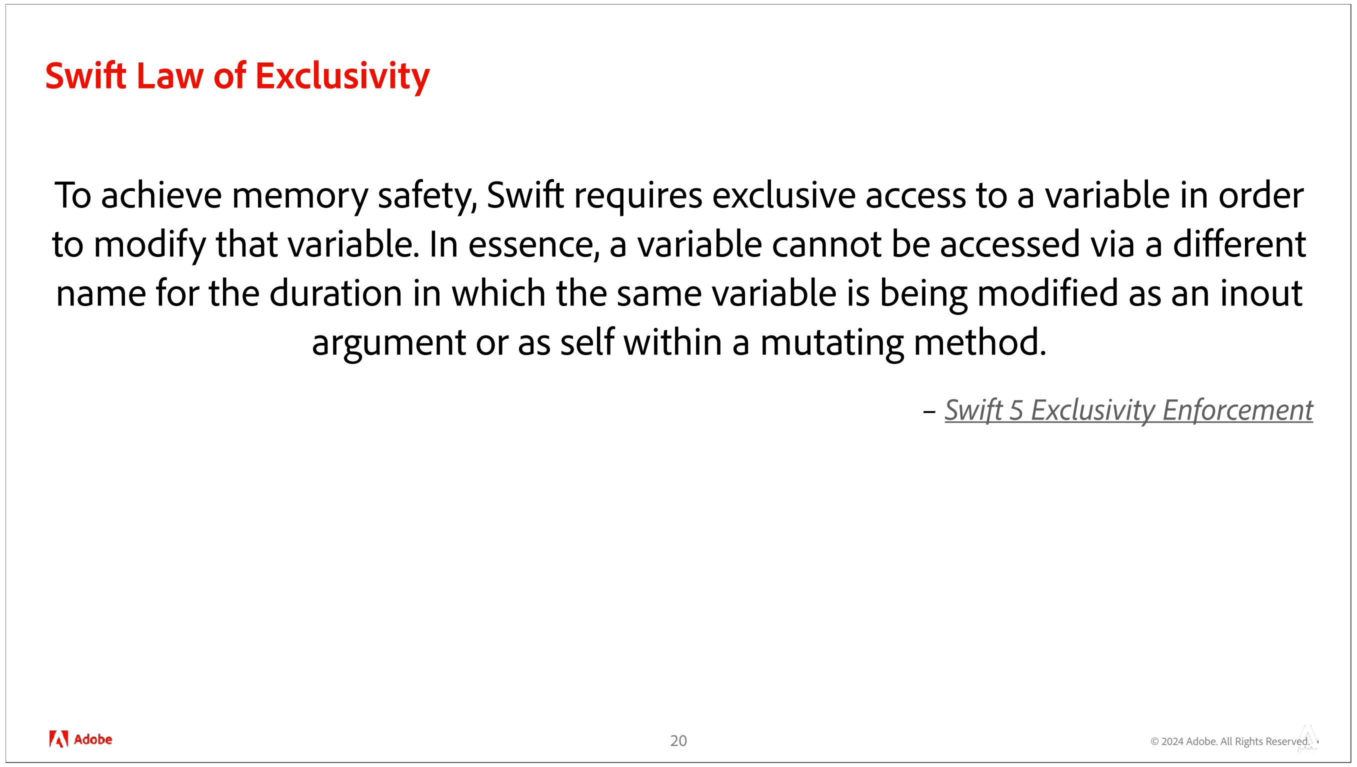
– https://godbolt.org/z/hP1dsTPsa



General Preconditions:

- inout and sink arguments cannot be accessed except directly by the implementation for the duration of the call
- let arguments passed by reference cannot be mutated for the duration of the call
    - Unless otherwise specified

These preconditions appear in various forms in several [modern? safe?] languages

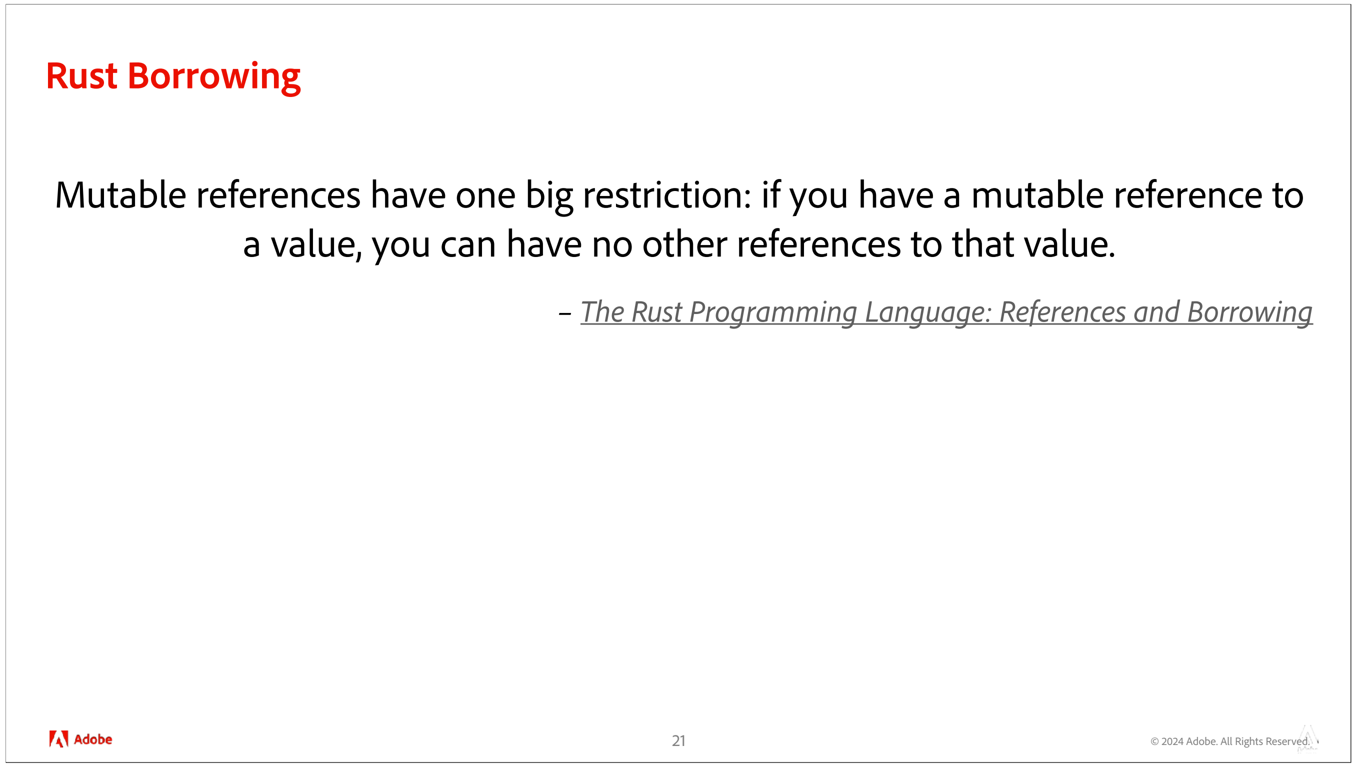


Swift Law of Exclusivity

> To achieve memory safety, Swift requires exclusive access to a variable in order to modify that variable. In essence, a variable cannot be accessed via a different name for the duration in which the same variable is being modified as an inout argument or as self within a mutating method.

– *Swift 5 Exclusivity Enforcement*

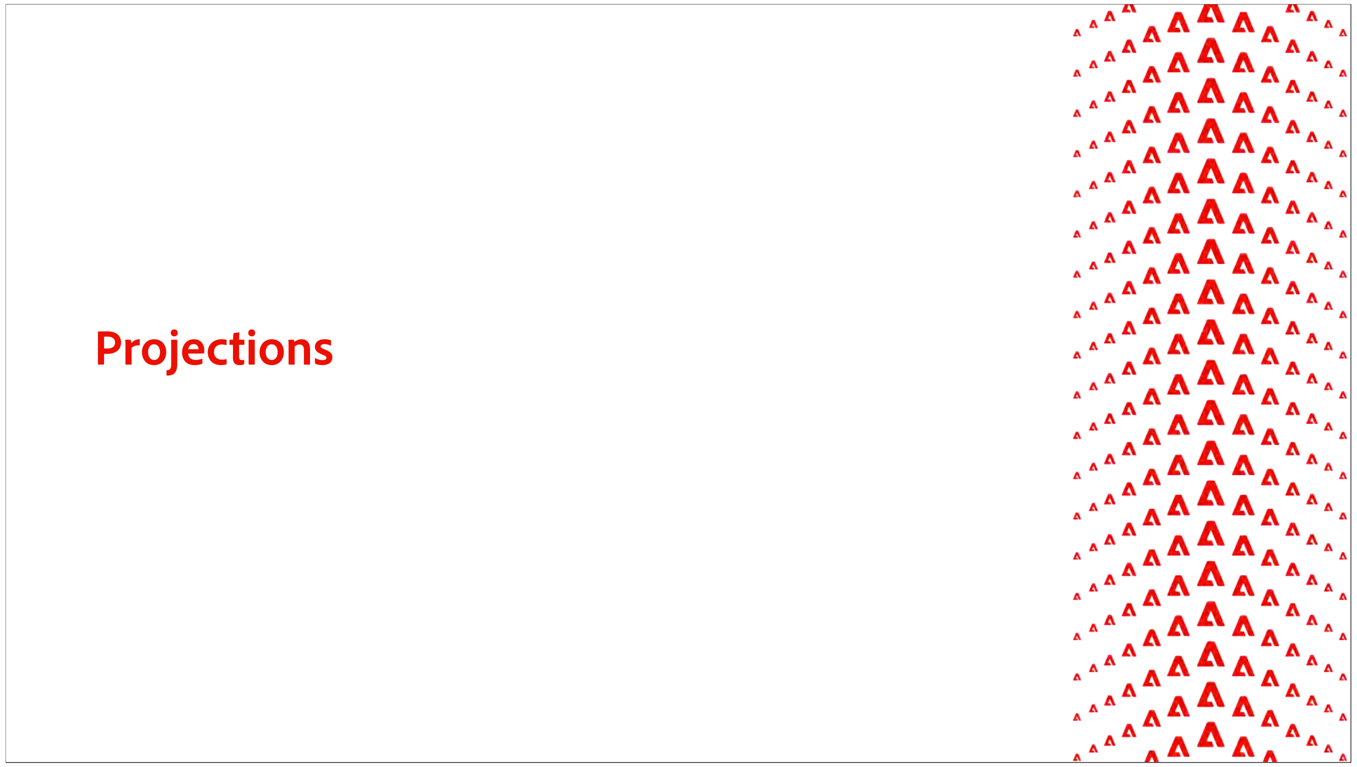
In Swift, this is known as "The Law of Exclusivity", a term coined by John McCall.



Rust Borrowing

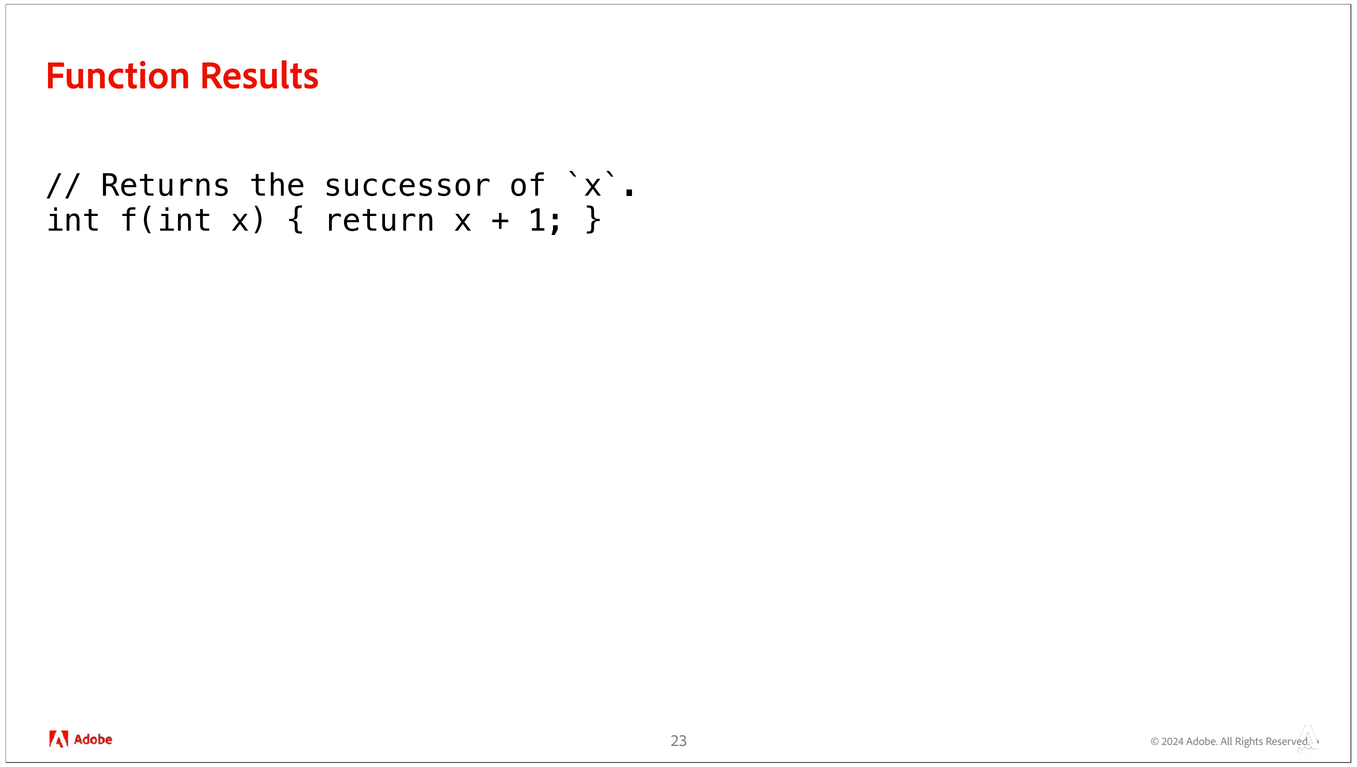
> Mutable references have one big restriction: if you have a mutable reference to a value, you can have no other references to that value.
>
> – *The Rust Programming Language: References and Borrowing*

In Rust, the borrow checker enforces this restriction. C++ does not have such a restriction. We must rely on conventions and diligence.



Projections

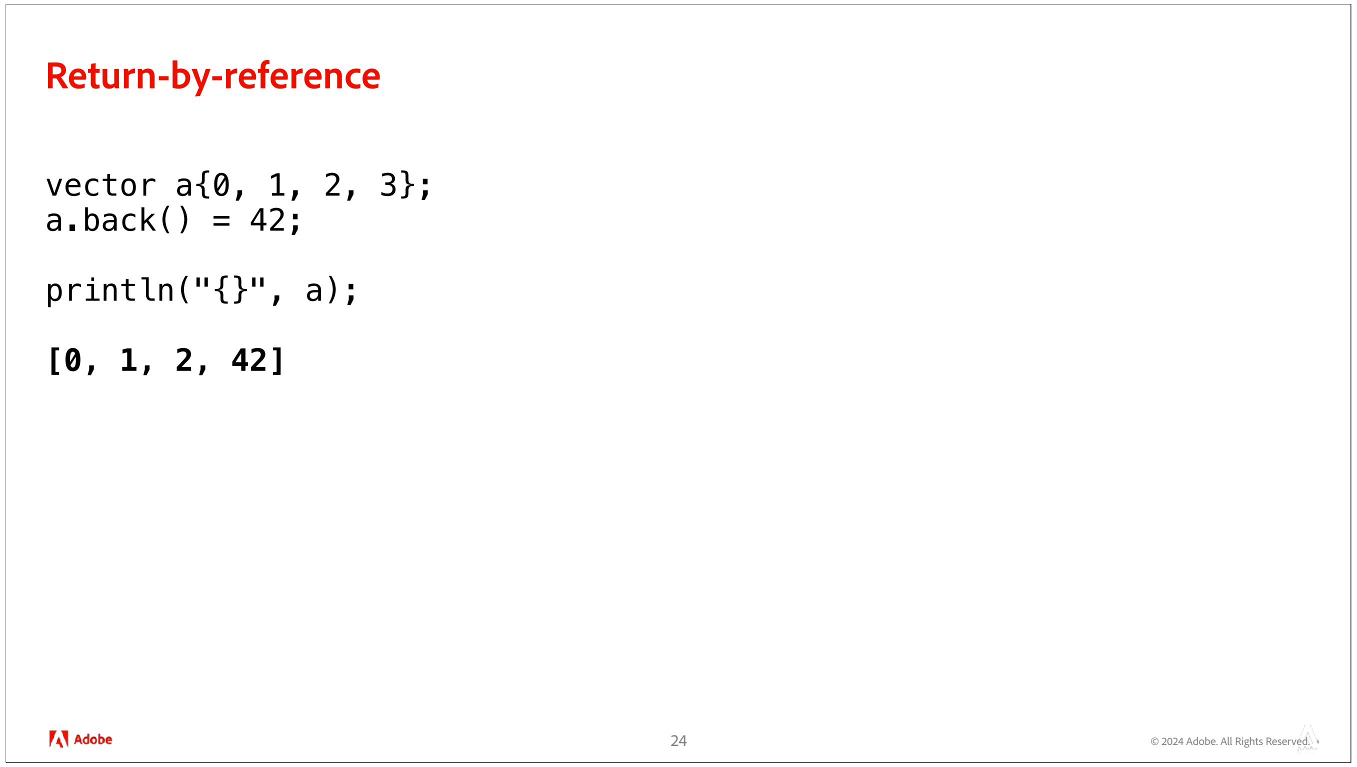
We haven't talked about function results yet. So let's start our discussion of projections there…



Function Results

```
// Returns the successor of `x`.
int f(int x) { return x + 1; }
```

Let's go back to an early simple function. Here, we are returning a new value. Would it ever make sense to return a reference from a function?



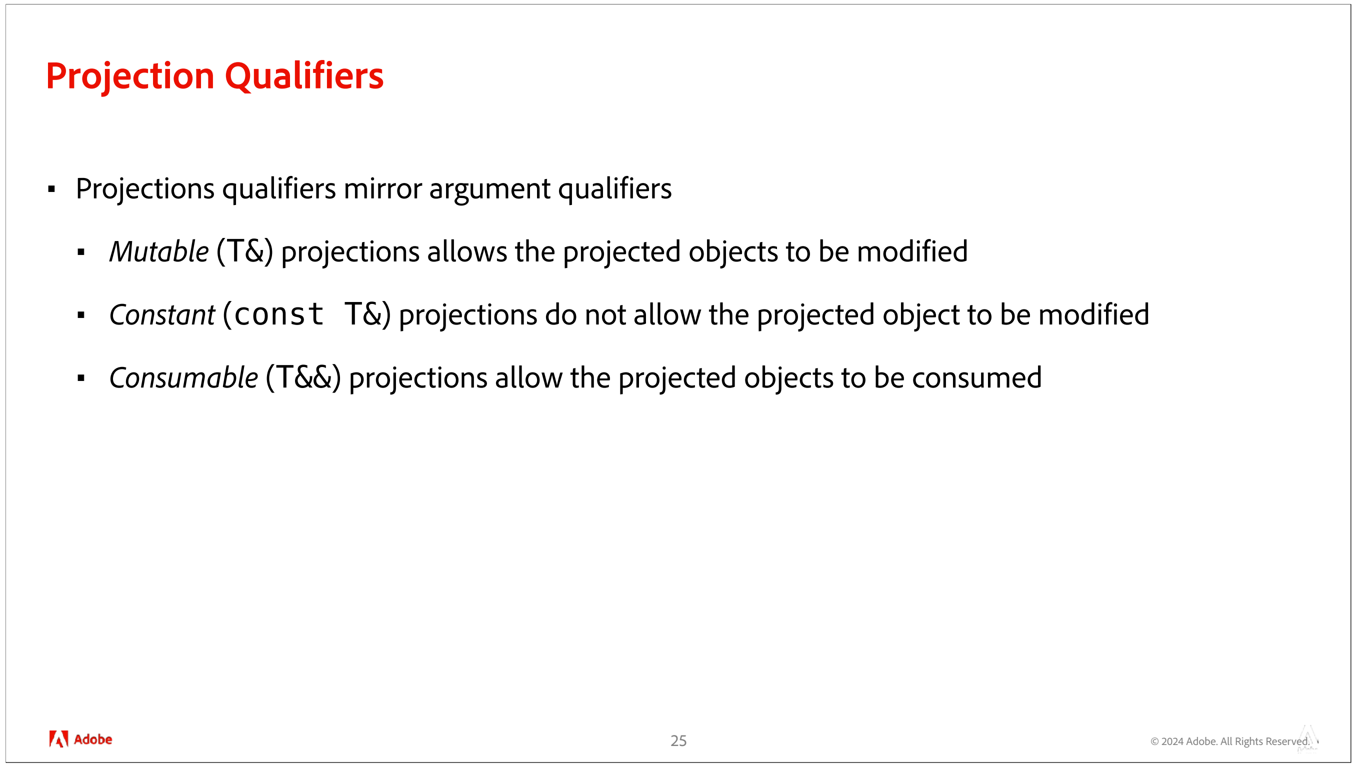
Return-by-reference

```
vector a{0, 1, 2, 3};
a.back() = 42;

println("{}", a);
```

```
[0, 1, 2, 42]
```

vector back is an example of returning a reference. There are many examples in the standard library, all assignment operators, indexing, and the min and max algorithms (by const reference, unfortunately)…

When we return a reference to a _part_ of something (and the whole is a part of the whole), we refer to it as a _projection_.



Projection Qualifiers

- Projections qualifiers mirror argument qualifiers
  - *Mutable* (T&) projections allows the projected objects to be modified
  - *Constant* (`const T&`) projections do not allow the projected object to be modified
  - *Consumable* (T&&) projections allow the projected objects to be consumed

The fact that projection qualifiers mirror argument qualifiers is not a coincidence -
By reference arguments _are_ projections.

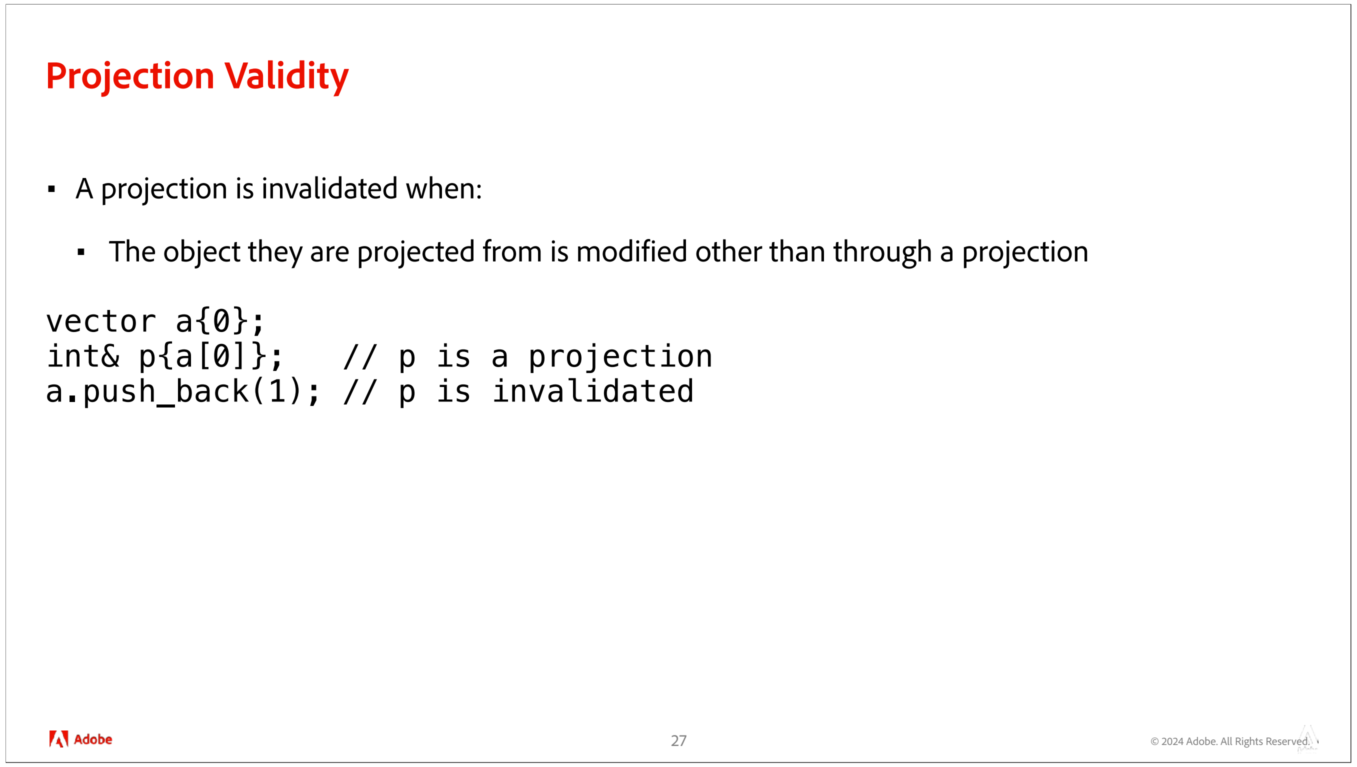
Projection Qualifiers

- Returning consumable projections are uncommon
  - Usually return by-value is used but consumables may be more efficient when extracting a value from an rvalue:

```
T&& extract() &&;
```

  - Mutable projections may also be consumed but require an additional operation to restore invariants on the owning object. i.e.

```
auto e{std::move(a.back());}
a.pop_back(); // erase the moved-from object
```

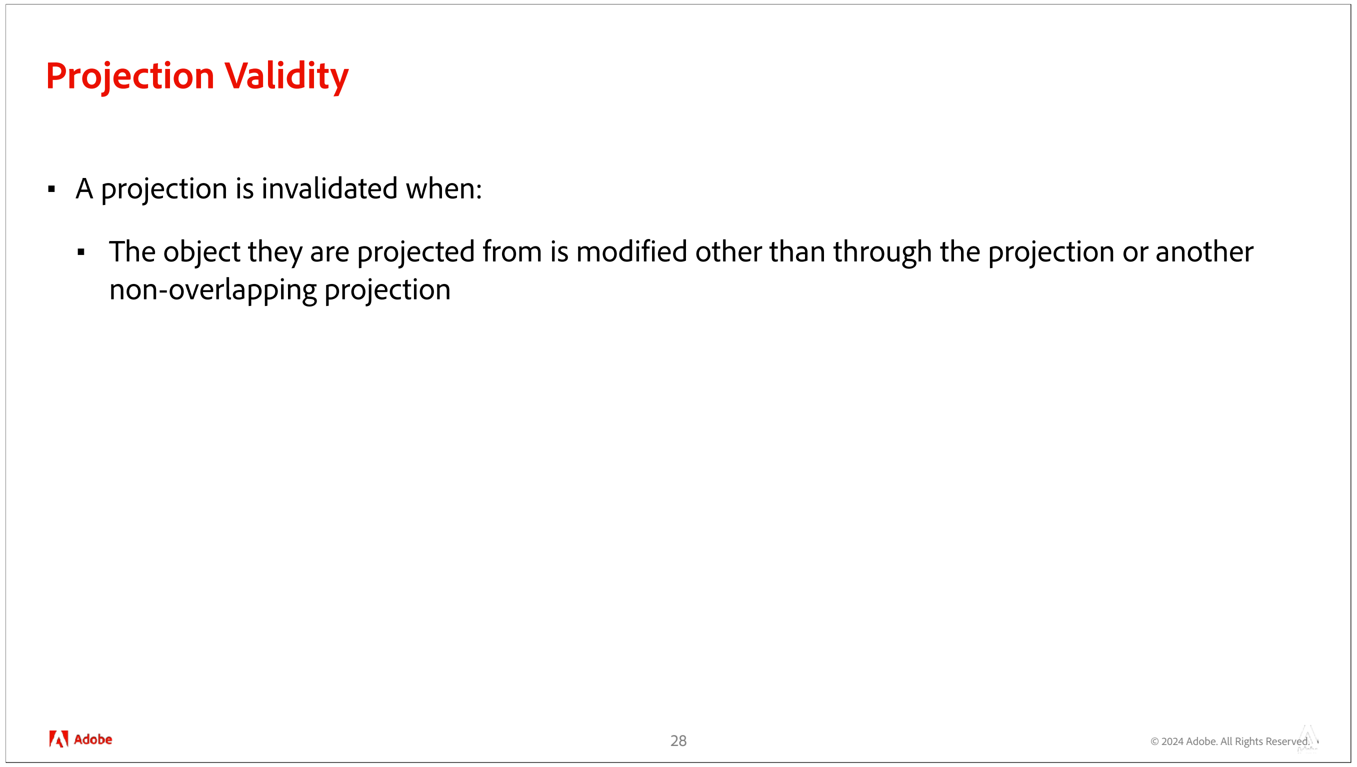


Projection Validity

- A projection is invalidated when:
  - The object they are projected from is modified other than through a projection

```
vector a{0};
int& p{a[0]};   // p is a projection
a.push_back(1); // p is invalidated
```

These are the general rules, a specific operation my provide stronger guarantees. It is the client responsibility to only pass valid projections to an operation



Projection Validity

- A projection is invalidated when:
  - The object they are projected from is modified other than through the projection or another non-overlapping projection

Adobe

These are the general rules, a specific operation my provide stronger guarantees. Unless otherwise specified.

Projection Validity

- A projection is invalidated when:
  - The object they are projected from is modified other than through the projection or another non-overlapping projection

```
vector a{0, 1, 2, 3};
const e& = a.back();
a.clear(); // invalidates e
```

Projection Validity

- A projection is invalidated when:
    - The object they are projected from is modified other than through the projection or another non-overlapping projection

```
vector a{0, 1, 2, 3};             vector a{0, 1, 2, 3};
const e& = a.back();              const e& = a.back();
a.clear(); // invalidates e       a[2] = 42; // e is not invalidated
```

Projection Validity

- A projection is invalidated when:
    - The object they are projected from is modified other than through the projection or another non-overlapping projection

```
vector a{0, 1, 2, 3};
const e& = a.back();
a.clear(); // invalidates e
```

```
vector a{0, 1, 2, 3};
const e& = a.back();
a[2] = 42; // e is not invalidated
```

-
    - The lifetime of the object they are projected from ends

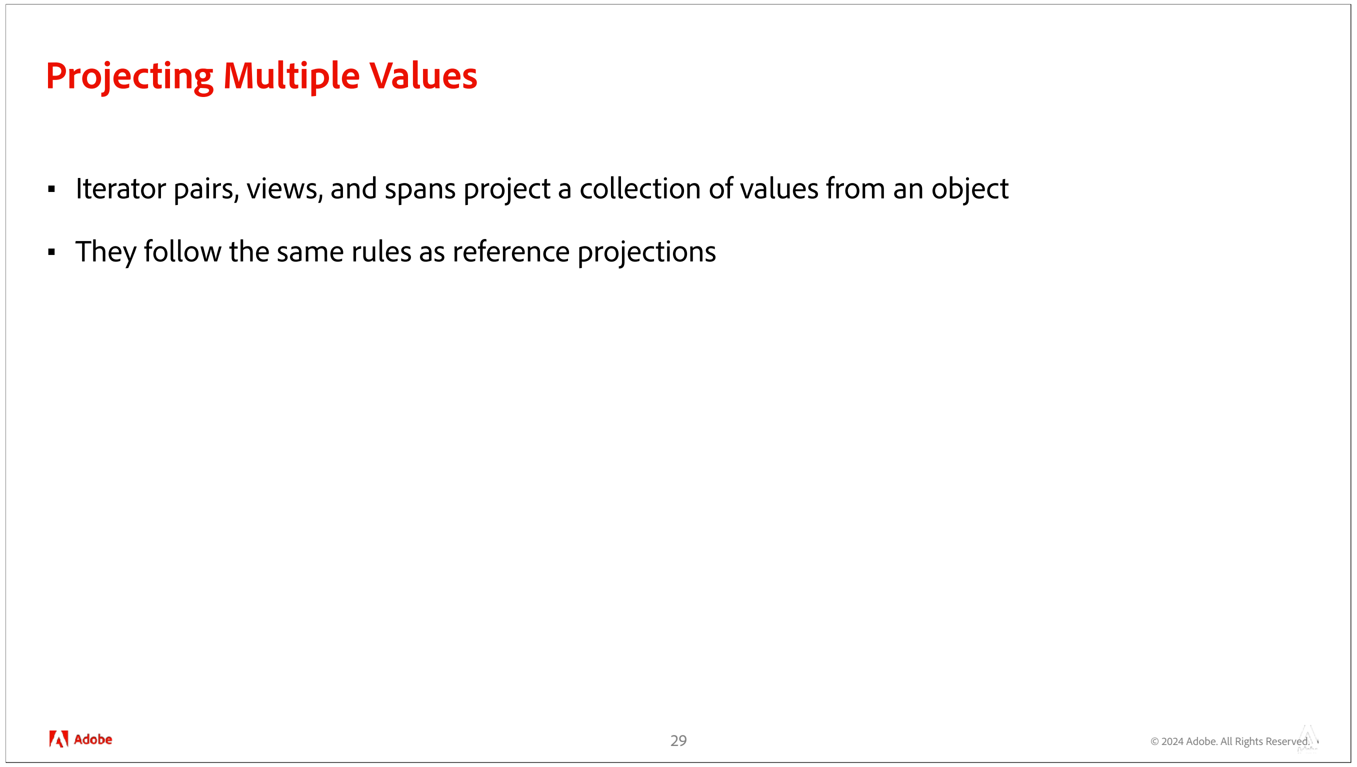
Projection Validity

- A projection is invalidated when:
    - The object they are projected from is modified other than through the projection or another non-overlapping projection

```
vector a{0, 1, 2, 3};             vector a{0, 1, 2, 3};
const e& = a.back();              const e& = a.back();
a.clear(); // invalidates e       a[2] = 42; // e is not invalidated
```

- The lifetime of the object they are projected from ends

```
int& p{vector{0}[0]}; // p is invalidated right after creation!
```



Projecting Multiple Values

- Iterator pairs, views, and spans project a collection of values from an object
- They follow the same rules as reference projections

Adobe

Copy has specific rules about overlapping ranges and copying to the left - as with our other rules there is an "unless otherwise specified" clause. If you rely on "otherwise specified" behavior - document it with a link the the relevant documentation.

Projecting Multiple Values

- Iterator pairs, views, and spans project a collection of values from an object
- They follow the same rules as reference projections

```
vector a{3, 2, 1, 0};
copy(begin(a), begin(a) + 2, begin(a) + 1); // Invalid – overlapping
```

Projecting Multiple Values

- Iterator pairs, views, and spans project a collection of values from an object
- They follow the same rules as reference projections

```
vector a{3, 2, 1, 0};
copy(begin(a), begin(a) + 2, begin(a) + 1); // Invalid – overlapping

vector a{3, 2, 1, 0};
copy(begin(a), begin(a) + 2, begin(a) + 2); // OK – not overlapping
```

Objects

Objects

```
void f(shared_ptr<widget> p);
```

Adobe

This seems like a ridiculous question - of course, the type is a shared widget pointer!
It could be a let or sink argument since it is pass by-value...
Do you think `f` is just operating on the pointer?
Maybe the type of the argument is the widget. And the widget is mutable so this could be an inout widget argument. It could be a nullptr, so it could be an optional inout widget argument!

f has exclusive access to the pointer (pass by-value). Am I confident f has exclusive mutable access to the widget for the duration of the call? Maybe the widget contains other child widgets held as shared pointers.

Why is the extent important?

Objects

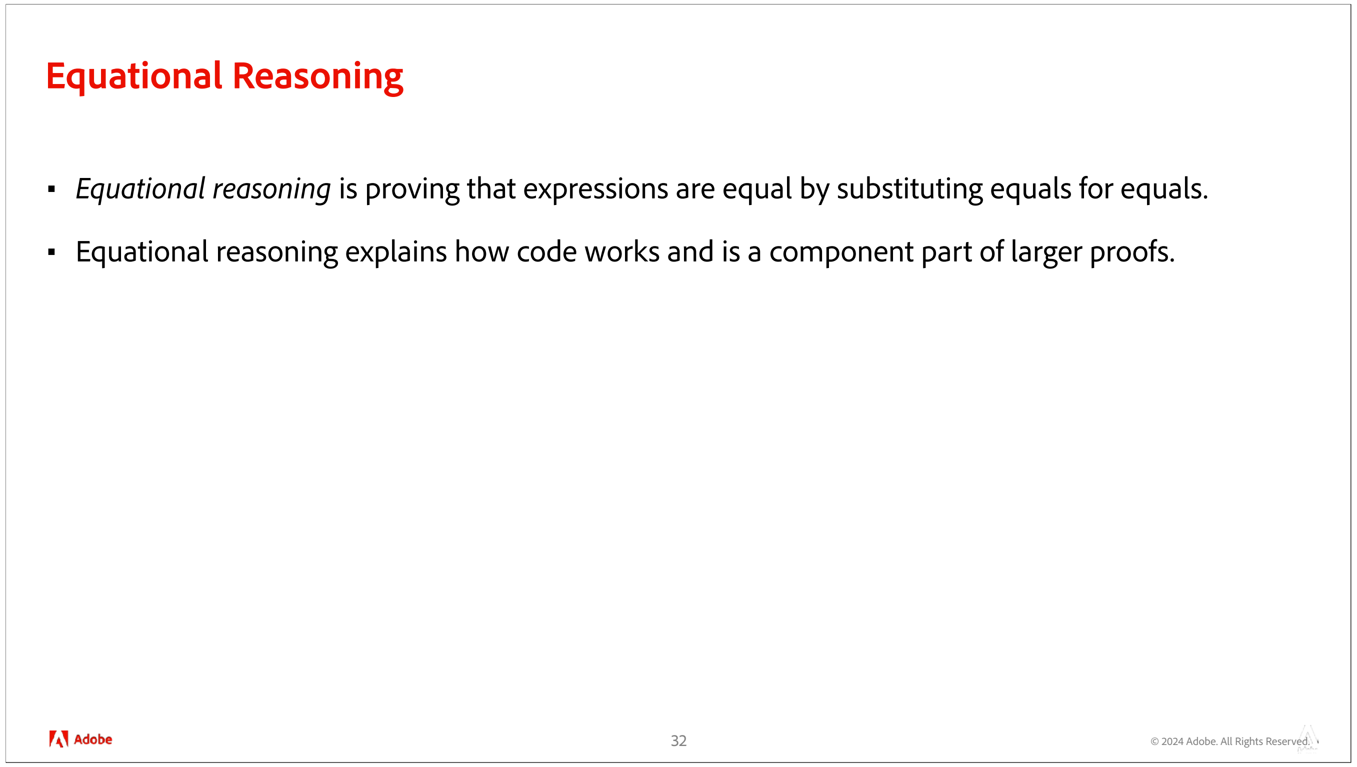
```
void f(shared_ptr<widget> p);
```

- What is the *type* of the argument for `f()`?

Objects

```
void f(shared_ptr<widget> p);
```

- What is the *type* of the argument for `f()`?
- To understand `f()` we need to understand the *extent* `p`



Equational Reasoning

- *Equational reasoning* is proving that expressions are equal by substituting equals for equals.
- Equational reasoning explains how code works and is a component part of larger proofs.

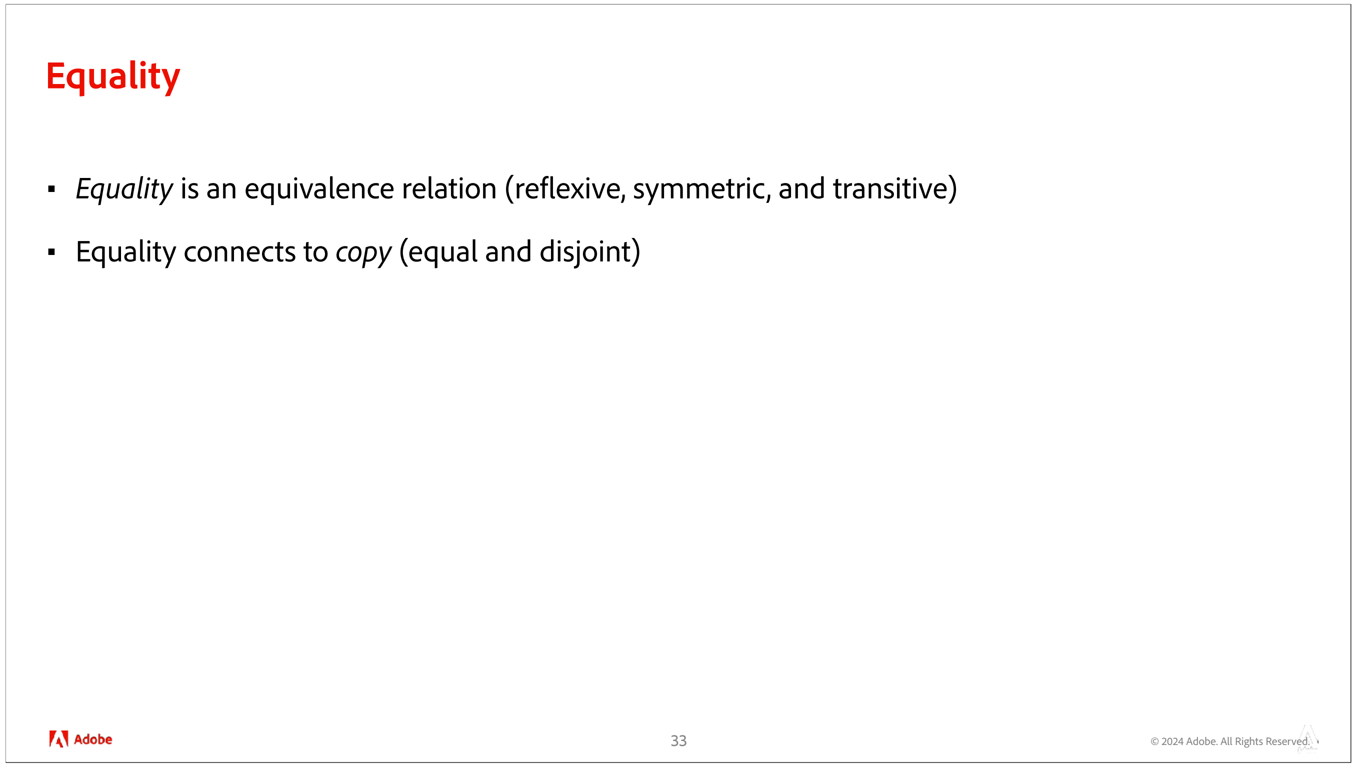
Adobe

Quick refresher on equality -

Equational Reasoning

- *Equational reasoning* is proving that expressions are equal by substituting equals for equals.
- Equational reasoning explains how code works and is a component part of larger proofs.

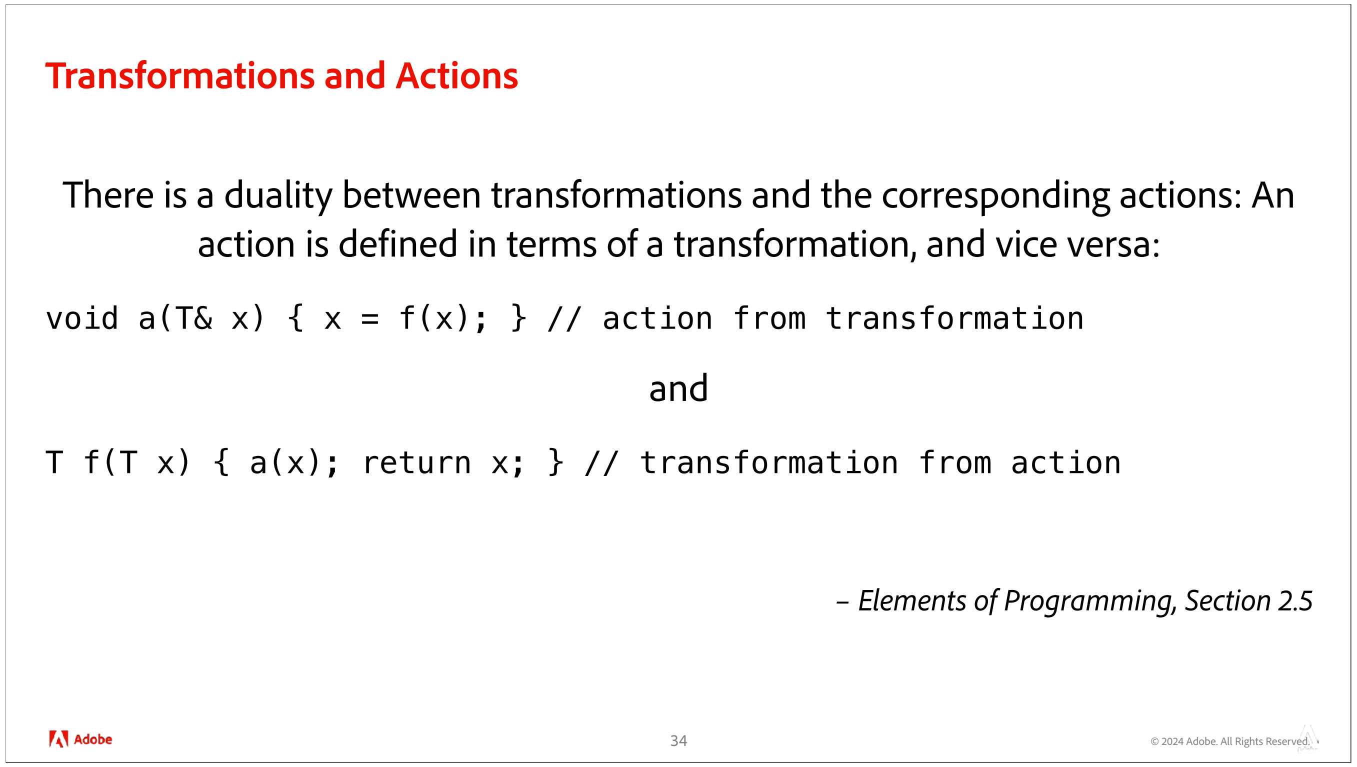
- To know if two values are equal, we need to know the *extent* of the values.



Equality

- *Equality* is an equivalence relation (reflexive, symmetric, and transitive)
- Equality connects to *copy* (equal and disjoint)

Equality also connects to move…
Recall the duality between transformations and actions -



Transformations and Actions

There is a duality between transformations and the corresponding actions: An action is defined in terms of a transformation, and vice versa:

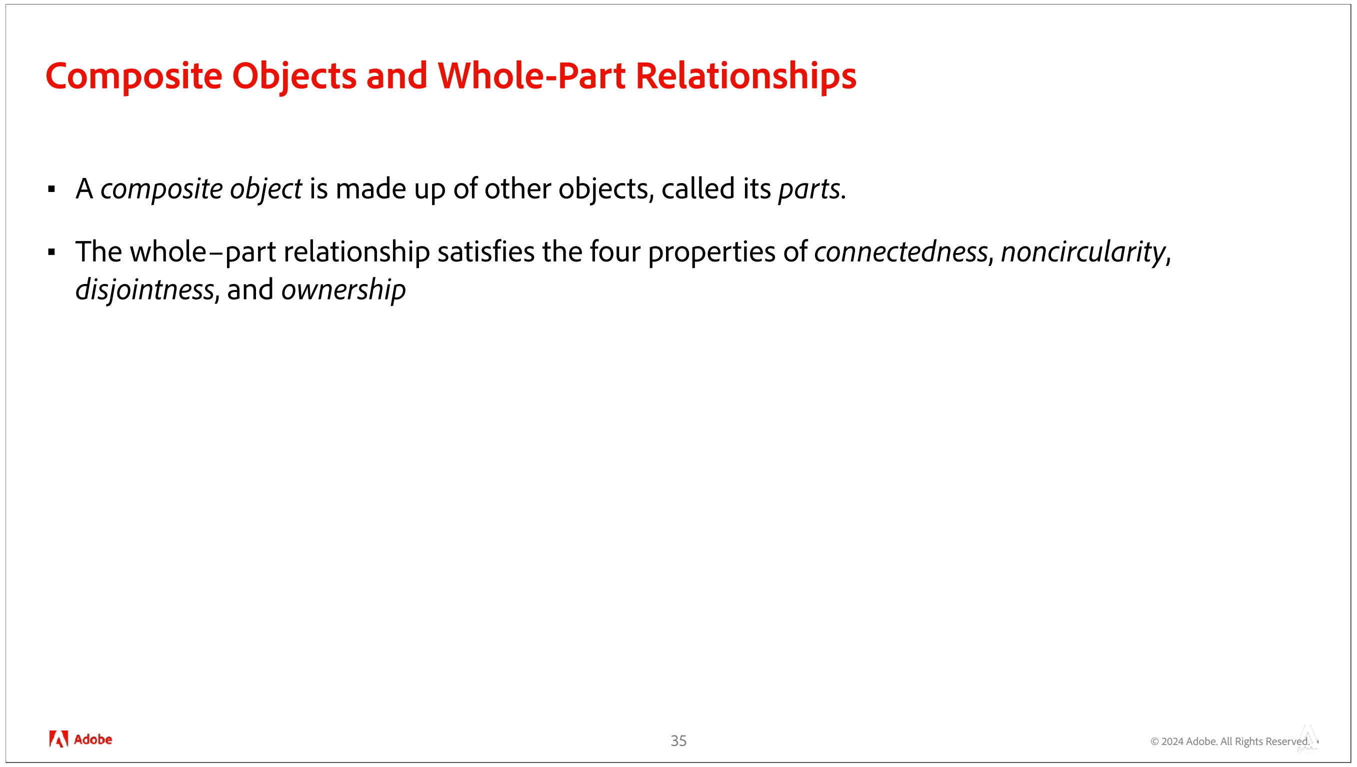
```
void a(T& x) { x = f(x); } // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

– *Elements of Programming, Section 2.5*

This is an example of equational reasoning. Projections are a proxy for a value, with rules the govern the validity of the proxy.



Composite Objects and Whole-Part Relationships

- A *composite object* is made up of other objects, called its *parts*.
- The whole–part relationship satisfies the four properties of *connectedness*, *noncircularity*, *disjointness*, and *ownership*

Adobe

a is a composite object with 4 integer part
b is a composite object with two named parts

disjointness - logically disjoint under mutation. immutable and copy-on-write objects may share storage.

Pointers, shared, unique or otherwise, witness a relationship. Which may, or may not, be a whole-part relationship. In an interface, their meaning is ambiguous and they are best avoided. Alone, they are disconnected from any whole.

Composite Objects and Whole-Part Relationships

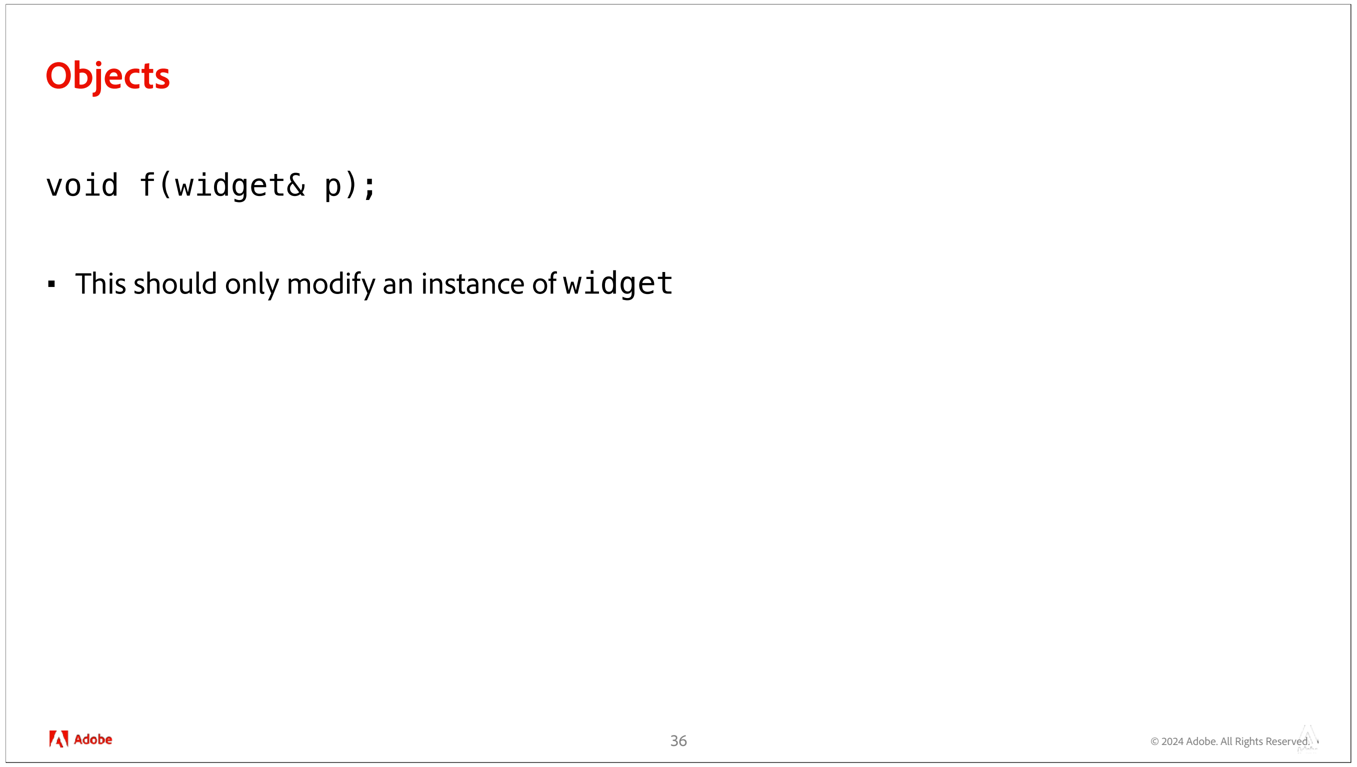
- A *composite object* is made up of other objects, called its *parts*.
- The whole–part relationship satisfies the four properties of *connectedness*, *noncircularity*, *disjointness*, and *ownership*

```
vector a{ 0, 1, 2, 3 };
```

Composite Objects and Whole-Part Relationships

- A *composite object* is made up of other objects, called its *parts*.
- The whole–part relationship satisfies the four properties of *connectedness*, *noncircularity*, *disjointness*, and *ownership*

```
vector a{ 0, 1, 2, 3 };

struct {
  string name{ "John" };
  int id{0}
} b;
```



Objects

```
void f(widget& p);
```

- This should only modify an instance of `widget`

Adobe

I prefer the sink/return-by-value form over mutation. It also allows for more concise code

But we need to talk a little about non-whole part relationships

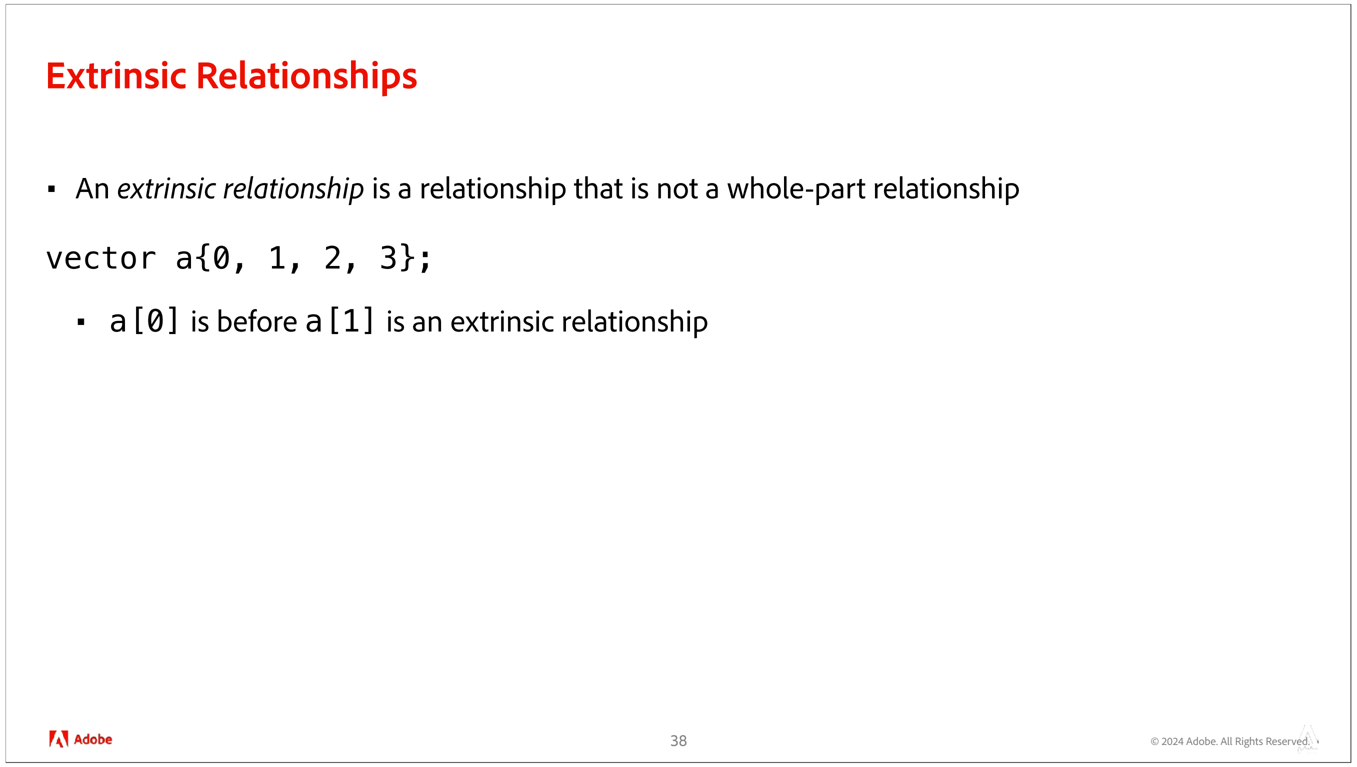
Objects

```
void f(widget& p);
```

- This should only modify an instance of `widget`
- It should be possible to express this as:

```
widget f(widget&& p);
```

Extrinsic Relationships



Extrinsic Relationships

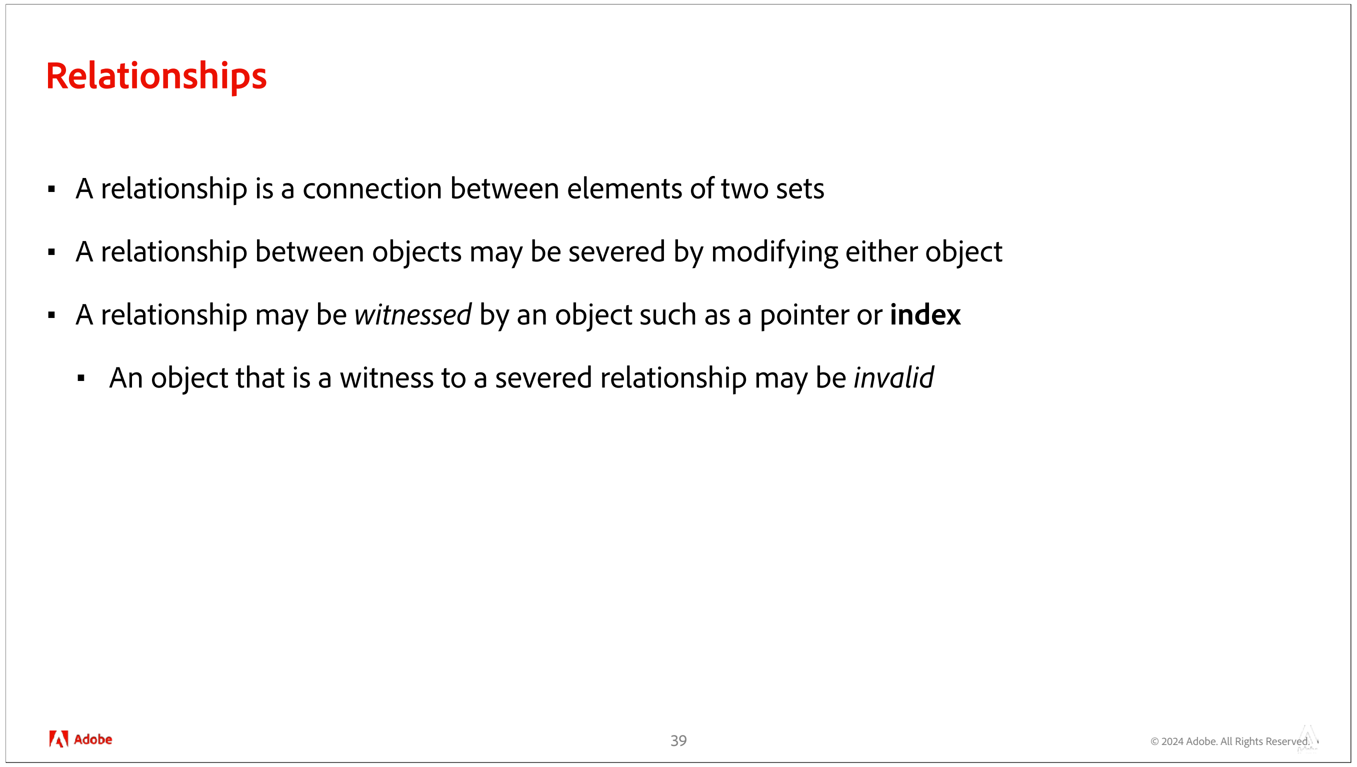
- An *extrinsic relationship* is a relationship that is not a whole-part relationship

```
vector a{0, 1, 2, 3};
```

  - `a[0]` is before `a[1]` is an extrinsic relationship

Adobe

Relationships exist all over in the code - the main challenge in programming isn't in functions or classes, but in finding and managing the essential relationships.



Relationships

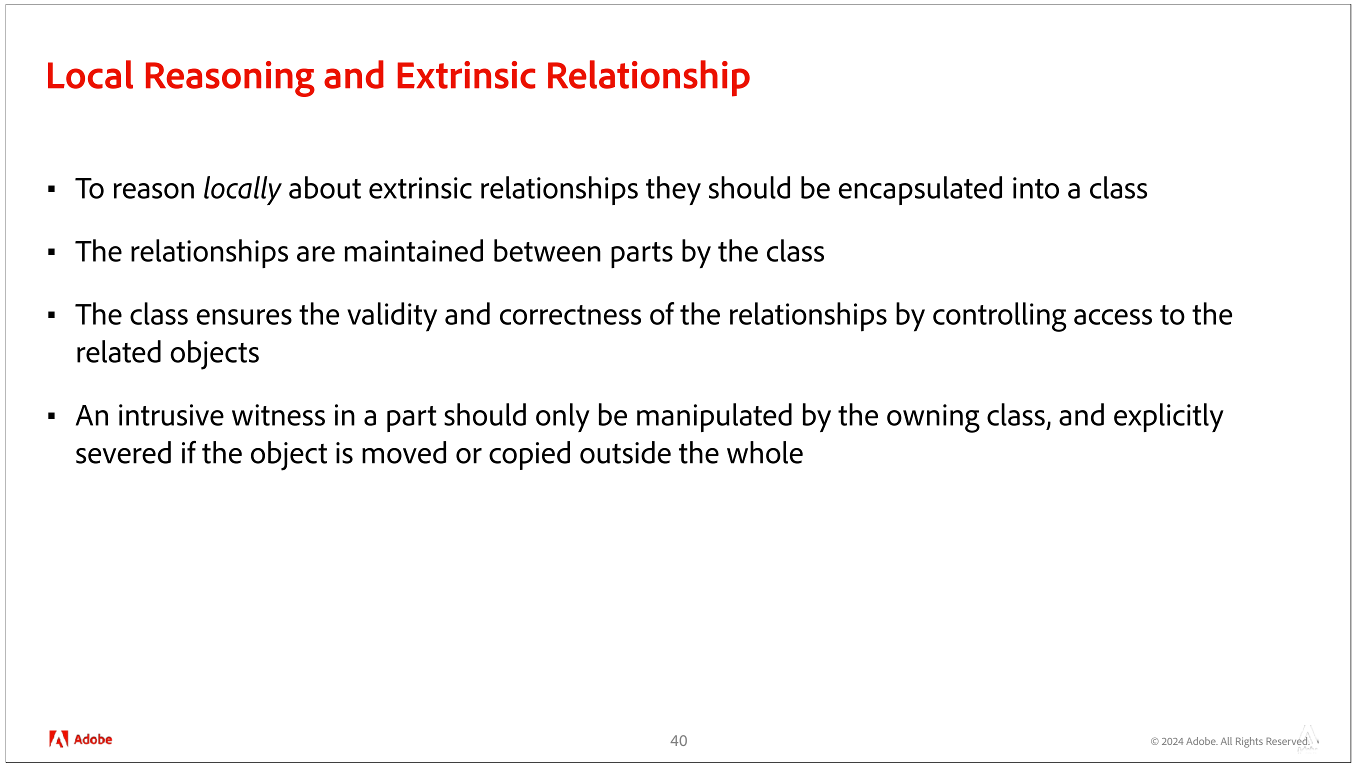
- A relationship is a connection between elements of two sets
- A relationship between objects may be severed by modifying either object
- A relationship may be *witnessed* by an object such as a pointer or **index**
    - An object that is a witness to a severed relationship may be *invalid*

Adobe

I'm emphasizing index - sometimes memory-safe or functional languages are described as solving the problems with pointers. They only solve the memory-safety problems, not correctness, and surprisingly (in the case of functional languages) not the problem of local reasoning

If I have an index to the largest element of an array, and I change the element such that it is no longer the largest, my index, as a witness to the relationship, is invalid.

This is where we get spooky action at a distance



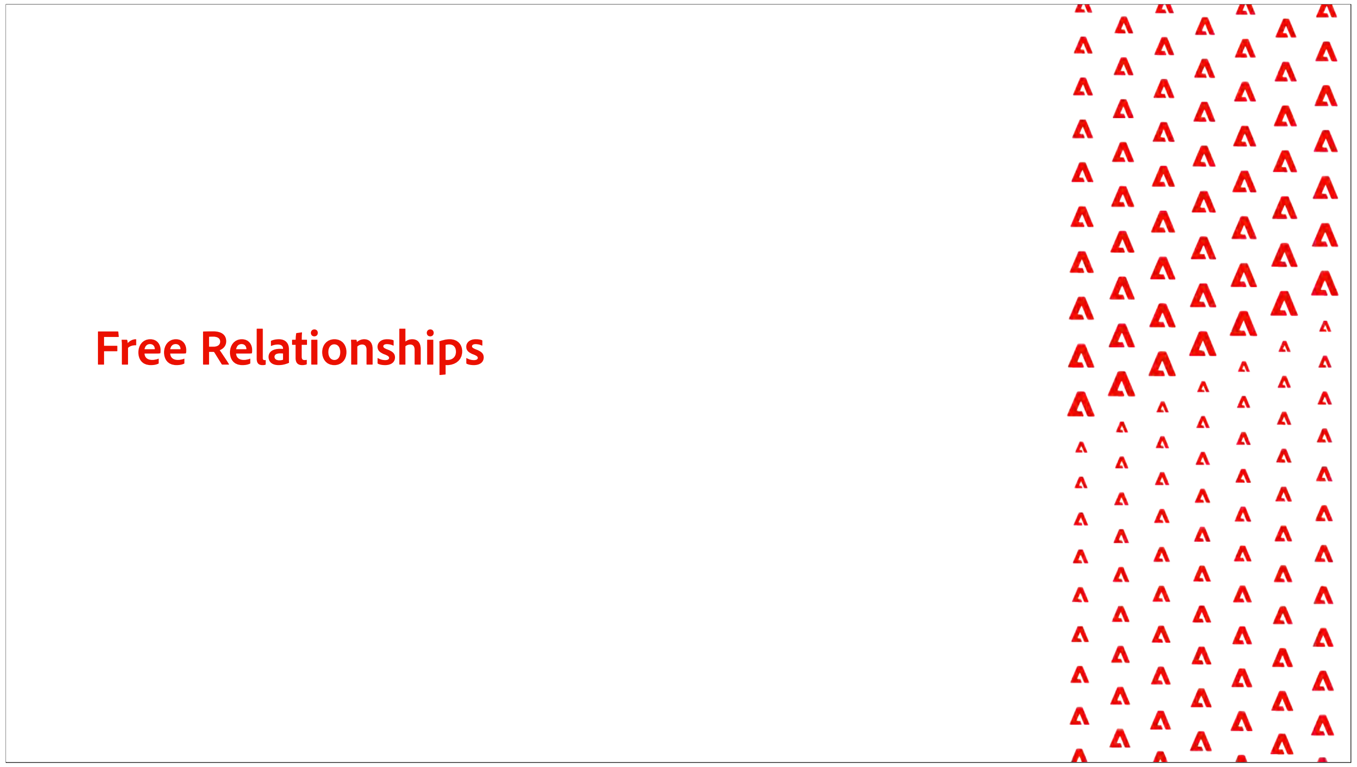
Local Reasoning and Extrinsic Relationship

- To reason *locally* about extrinsic relationships they should be encapsulated into a class
- The relationships are maintained between parts by the class
- The class ensures the validity and correctness of the relationships by controlling access to the related objects
- An intrusive witness in a part should only be manipulated by the owning class, and explicitly severed if the object is moved or copied outside the whole

These are class invariants

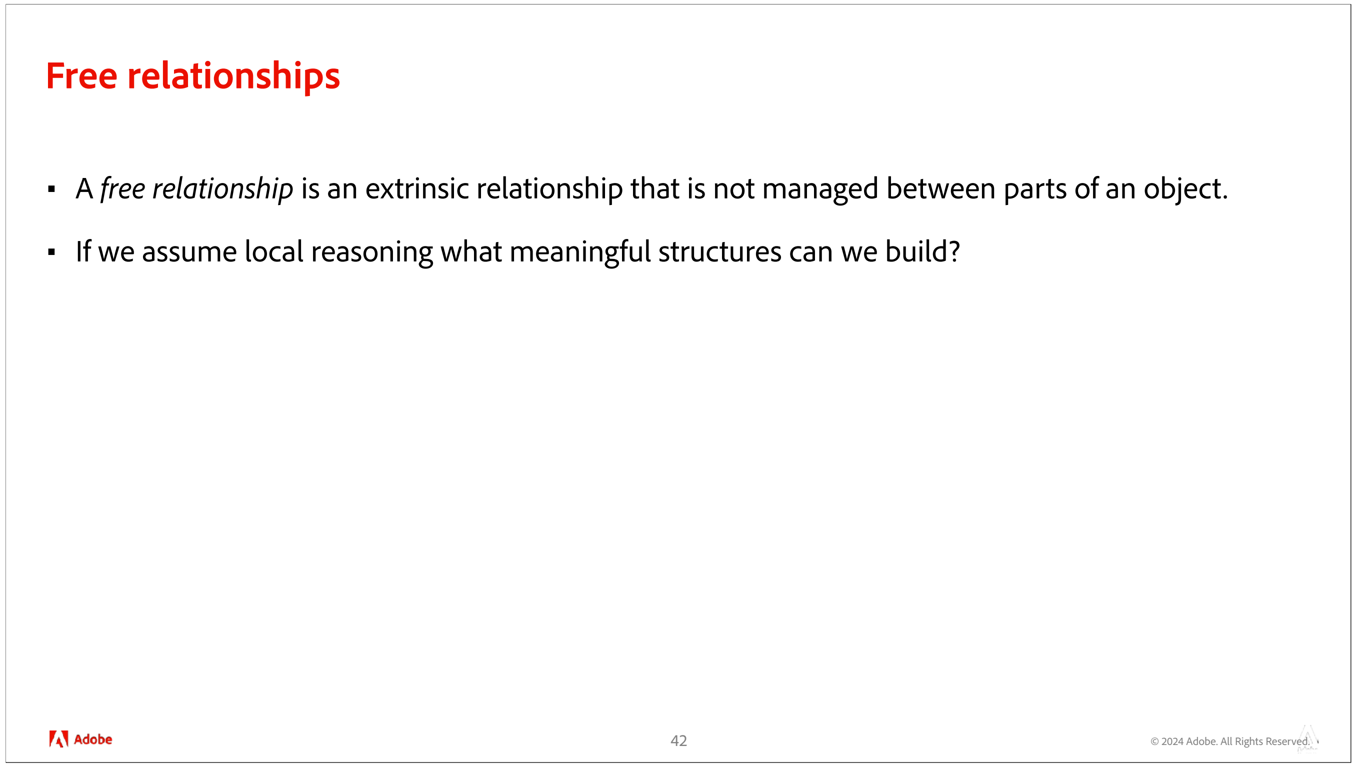
"explicitly severed" such as by nulling the pointer, using an optional, or other value such as a negative index to represent severed.

Linked list example. Splicing doesn't entangle lists. A container view of the world.



Free Relationships

In the 80s and into the 90s, there was a view that you could build systems at scale consisting of networks of objects. The entire OOP ethos was built around this idea. The view was always flawed but persists in reference-semantic languages.



Free relationships

- A *free relationship* is an extrinsic relationship that is not managed between parts of an object.
- If we assume local reasoning what meaningful structures can we build?

We only have local knowledge of each object, which follows a set of rules.

CALM

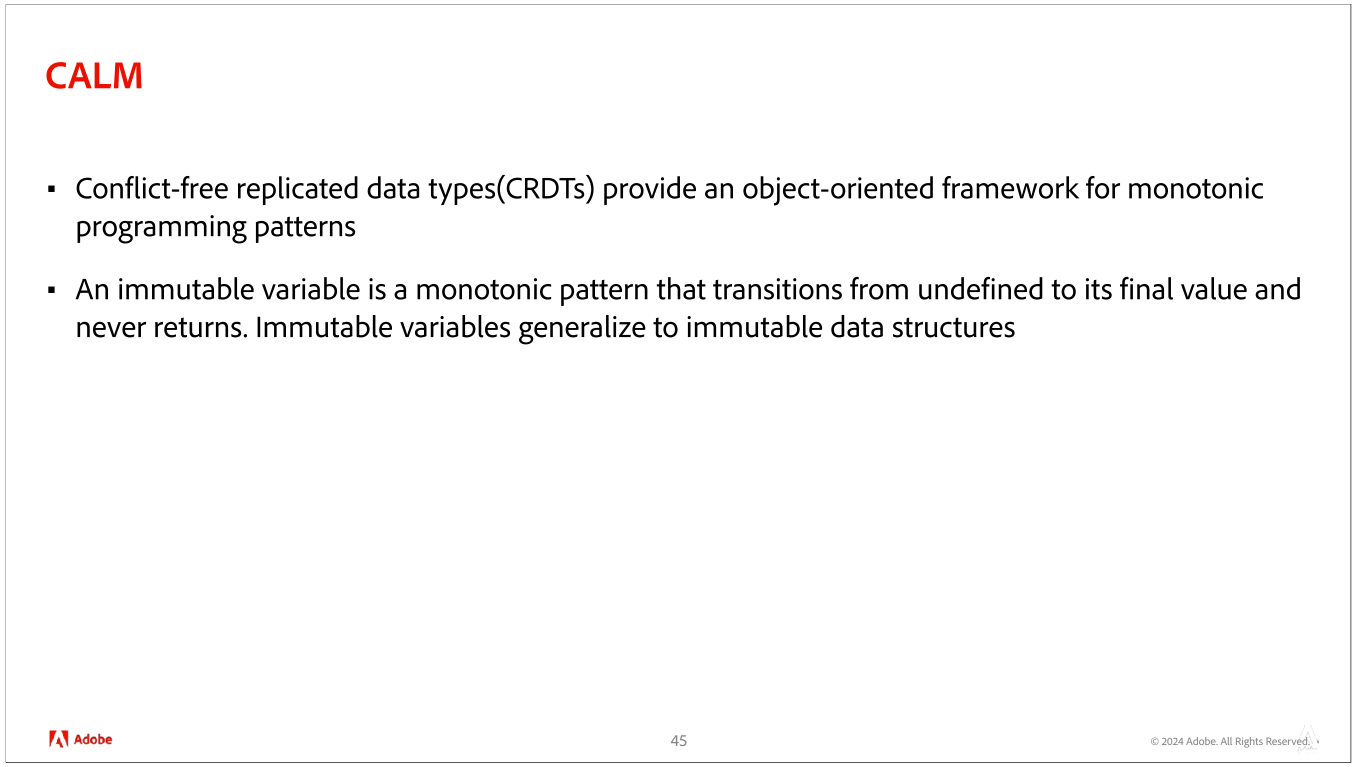
"Question: What is the family of problems that can be consistently computed in a distributed fashion without coordination, and what problems lie outside that family?"

– *Keeping CALM: WhenDistributed Consistency is Easy*

CALM

"Consistency As Logical Monotonicity (CALM). A program has a consistent, coordination-free distributed implementation if and only if it is monotonic."

– *Keeping CALM: WhenDistributed Consistency is Easy*



CALM

- Conflict-free replicated data types(CRDTs) provide an object-oriented framework for monotonic programming patterns
- An immutable variable is a monotonic pattern that transitions from undefined to its final value and never returns. Immutable variables generalize to immutable data structures

Immutable globals are okay. They don't require any additional coordination. Registries with tomb-stoning are another example. Delete is not a monotonic operation.

In 2008 I gave a Google tech talk on a possible future of software development. I conjectured that _at_ some scale we require coordination free computation. That scale is determined by the latency required for coordination. Significant progress has been made in recent years in this space, but there are still many open issues.

Summary

Summary

- Interfaces should make the scope of the operation clear
- Projections provide an efficient way to achieve value semantics and manipulate parts
- It is the client's responsibility to uphold the Law of Exclusivity
    - Don't pass projections that overlap an inout argument projection
- Implementors provide types with value semantics
- Confine extrinsic relationships between parts within a class

About the artist

Leandro Alzate

Berlin-based illustrator Leandro Alzate mixes bright color palettes and stylized characters in his fanciful work for editorial and advertising clients. He draws inspiration from observing the ways people interact, and combines that with his passion for architectural shapes and spaces. He created this piece for the German Ministry of Economy to encourage people to explore work-from-home career opportunities. Working with brushes and vector shapes, Alzate created this piece entirely in Adobe Photoshop.

Made with
Adobe Photoshop

Adobe

Artwork by **Leandro Alzate**