accu conference 2024

Regular, Revisited

Victor Ciura

Regular, Revisited

ACCU

April 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
M365 Substrate

Abstract

"Regular" is not exactly a new concept. If we reflect back on STL and its design principles, as best described by Alexander Stepanov in his "Fundamentals of Generic Programming" paper, we see that regular types naturally appear as necessary foundational concepts in programming.

Why do we need to bother with such taxonomies? Because STL assumes such properties about the types it deals with and imposes such conceptual requirements for its data structures and algorithms to work properly.

STL vocabulary types such as string_view, span, optional, expected etc., raise new questions regarding values types, whole-part semantics, copies, composite objects, ordering and equality.

Designing and implementing regular types is crucial in everyday programming, not just library design. Properly constraining types and function prototypes will result in intuitive usage; conversely, breaking subtle contracts for functions and algorithms will result in unexpected behavior for the caller.

This talk will explore the relation between Regular types (plus other concepts) and STL constructs, with examples, common pitfalls and guidance.

About me

Advanced Installer

Clang Power Tools

Visual C++

M365 Substrate

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

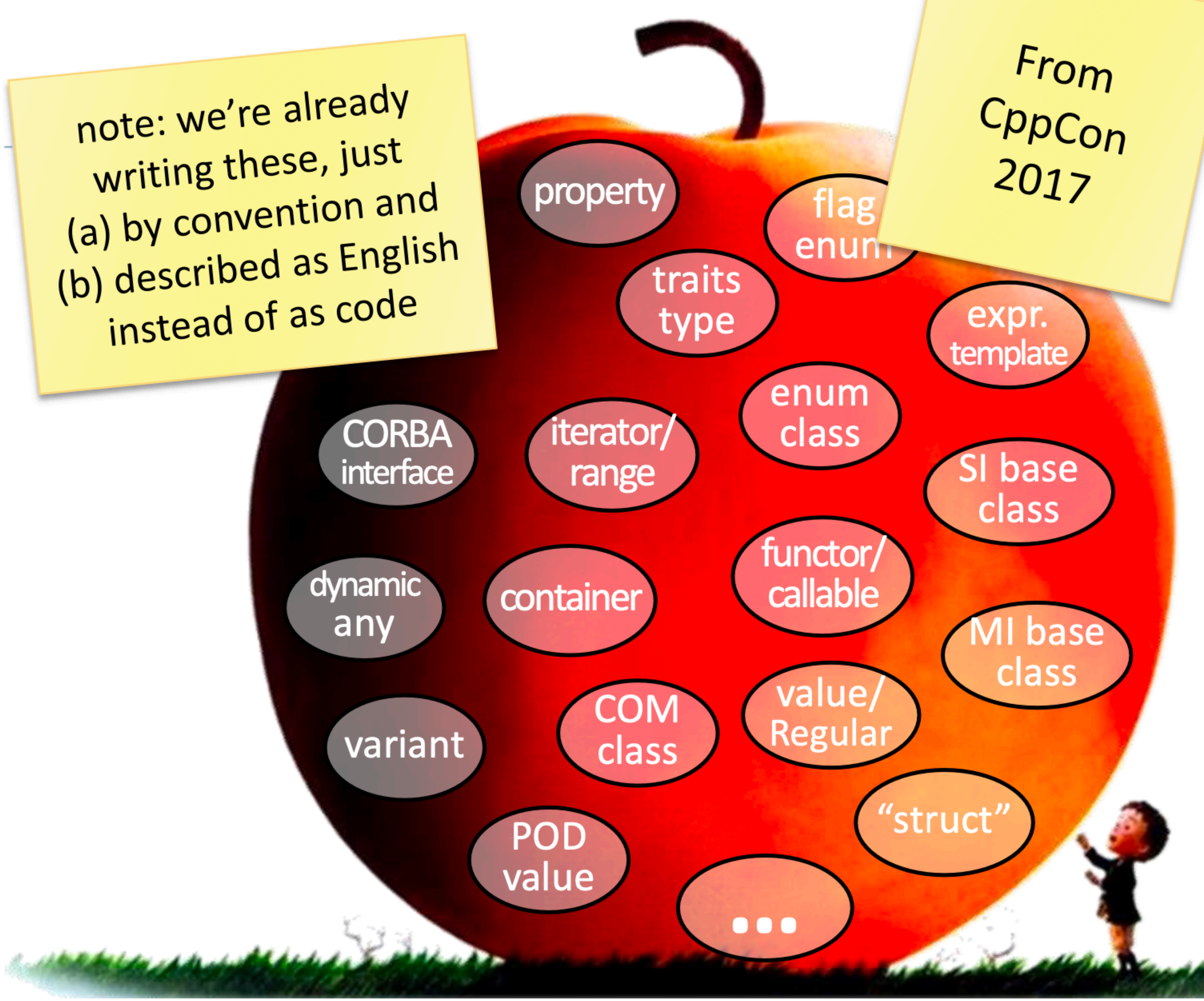
Conversation between Captain Picard and an alien:

Alien: "So, this teleporter, how does it work?"
Picard: "It beams people and arbitrary matter from one place to another."
Alien: "Hmmm... is it safe?"
Picard: "Yes, but earlier models were a hassle. They'd clone the person to another location. Then the teleporting chief would have to shoot the original.
Ask O'Brien, he was an intern during those times. A bloody mess, that's what it was."

Classified

The classes we write:

- RAII
- Utility
- Callable
- Wrappers
- Function bundles :(
- Polymorphic types / Hierarchies
- Containers
- Values
- ...



"The Evolution of C++ - A Typescript for C++", Herb Sutter - CppNow 2023 youtube.com/watch?v=fJvPBHErF2U

Some are more special than others...

v1 ~ 2018


cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
VICTOR CIURA
Regular Types
and Why Do I Care ?
cppcon
the c++ conference
Regular Types
and
Why Do I Care ?
September, 2018
CAPHYON
Victor Ciura
Technical Lead, Advanced Installer
www.advancedinstaller.com

Revisiting Regular Types

Anna Karenina principle to designing C++ types:

" *Good types are all alike.*
Every poorly designed type is poorly defined in its own way.

- adapted with apologies to Leo Tolstoy

Titus Winters, 2018
abseil.io/blog/20180531-regular-types

Why `Regular types` ?

Why are we talking about this ?

Why are we talking about this ?

We shall see that **Regular types** naturally appear as necessary foundational concepts in programming and try to investigate how these requirements fit in the ever expanding C++ standard, bringing new data structures & algorithms.

Why are we talking about this ?

Even the CppCoreGuidelines preach about this thing:

C.11: Make concrete types Regular

Regular types are easier to understand and reason about than types that are not regular (irregularities requires extra effort to understand and use).
The C++ built-in types are regular, and so are standard-library classes such as string, vector, and map.
Concrete classes without assignment and equality can be defined, but they are (and should be) rare.

isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-regular

Why are we talking about this ?

Even the CppCoreGuidelines preach about this thing:

T.46: Require template arguments to be at least Semiregular

Reason: Readability.

Preventing surprises and errors.

Most uses support that anyway.

isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-regular

Why are we talking about this ?

This talk is not just about `Regular` types

A moment to reflect back on **STL** and its **design principles**,

as best described by Alexander Stepanov in his 1998 paper

"Fundamentals of Generic Programming"

25 years!

This talk is not just about `Regular types`

This talk is not just about `Regular types`

`Values`

This talk is not just about `Regular types`

`Values`

`Objects`

This talk is not just about `Regular types`

`Values`

`Objects`

`Concepts`

This talk is not just about `Regular types`

`Values`

`Objects`

`Concepts`

`Ordering Relations`

This talk is not just about `Regular types`

`Values`

`Objects`

`Concepts`

`Ordering Relations`

`Requirements`

This talk is not just about `Regular types`

`Values`

`Objects`

`Concepts`

`Ordering Relations`

`Requirements`

`Equality`

This talk is not just about Regular types

Values

Objects

Whole-part semantics

Concepts

Ordering Relations

Requirements

Equality

This talk is not just about Regular types

Values

Objects

Whole-part semantics

Concepts

Ordering Relations

Requirements

Lifetimes

Equality

This talk is not just about `Regular types`

Values

Objects

Whole-part semantics

Concepts

Ordering Relations

Requirements

Lifetimes

Equality

Cpp Core Guidelines

This talk is not just about `Regular types`

Values

Objects

Whole-part semantics

C++17

Ordering Relations

Concepts

Requirements

Lifetimes

Cpp Core Guidelines

Equality

This talk is not just about `Regular` types

Values

Objects

Whole-part semantics

C++17

Concepts

Ordering Relations

C++20

Requirements

Lifetimes

Equality

Cpp Core Guidelines

This talk is not just about `Regular types`

Values

Objects

Whole-part semantics

C++17

Concepts

Ordering Relations

C++23

C++20

Requirements

Lifetimes

Equality

Cpp Core Guidelines

Modern C++ API Design

Type Properties

What properties can we use to describe types ?

Type Families

What combinations of type properties make useful / good type designs ?

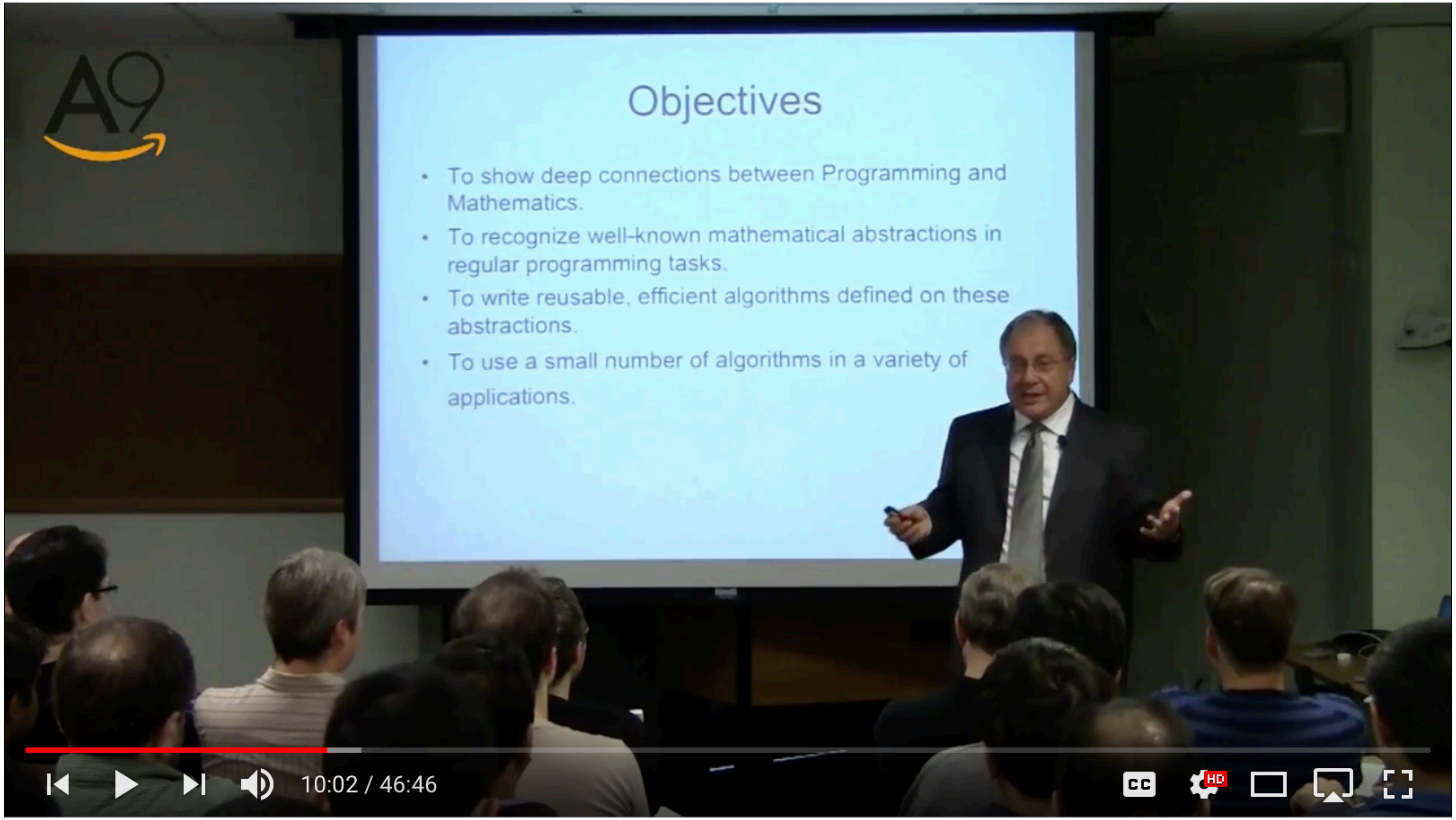
Titus Winters - Modern C++ API Design
youtube.com/watch?v=tn7oVNrPM8I

Let's start with the beginning...

2,000 BC

😁

~~Four~~ Three Algorithmic Journeys



Lectures presented at

A9

(2012)

Spoils of the Egyptians: Lecture 1 Part 1

https://www.youtube.com/watch?v=wrmXDxn_Zuc

~~Four~~ Three Algorithmic Journeys

I. Spoils of the Egyptians (10h)
How elementary properties of commutativity and associativity of addition and multiplication led to fundamental algorithmic and mathematical discoveries.

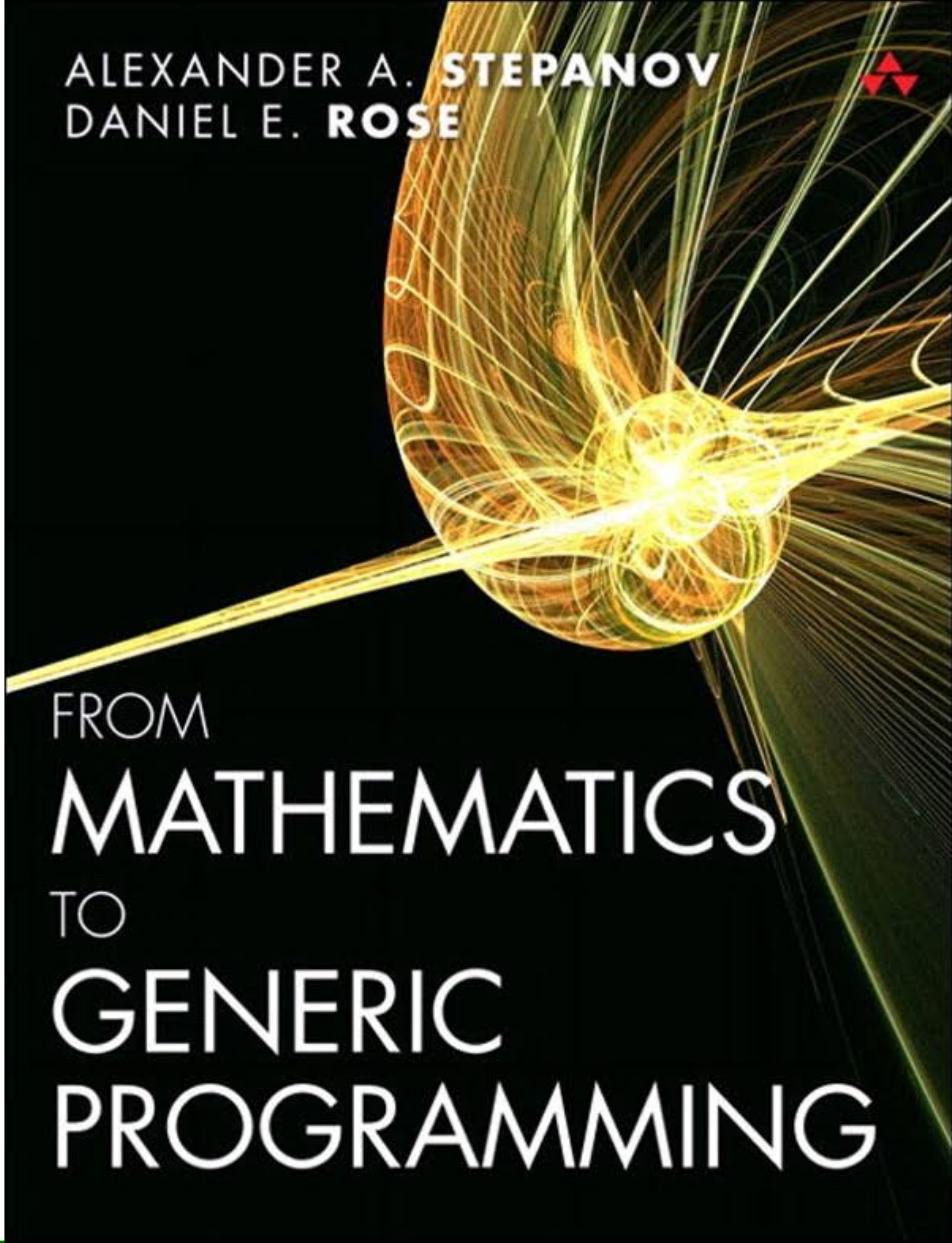
II. Heirs of Pythagoras (12h)
How division with remainder led to discovery of many fundamental abstractions.

III. Successors of Peano (10h)
The axioms of natural numbers and their relation to iterators.

Lectures presented at
A9

https://www.youtube.com/watch?v=wrmXDxn_Zuc

- Egyptian multiplication ~ **1900-1650 BC**
- Ancient Greek number theory
- Prime numbers
- Euclid's GCD algorithm
- Abstraction in mathematics
- Deriving generic algorithms
- Algebraic structures
- Programming concepts
- Permutation algorithms
- Cryptology (RSA) ~ **1977 AD**



In the beginning there were just `0`s and `1`s

#define

Datum

A datum is a finite sequence of `0`s and `1`s

Can represent anything...

🍌🍎🌴

#EoP

#define

Value Type

A **value type** is a correspondence between

a species (abstract/concrete) and a *set of datums*.

#EoP

`#define`

`Value`

Value is a datum together with its ***interpretation***.

Eg.

an integer represented in 32-bit two's complement, big endian

#EoP

#define

Value

Value is a datum together with its ***interpretation***.

Eg.

an integer represented in 32-bit two's complement, big endian

A value cannot change.

#EoP

Value Type & Equality

Lemma 1

If a value type is **uniquely** represented,

equality implies *representational equality*.

#EoP

Value Type & Equality

Lemma 1

If a value type is **uniquely** represented,

equality implies *representational equality*.

Lemma 2

If a value type is not ambiguous,

representational equality implies *equality*.

#EoP

#define

Object

An **object** is a representation of a concrete entity as a **value** in computer ***memory*** (address & length).

#EoP

#define

Object

An **object** is a representation of a concrete entity as a **value**

in computer ***memory*** (address & length).

An object has a **state** that is a ***value*** of some value type.

#EoP

`#define`

`Object`

An **object** is a representation of a concrete entity as a **value** in computer ***memory*** (address & length).

An object has a **state** that is a ***value*** of some value type.

The state of an object can change.

#EoP

#define

Type

Type is a *set of values* with the same interpretation function and operations on these values.

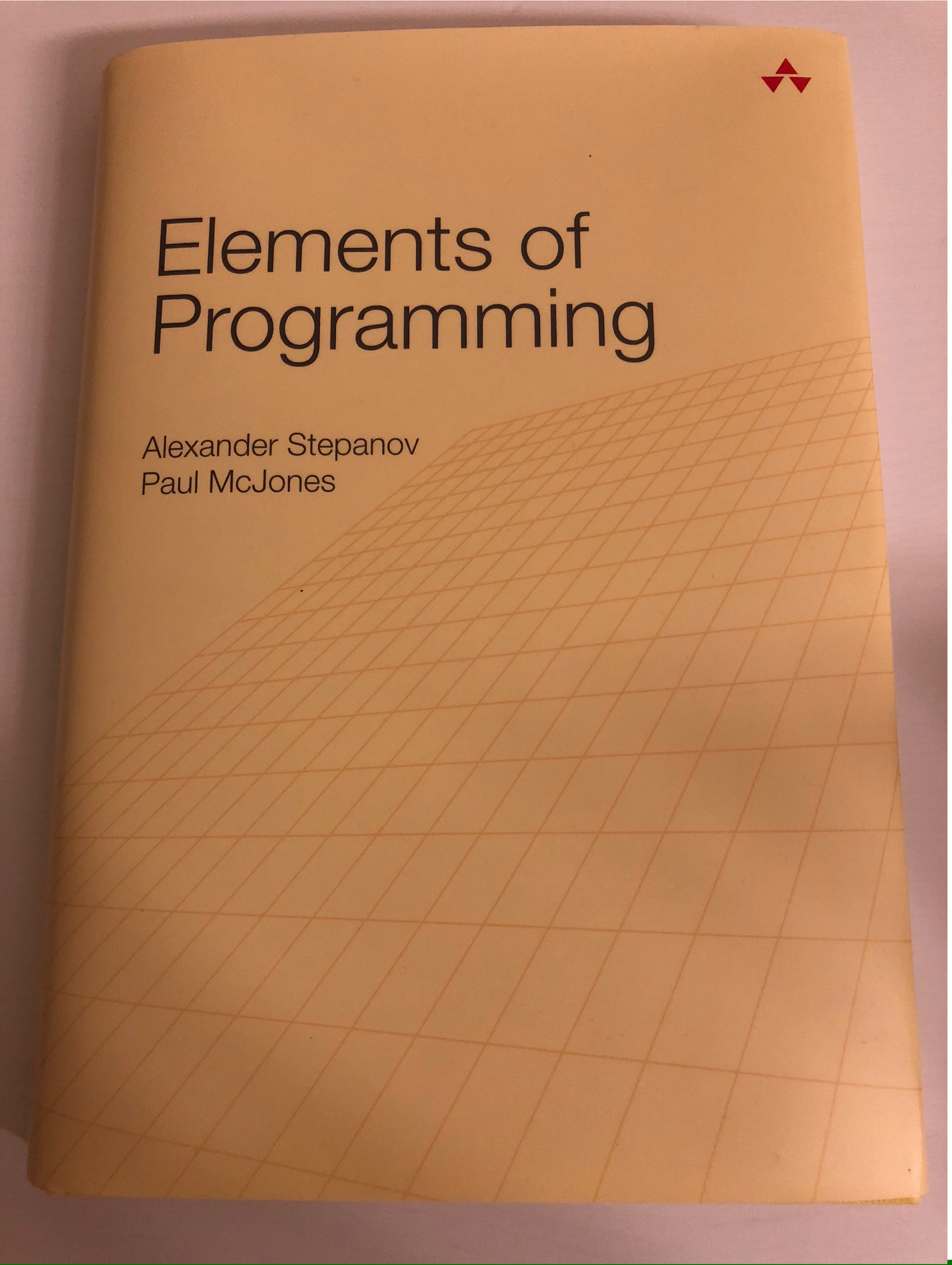
#EoP

`#define`

Concept

A **concept** is a collection of similar types.

#EoP

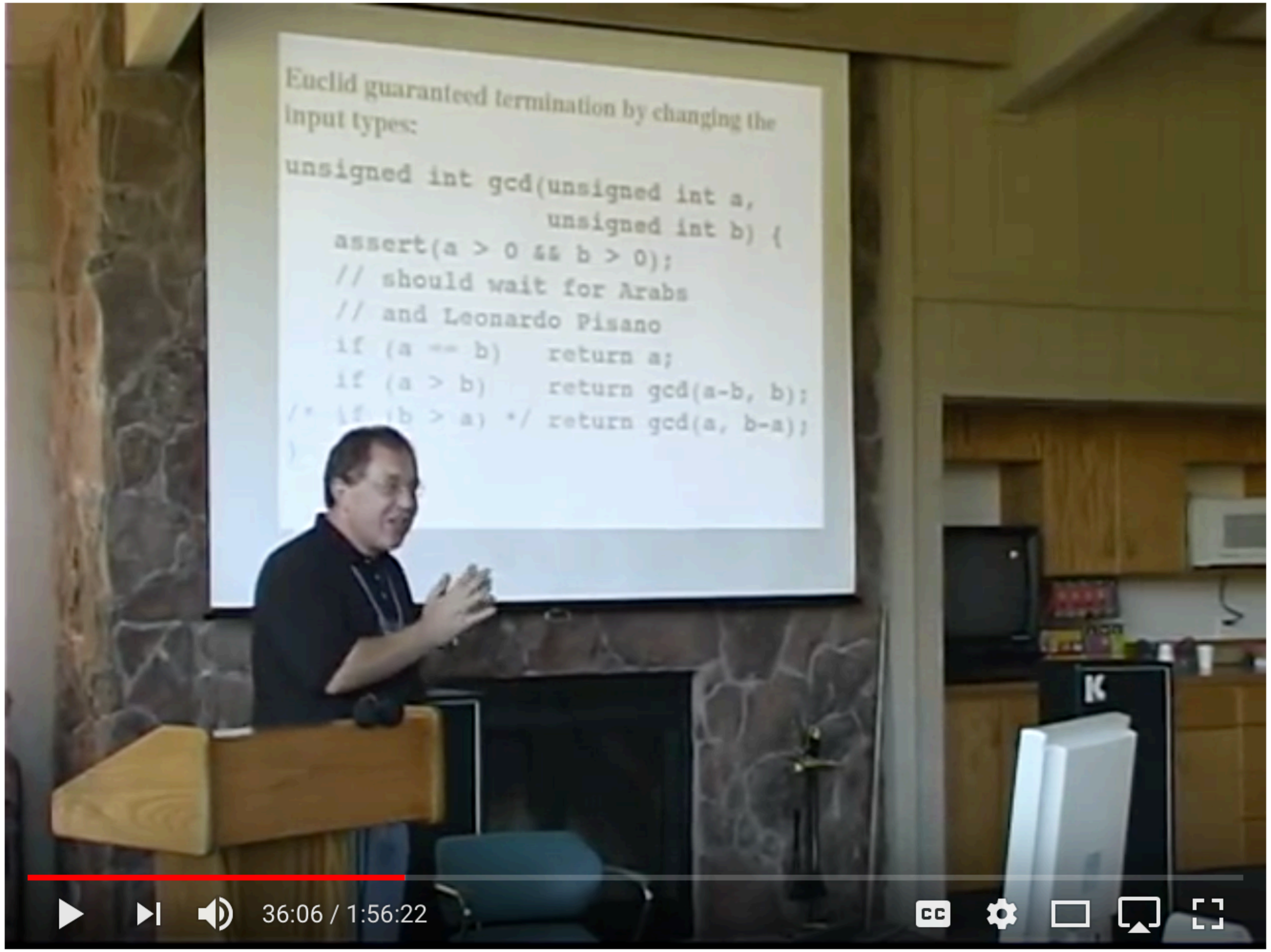


- **Foundations**
- Transformations and Their Orbits
- Associative Operations
- **Linear Orderings**
- **Ordered Algebraic Structures**
- Iterators
- Coordinate Structures
- Coordinates with Mutable Successors
- Copying
- Rearrangements
- Partition and Merging
- Composite Objects

Free PDF

elementsofprogramming.com

Mathematics Really Does Matter



Greatest Common Measure: The Last 2500 Years

GCD

One simple algorithm,
refined and improved
over 2,500 years,
while advancing
human understanding
of mathematics

SmartFriends U

September 27, 2003

https://www.youtube.com/watch?v=fanm5y00joc

Hold on !

*"I've been programming for over N years, and I've never needed any **math** to do it. I'll be just fine, thank you."*

The reason things **just worked** for you

is that <u>other</u> people thought long and hard

about the details of the type system

and the libraries you are using

... such that it feels **natural** and **intuitive** to you

4,000 years of mathematics

It all leads up to...

Fundamentals of Generic Programming

http://stepanovpapers.com/DeSt98.pdf

James C. Dehnert and Alexander Stepanov
1998

" Generic programming depends on the *decomposition* of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined **interfaces**.

Fundamentals of Generic Programming

http://stepanovpapers.com/DeSt98.pdf

James C. Dehnert and Alexander Stepanov
1998

" Among the *interfaces* of interest, the most *pervasively* and *unconsciously used*, are the fundamental operators *common* to all C++ **built-in types**, as extended to **user-defined types**, eg. *copy constructors*, *assignment*, and *equality*.

Fundamentals of Generic Programming

http://stepanovpapers.com/DeSt98.pdf

James C. Dehnert and Alexander Stepanov
1998

" We must investigate the ***relations*** which must hold among these operators to preserve **consistency** with their semantics for the built-in types and with the ***expectations*** *of programmers*.

Fundamentals of Generic Programming

http://stepanovpapers.com/DeSt98.pdf

James C. Dehnert and Alexander Stepanov
1998

We can produce an axiomatization of these operators which:

- yields the required **consistency** with built-in types
- matches the **intuitive** expectations of programmers
- reflects our underlying mathematical **expectations**

Fundamentals of Generic Programming

http://stepanovpapers.com/DeSt98.pdf

James C. Dehnert and Alexander Stepanov
1998

In other words:

We want a foundation powerful enough to support any sophisticated programming tasks, but **simple** and **intuitive** to reason about.

Fundamentals of Generic Programming

Is simplicity a good goal ?

We're C++ programmers, are we not ?

Fundamentals of Generic Programming

Is simplicity a good goal ?

I hate it when C++ programmers brag about being able to reason about some obscure language construct, proud as if they just discovered some new physical law

:(

Revisiting Regular Types

abseil.io/blog/20180531-regular-types

Titus Winters, 2018

This essay is both the best up to date synthesis of the original **Stepanov** paper,

as well as an investigation on using *non-values* as if they were `Regular` types.

Revisiting Regular Types

abseil.io/blog/20180531-regular-types

Titus Winters, 2018

This essay is both the best up to date synthesis of the original **Stepanov** paper,

as well as an investigation on using *non-values* as if they were `Regular` types.

This analysis provides us some basis to evaluate *non-owning reference parameters types* (like `string_view` and `span`) in a practical fashion,

without discarding `Regular` design.

Let's go back to the roots...

STL and Its Design Principles

STL and Its Design Principles

Talk presented at Adobe Systems Inc.
January 30, 2002

stepanovpapers.com/stl.pdf

Alexander Stepanov: STL and Its Design Principles youtube.com/watch?v=COuHLky7E2Q

STL and Its Design Principles

Fundamental Principles

- Systematically **identifying** and organizing useful **algorithms** and **data structures**
- Finding the most **general** representations of algorithms
- Using **whole-part value semantics** for data structures
- Using abstractions of addresses (**iterators**) as the interface between algorithms and data structures

STL and Its Design Principles

- algorithms are associated with a set of ***common properties***

  Eg. `{ +, *, min, max }` => **associative** operations

  => **reorder** operands

  => parallelize + reduction

  ```
  C++98        std::accumulate()
  C++17        std::transform_reduce()
  ```

- natural extension of 4,000 years of mathematics

- exists a generic algorithm behind every `while()` or `for()` loop

STL and Its Design Principles

STL data structures

- STL data structures extend the semantics of C structures
- two objects never intersect (they are separate entities)
- two objects have separate lifetimes

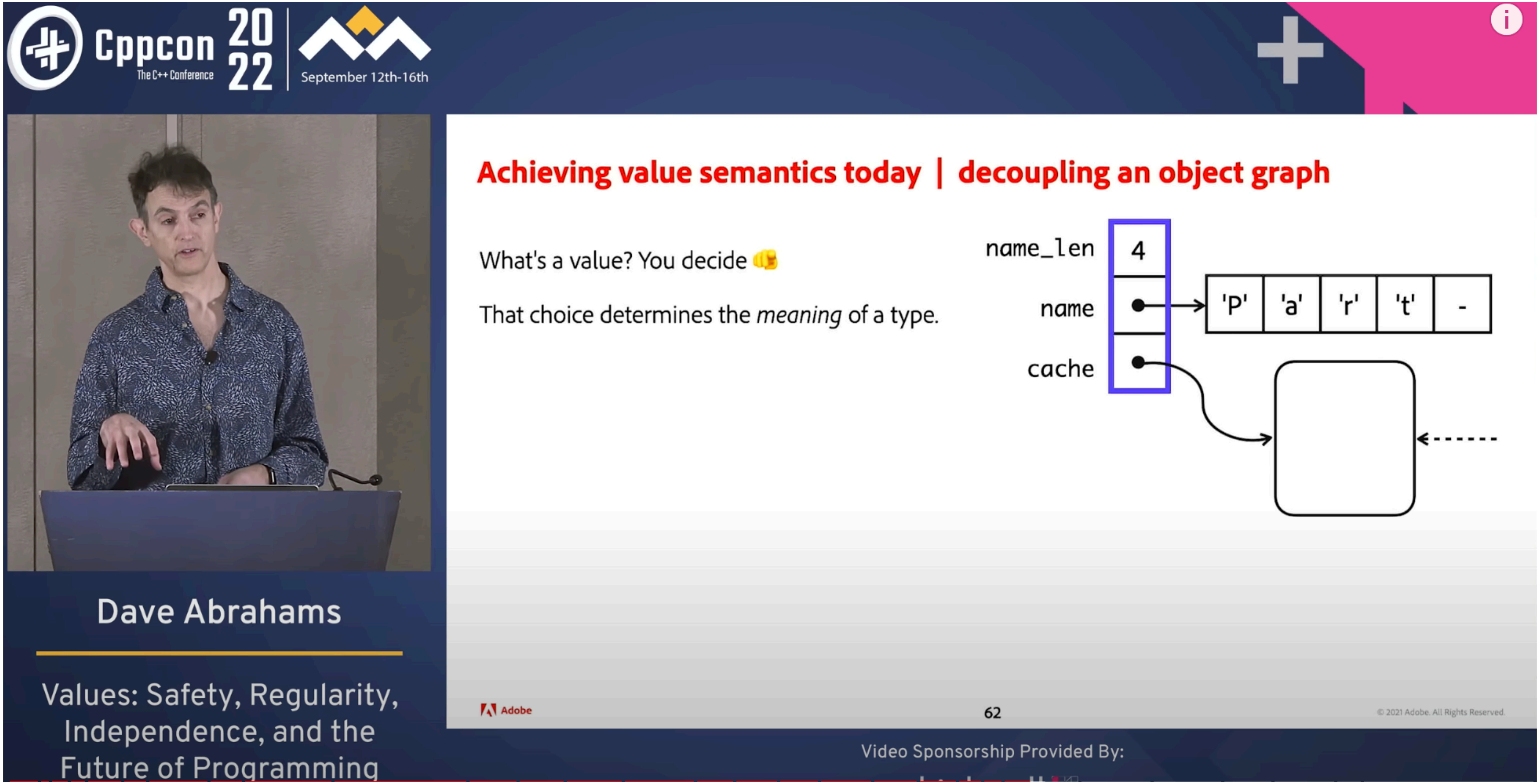
STL and Its Design Principles

STL data structures have whole-part semantics

- copy of the whole, copies the parts
- when the whole is destroyed, all the parts are destroyed
- two things are equal when they have the same number of parts

  and their corresponding parts are equal

whole-part semantics



youtube.com/watch?v=QthAU-t3PQ4

STL and Its Design Principles

Generic Programming Drawbacks

- abstraction penalty (rarely)
- implementation in the interface
- early binding
- horrible error messages (only in 99% of the cases 😄)
- duck typing
- algorithm could work on some data types, but fail to work/compile on some other new data structures

👉 We need to fully specify **requirements** on algorithm types.

Named Requirements

Examples from STL

`DefaultConstructible, MoveConstructible, CopyConstructible`

`MoveAssignable, CopyAssignable, Swappable`

`Destructible`

`EqualityComparable, LessThanComparable`

`Predicate, BinaryPredicate`

`Compare`

`FunctionObject`

`Container, SequenceContainer, ContiguousContainer, AssociativeContainer`

`InputIterator, OutputIterator`

`ForwardIterator, BidirectionalIterator, RandomAccessIterator`

cppreference.com/w/cpp/named_req

Named Requirements

Named requirements are used in the normative text of the C++ standard to define the **expectations** of the standard library.

Some* of these requirements have been formalized in **C++20** using **concepts**.

C++20 Concepts

Core language concepts

Defined in header <concepts>

- same_as (C++20)
- derived_from (C++20)
- convertible_to (C++20)
- common_reference_with (C++20)
- common_with (C++20)
- integral (C++20)
- signed_integral (C++20)
- unsigned_integral (C++20)
- floating_point (C++20)
- assignable_from (C++20)
- swappable, swappable_with (C++20)
- destructible (C++20)
- constructible_from (C++20)
- default_initializable (C++20)
- move_constructible (C++20)
- copy_constructible (C++20)

Comparison concepts

Defined in header <concepts>

- *boolean-testable* (C++20)
- equality_comparable, equality_comparable_with (C++20)
- totally_ordered, totally_ordered_with (C++20)

Defined in header <compare>

- three_way_comparable, three_way_comparable_with (C++20)

Object concepts

Defined in header <concepts>

- movable (C++20)
- copyable (C++20)
- semiregular (C++20)
- regular (C++20)

Callable concepts

Defined in header <concepts>

- invocable, regular_invocable (C++20)
- predicate (C++20)
- relation (C++20)
- equivalence_relation (C++20)
- strict_weak_order (C++20)

+ concepts in the iterators library, algorithms library, ranges library   cppreference.com/w/cpp/concepts

What is a `Concept`, anyway ?

Formal specification of concepts makes it possible to **verify** that template arguments satisfy the **expectations** of a template or function during overload resolution and template specialization (requirements).

Each concept is a **predicate**, evaluated at *compile time*, and becomes a part of the ***interface*** of a template where it is used as a constraint.

cppreference.com/w/cpp/language/constraints

What's the Practical Upside ?

If I'm not a library writer 🤓,
Why Do I Care ?

What's the Practical Upside ?

Using STL algorithms & data structures

What's the Practical Upside ?

Using STL algorithms & data structures

Designing & exposing your own vocabulary types (interfaces, APIs)

Using STL - `Compare` Requirements

Eg.

```
template<class RandomIt, class Compare>
constexpr void std::sort(RandomIt first, RandomIt last, Compare comp);
```

cppreference.com/w/cpp/named_req/Compare

Using STL - Compare Requirements

Eg.

```
template<class RandomIt, class Compare>
constexpr void std::sort(RandomIt first, RandomIt last, Compare comp);
```

What are the requirements for a Compare type ?

cppreference.com/w/cpp/named_req/Compare

Using STL - Compare Requirements

Eg.

```
template<class RandomIt, class Compare>
constexpr void std::sort(RandomIt first, RandomIt last, Compare comp);
```

What are the requirements for a Compare type ?

`Compare << BinaryPredicate << Predicate << FunctionObject << Callable`

cppreference.com/w/cpp/named_req/Compare

Using STL - Compare Requirements

Eg.

```
template<class RandomIt, class Compare>
constexpr void std::sort(RandomIt first, RandomIt last, Compare comp);
```

What are the requirements for a Compare type ?

Compare << BinaryPredicate << Predicate << FunctionObject << Callable

```
bool comp(*iter1, *iter2);
```

cppreference.com/w/cpp/named_req/Compare

Using STL - Compare Requirements

Eg.

```
template<class RandomIt, class Compare>
constexpr void std::sort(RandomIt first, RandomIt last, Compare comp);
```

What are the requirements for a `Compare` type ?

`Compare << BinaryPredicate << Predicate << FunctionObject << Callable`

```
bool comp(*iter1, *iter2);
```

But what kind of *ordering* relationship is needed for the *elements* of the collection ?

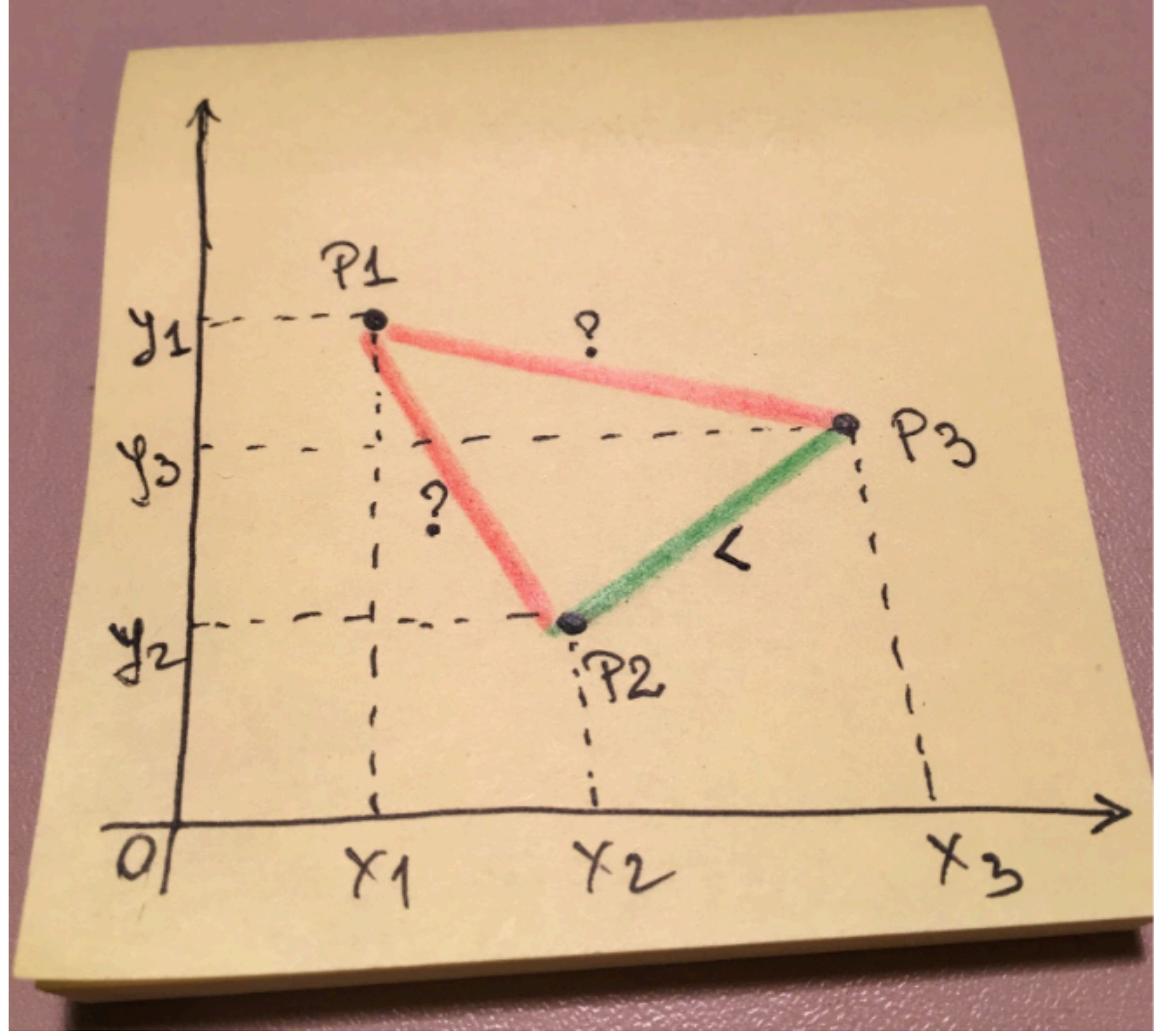
🤔

cppreference.com/w/cpp/named_req/Compare

`Compare` Requirements

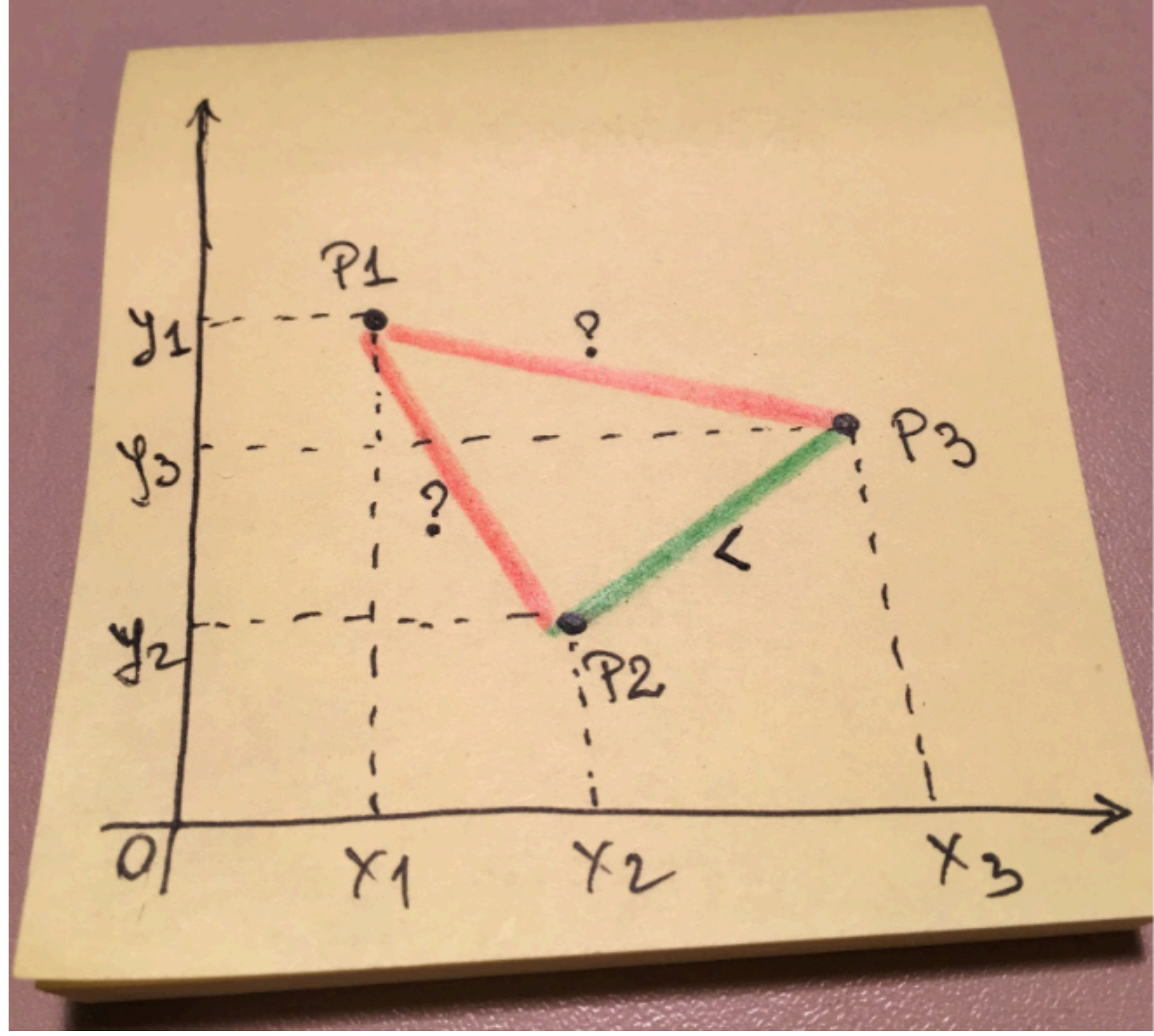
Partial ordering **relationship is not enough** 🤔

`Compare` **needs a *stronger* constraint**



🔥 `(a.x < b.x) && (a.y < b.y)`

`Compare` Requirements



Partial ordering **relationship is not enough** 🤔

`Compare` **needs a *stronger* constraint**

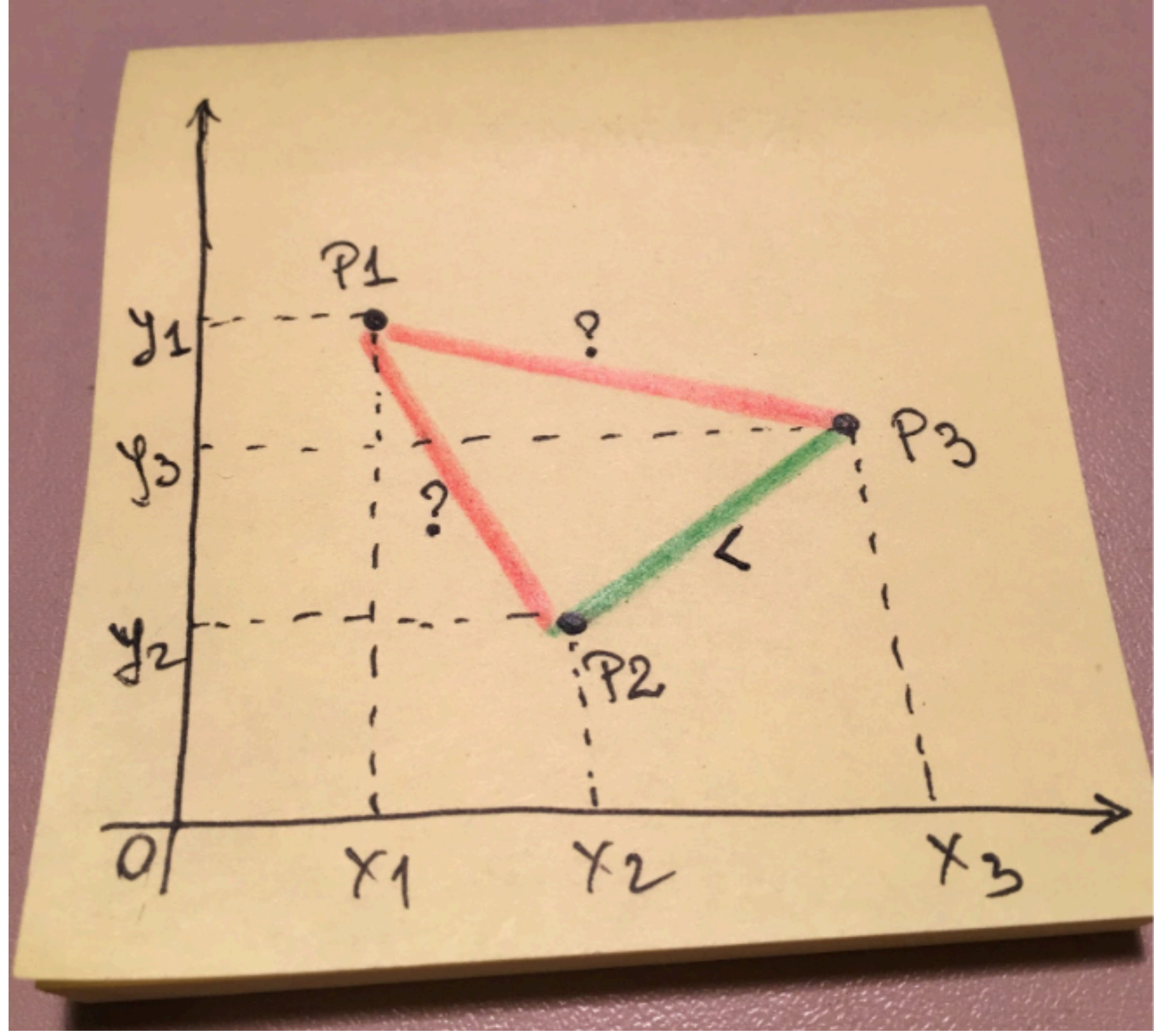
🔥 `(a.x < b.x) && (a.y < b.y)`

Strict weak ordering = Partial ordering + *Transitivity of Equivalence*

`Compare` Requirements

Partial ordering **relationship is not enough** 🤔

`Compare` **needs a *stronger* constraint**



🔥 `(a.x < b.x) && (a.y < b.y)`

Strict weak ordering = Partial ordering + *Transitivity of Equivalence*

where:

```
equiv(a,b) : comp(a,b)==false && comp(b,a)==false
```

Strict weak ordering

wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

Irreflexivity	`∀ a, comp(a,a)==false`
Antisymmetry	`∀ a, b, if comp(a,b)==true => comp(b,a)==false`
Transitivity	`∀ a, b, c, if comp(a,b)==true and comp(b,c)==true => comp(a,c)==true`
Transitivity of equivalence	`∀ a, b, c, if equiv(a,b)==true and equiv(b,c)==true => equiv(a,c)==true`

where:

```
equiv(a,b) : comp(a,b)==false && comp(b,a)==false
```

Concept: *Strict weak ordering*

std::strict_weak_order

Defined in header <concepts>

```
template< class R, class T, class U >
concept strict_weak_order = std::relation<R, T, U>;
```

(since C++20)

The concept `strict_weak_order<R, T, U>` specifies that the `relation` R imposes a strict weak ordering on its arguments.

Semantic requirements

A relation `r` is a strict weak ordering if

- it is irreflexive: for all `x`, `r(x, x)` is `false`;
- it is transitive: for all `a`, `b` and `c`, if `r(a, b)` and `r(b, c)` are both `true` then `r(a, c)` is `true`;
- let `e(a, b)` be `!r(a, b) && !r(b, a)`, then `e` is transitive: `e(a, b) && e(b, c)` implies `e(a, c)`.

Under these conditions, it can be shown that `e` is an equivalence relation, and `r` induces a strict total ordering on the equivalence classes determined by `e`.

cppreference.com/w/cpp/concepts/strict_weak_order

< `LessThanComparable`

cppreference.com/w/cpp/named_req/LessThanComparable

Irreflexivity	`∀ a, (a < a)==false`
Antisymmetry	`∀ a, b, if (a < b)==true => (b < a)==false`
Transitivity	`∀ a, b, c, if (a < b)==true and (b < c)==true => (a < c)==true`
Transitivity of equivalence	`∀ a, b, c, if equiv(a,b)==true and equiv(b,c)==true => equiv(a,c)==true`

where:

```
equiv(a,b) : (a < b)==false && (b < a)==false
```

Named Requirements

Examples from STL

wg21.link/p0898

DefaultConstructible, MoveConstructible, CopyConstructible

MoveAssignable, CopyAssignable, Swappable

Destructible

< LessThanComparable, EqualityComparable

Predicate, BinaryPredicate

Compare

FunctionObject

Container, SequenceContainer, ContiguousContainer, AssociativeContainer

InputIterator, OutputIterator

ForwardIterator, BidirectionalIterator, RandomAccessIterator

cppreference.com/w/cpp/named_req

EqualityComparable

cppreference.com/w/cpp/named_req/EqualityComparable

Reflexivity	`∀ a, (a == a)==true`
Symmetry	`∀ a, b, if (a == b)==true => (b == a)==true`
Transitivity	`∀ a, b, c, if (a == b)==true and (b == c)==true => (a == c)==true`

The type must work with `operator==` and the result should have ***standard semantics***.

wikipedia.org/wiki/Equivalence_relation

Concept: EqualityComparable

cppreference.com/w/cpp/concepts/equality_comparable

```
template< class T, class U >
concept __WeaklyEqualityComparableWith =
  requires(const std::remove_reference_t<T>& t,
           const std::remove_reference_t<U>& u) {
    { t == u } -> boolean-testable;
    { t != u } -> boolean-testable;
    { u == t } -> boolean-testable;
    { u != t } -> boolean-testable;
  };

template< class T >
concept equality_comparable = __WeaklyEqualityComparableWith<T, T>;
```

wikipedia.org/wiki/Equivalence_relation

Equality vs. Equivalence

For the types that are both `EqualityComparable` and `LessThanComparable`,

the STL makes a clear **distinction** between **equality** and **equivalence**

where:

```
equal(a,b) : (a == b)
equiv(a,b) : (a < b)==false && (b < a)==false
```

Equality is a special case of **equivalence**

Equality vs. Equivalence

For the types that are both `EqualityComparable` and `LessThanComparable`,

the STL makes a clear **distinction** between **equality** and **equivalence**

where:

```
equal(a,b) : (a == b)
equiv(a,b) : (a < b)==false && (b < a)==false
```

Equality is a special case of **equivalence**

Equality is both an *equivalence relation* and a *partial order*.

Total ordering relationship

Total ordering relationship

`comp()` induces a ***strict total ordering***
on the equivalence classes determined by `equiv()`

Total ordering relationship

`comp()` induces a ***strict total ordering***
on the equivalence classes determined by `equiv()`

The equivalence relation and its equivalence classes
partition the elements of the set,
and are **totally ordered** by **<**

< LessThanComparable

cppreference.com/w/cpp/named_req/LessThanComparable

```
template< class T, class U >
concept __PartiallyOrderedWith =
  requires(const std::remove_reference_t<T>& t,
           const std::remove_reference_t<U>& u) {
    { t <  u } -> boolean-testable;
    { t >  u } -> boolean-testable;
    { t <= u } -> boolean-testable;
    { t >= u } -> boolean-testable;
    { u <  t } -> boolean-testable;
    { u >  t } -> boolean-testable;
    { u <= t } -> boolean-testable;
    { u >= t } -> boolean-testable;
  };

template< class T >
concept totally_ordered = std::equality_comparable<T> &&
                          __PartiallyOrderedWith<T, T>;
```

A Concept Design for the STL

Palo Alto TR
open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

A. Stepanov et al.

STL assumes **equality** is always defined (or at least, equivalence relation)

STL algorithms assume `Regular` data structures

The STL was written with *Regularity* as its basis

wg21.link/p0898

`#define`

`SemiRegular`

`DefaultConstructible, MoveConstructible, CopyConstructible`

`MoveAssignable, CopyAssignable, Swappable`

`Destructible`

#define

SemiRegular

```
template <class T>
concept semiregular = std::copyable<T> &&
                      std::default_initializable<T>;
```

```
template <class T>
concept copyable =
  std::copy_constructible<T> &&
  std::movable<T> &&
  std::assignable_from<T&, T&> &&
  std::assignable_from<T&, const T&> &&
  std::assignable_from<T&, const T>;
```

```
template<class T>
concept default_initializable =
    std::constructible_from<T> &&
    requires { T{}; } &&
    requires { ::new T; };
```

cppreference.com/w/cpp/concepts/semiregular

#define

Regular

(aka "Stepanov Regular")

```
SemiRegular {
        DefaultConstructible, MoveConstructible, CopyConstructible
        MoveAssignable, CopyAssignable, Swappable
        Destructible
}

        +

        EqualityComparable
```

#define

Regular

```
template <class T>
concept regular = std::semiregular<T> &&
                  std::equality_comparable<T>;

template< class T, class U >
concept __WeaklyEqualityComparableWith =
  requires(const std::remove_reference_t<T>& t,
           const std::remove_reference_t<U>& u) {
    { t == u } -> boolean-testable;
    { t != u } -> boolean-testable;
    { u == t } -> boolean-testable;
    { u != t } -> boolean-testable;
  };

template< class T >
concept equality_comparable = __WeaklyEqualityComparableWith<T, T>;
```

cppreference.com/w/cpp/concepts/regular

Equality

Defining **equality** is hard 🤕

Equality

Ultimately, **Stepanov** proposes the following *definition*:

> " Two objects are **equal** if their corresponding *parts* are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

😓

stepanovpapers.com/DeSt98.pdf

Equality

“although it still leaves room for judgement”

Ultimately, **Stepanov** proposes the following *definition*:

" Two objects are **equal** if their corresponding *parts* are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

😓

stepanovpapers.com/DeSt98.pdf

C++20 Three-way comparison

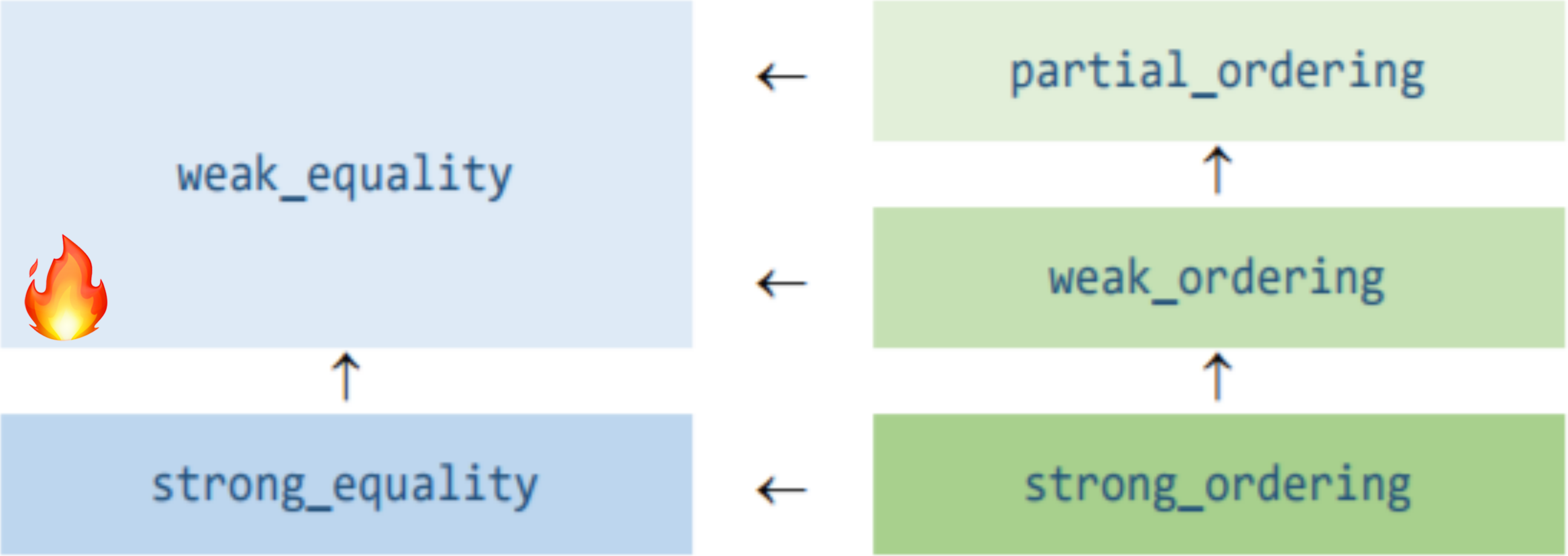
Bringing consistent comparison operations…

operator <=>

```
(a <=> b) <  0  if  a < b
(a <=> b) >  0  if  a > b
(a <=> b) == 0  if  a and b are equal/equivalent
```

C++20 Three-way comparison

The comparison categories for: `operator <=>`



It's all about *relation strength*

C++20 Three-way comparison

<, <=, >, >= synthesized from operator<=>
!= synthesized from operator==

operator<=>

operator!=

operator==

Convenience

Efficiency ?

The problem: implement <=> optimally for "wrapper" types

```
struct S {
  vector<string> names;
  auto operator<=>(S const&) const = default;
};
```

wg21.link/P1185

operator<=>

```
template<typename T>
strong_ordering operator<=>(vector<T> const& lhs, vector<T> const& rhs)
{
    size_t min_size = min(lhs.size(), rhs.size());

    for (size_t i = 0; i != min_size; ++i)
    {
        if (auto const cmp = compare_3way(lhs[i], rhs[i]); cmp != 0) {
            return cmp;
        }
    }

    return lhs.size() <=> rhs.size();
}
```

operator<=>

```
template<typename T>
bool operator==(vector<T> const& lhs, vector<T> const& rhs)
{
    // short-circuit on size early
    const size_t size = lhs.size();
    if (size != rhs.size()) {
        return false;
    }

    for (size_t i = 0; i != size; ++i) {
        // use ==, not <=>, in all nested comparisons
        if (lhs[i] != rhs[i]) {
            return false;
        }
    }

    return true;
}
```

Rust

Rust deals in Traits (which are roughly analogous to C++0x concepts and Swift protocols). It has four relevant traits that have to do with *comparisons*:

- PartialEq (which is a partial equivalence relation spelled which only requires symmetry and transitivity)
- Eq (which extends PartialEq, adding reflexivity)
- PartialOrd (which allows for incomparability by returning Option<Ordering>, where Ordering is an enum)
- Ord (a total order, which extends Eq and PartialOrd)

Rust

`Eq`

```rust
impl<A, B> SlicePartialEq<B> for [A]
    where A: PartialEq<B>
{
    default fn eq(&self, other: &[B]) -> bool {
        if self.len() != other.len() {
            return false;
        }

        for i in 0..self.len() {
            if !self[i].eq(&other[i]) {
                return false;
            }
        }

        true
    }
}
```

`Ord`

```rust
impl<A> SliceOrd<A> for [A]
    where A: Ord
{
    default fn cmp(&self, other: &[A]) -> Ordering {
        let l = cmp::min(self.len(), other.len());

        let lhs = &self[..l];
        let rhs = &other[..l];

        for i in 0..l {
            match lhs[i].cmp(&rhs[i]) {
                Ordering::Equal => (),
                non_eq => return non_eq,
            }
        }

        self.len().cmp(&other.len())
    }
}
```

Let's spend a few minutes on an interesting C++17 type

`std::optional<T>`

Any time you need to express:

- *value or not value*
- *possibly an answer*
- *object with delayed initialization*

Using a common **vocabulary type** for these cases raises the *level of abstraction*, making it easier for others to understand what your code is doing.

std::optional<T>

optional<T> extends T's ordering operations:

< > <= >= == !=

an empty optional compares as **less than** any optional that *contains* a T

=> you can use it in some contexts exactly *as if* it *were a* T

std::optional<T>

Using std::optional as *vocabulary type* allows us to simplify code and compose functions easily.

Write waaaaay less error checking code

But, wait...

```
std::optional<T&>
```

😱

```
operator=
operator==
```

std::optional<T&>

- **References for Standard Library Vocabulary Types - an optional<> case study**

  wg21.link/p1683

- **To Bind and Loose a Reference**

  thephd.dev/to-bind-and-loose-a-reference-optional

📝 Recommendation:

- **rebinding**
- **shallow const**
- **deep comparison**

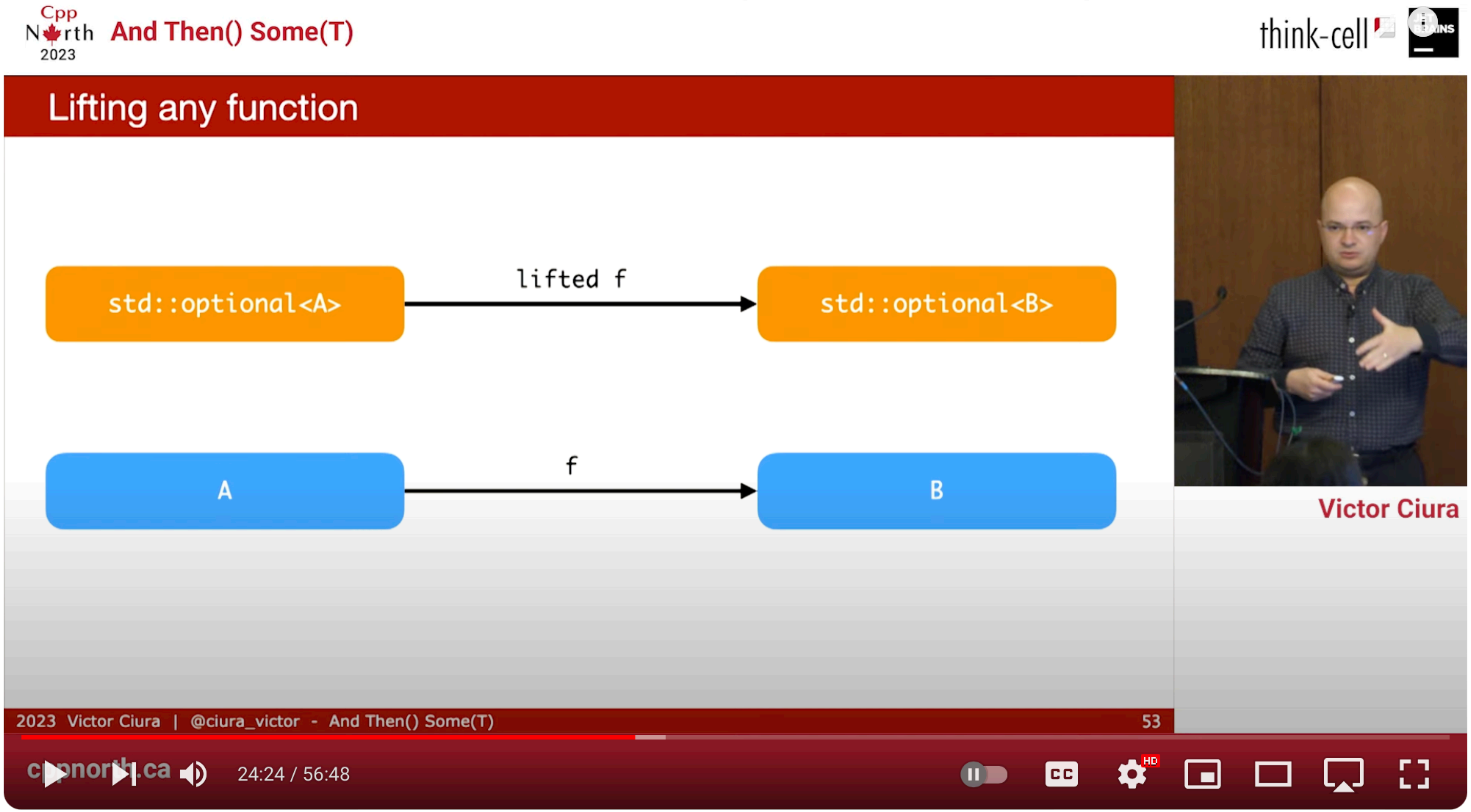
👉 rebinding optional reference

This is the solution that is seen as a step up from the conservative solution.

It is the version used in boost::optional for over 15 years + many other implementations.

std::optional<T&>

youtube.com/watch?v=6c7pZYP_iIE



And Then() Some(T) Functional Adventures With C++23 std::optional and std::expected - Victor Ciura

youtube.com/watch?v=06VNq_tC-l0

C++17

`std::string_view`

An object that can refer to a **constant**

contiguous sequence of `char`-like objects

A `string_view` does not manage the **storage** that it refers to

Lifetime management is up to the user

`std::string_view` **is a borrow type**

⚠️ `string_view` succeeds admirably in the goal of "*drop-in replacement*" for `const string &` parameters.

The problem:

The two relatively **old** kinds of types are **object types** and **value types**

The new kid on the block is the ***borrow type***

`string_view` was our first "mainstream" ***borrow type***

Borrow types are essentially "*borrowed*" references to existing objects

Borrow types are essentially "*borrowed*" references to existing objects

- they ***lack ownership***

Borrow types are essentially "*borrowed*" references to existing objects

- they ***lack ownership***
- they are ***short-lived***

Borrow types are essentially "*borrowed*" references to existing objects

- they ***lack ownership***
- they are ***short-lived***
- they generally can do without an ***assignment operator***

Borrow types are essentially “*borrowed*” references to existing objects

- they ***lack ownership***
- they are ***short-lived***
- they generally can do without an ***assignment operator***
- they generally appear only in ***function parameter*** lists

Borrow types are essentially "*borrowed*" references to existing objects

- they ***lack ownership***
- they are ***short-lived***
- they generally can do without an ***assignment operator***
- they generally appear only in ***function parameter*** lists
- they generally ***cannot be stored in data structures*** or ***returned*** <u>safely</u> from functions (no ownership semantics)

`std::string_view` is a borrow type

⚠️

`string_view` is ***assignable***: `sv1 = sv2`

Assignment has ***shallow*** semantics (of course, the viewed strings are ***immutable***)

Meanwhile, the comparison `sv1 == sv2` has ***deep*** semantics (lexicographic comp)

`std::string_view`

Non-owning reference type

When the underlying data is **extant** and **constant**

we can determine whether the rest of its usage still **looks Regular**

`std::string_view`

Non-owning reference type

When the underlying data is **extant** and **constant**

we can determine whether the rest of its usage still **looks Regular**

When used properly (eg. *function parameter*),

`string_view` works well...

as if it is a **Regular** type

C++20 `std::span<T>`

Think `"array view"` as in `std::string_view`,

but **mutable** on underlying data

https://en.cppreference.com/w/cpp/container/span

C++20 `std::span<T>`

A `std::span` does not manage the **storage** that it refers to

Lifetime management is up to the user

https://en.cppreference.com/w/cpp/container/span

WWSD

WWSD

What Would Stepanov Do?

What Would Stepanov Do?

What Would Stepanov Do?

- *overloading operators* can be dangerous when you change the ***common meaning*** of the operator

What Would Stepanov Do?

- *overloading operators* can be dangerous when you change the ***common meaning*** of the operator
- the meaning of **copy construction** and **copy assignment** is to **copy the value** of the object

What Would Stepanov Do?

- *overloading operators* can be dangerous when you change the ***common meaning*** of the operator
- the meaning of **copy construction** and **copy assignment** is to **copy the value** of the object
- the meaning of `==` and `<` is to **compare the value** of the object

What Would Stepanov Do?

- *overloading operators* can be dangerous when you change the ***common meaning*** of the operator
- the meaning of **copy construction** and **copy assignment** is to **copy the value** of the object
- the meaning of `==` and `<` is to **compare the value** of the object
- **copy**, **assignment**, **equality** are expected to go together (act as built-in types -- *intuitively*)

What Would Stepanov Do?

- *overloading operators* can be dangerous when you change the ***common meaning*** of the operator
- the meaning of **copy construction** and **copy assignment** is to **copy the value** of the object
- the meaning of `==` and `<` is to **compare the value** of the object
- **copy**, **assignment**, **equality** are expected to go together (act as built-in types -- *intuitively*)
- when designing a class type, where possible it should be a **Regular** type (see **EoP**)

What Would Stepanov Do?

- *overloading operators* can be dangerous when you change the ***common meaning*** of the operator
- the meaning of **copy construction** and **copy assignment** is to **copy the value** of the object
- the meaning of `==` and `<` is to **compare the value** of the object
- **copy**, **assignment**, **equality** are expected to go together (act as built-in types -- *intuitively*)
- when designing a class type, where possible it should be a **Regular** type (see **EoP**)

"Copy or copy not; there is no shallow"
- Master Yoda

Should Span be Regular?

wg21.link/p1085

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)
- we could make `operator==` **deep** (elements in the span are compared with `std::equal()`), just like `std::string_view`

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)
- we could make `operator==` **deep** (elements in the span are compared with `std::equal()`), just like `std::string_view`
    - however `string_view` **can't modify** the elements it points at (**const**)

      => the shallow copy of `string_view` is similar to a *copy-on-write optimization*

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)
- we could make `operator==` **deep** (elements in the span are compared with `std::equal()`), just like `std::string_view`
  - however `string_view` **can't modify** the elements it points at (**const**)
    => the shallow copy of `string_view` is similar to a *copy-on-write optimization*
  - but is span a **value** ? do we need a **deep** compare ?

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)
- we could make `operator==` **deep** (elements in the span are compared with `std::equal()`), just like `std::string_view`
  - however `string_view` **can't modify** the elements it points at (**const**)
    => the shallow copy of `string_view` is similar to a *copy-on-write optimization*
  - but is span a **value** ? do we need a **deep** compare ?
- `std::span` is trying to act like a collection of the elements over which it spans

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)
- we could make `operator==` **deep** (elements in the span are compared with `std::equal()`), just like `std::string_view`
  - however `string_view` **can't modify** the elements it points at (**const**)

    => the shallow copy of `string_view` is similar to a *copy-on-write optimization*
  - but is span a **value** ? do we need a **deep** compare ?
- `std::span` is trying to act like a collection of the elements over which it spans
  - but it's not `Regular` !

Should Span be Regular?

wg21.link/p1085

- `operator=` (copy) is **shallow** (just **pointer** and **size** are copied)
- we could make `operator==` **deep** (elements in the span are compared with `std::equal()`), just like `std::string_view`
  - however `string_view` **can't modify** the elements it points at (**const**)

    => the shallow copy of `string_view` is similar to a *copy-on-write optimization*
  - but is span a **value** ? do we need a **deep** compare ?
- `std::span` is trying to act like a collection of the elements over which it spans
  - but it's not `Regular` !
- basically `std::span` has ***reference semantics***

Should Span be Regular?

wg21.link/p1085

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in `==` and **copy**)

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in `==` and **copy**)
  - **deep** `equality` means the *value* of span are the **elements** it spans, not `{ ptr + size }`

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in `==` and **copy**)
  - **deep** `equality` means the *value* of span are the **elements** it spans, not `{ ptr + size }`
- if we want span to act like a *lightweight* representation of the elements it references:

=> we need to have a **shallow** `operator==` (*just like smart pointers*)

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in `==` and **copy**)
  - **deep** `equality` means the *value* of span are the **elements** it spans, not `{ ptr + size }`
- if we want span to act like a *lightweight* representation of the elements it references:

=> we need to have a **shallow** `operator==` (*just like smart pointers*)

  - **shallow const** => **shallow** `operator==`

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in **==** and **copy**)
  - **deep** `equality` means the *value* of span are the **elements** it spans, not `{ ptr + size }`
- if we want span to act like a *lightweight* representation of the elements it references:

  => we need to have a **shallow** `operator==` (*just like smart pointers*)
  - **shallow const** => **shallow** `operator==`
- but **shallow** `operator==` might be really confusing to users (especially because of `string_view`)

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in `==` and **copy**)
  - **deep** `equality` means the *value* of span are the **elements** it spans, not `{ ptr + size }`
- if we want span to act like a *lightweight* representation of the elements it references:

  => we need to have a **shallow** `operator==` (*just like smart pointers*)
  - **shallow** `const` => **shallow** `operator==`
- but **shallow** `operator==` might be really confusing to users (especially because of `string_view`)
- final decision was to REMOVE `operator==` completely

Should Span be Regular?

wg21.link/p1085

- **deep** `operator==` also implies **deep const** (logical const) - extend protection to all parts (**EoP**)
  - all parts of the type that constitute its **value** (eg. participate in `==` and **copy**)
  - **deep** `equality` means the *value* of span are the **elements** it spans, not `{ ptr + size }`
- if we want span to act like a *lightweight* representation of the elements it references:

  => we need to have a **shallow** `operator==` (*just like smart pointers*)
  - **shallow const** => **shallow** `operator==`
- but **shallow** `operator==` might be really confusing to users (especially because of `string_view`)
- final decision was to REMOVE `operator==` completely

A Strange Beast

`std::span` - a case of unmet expectations...

- Users of the STL can reasonably expect span to be a *drop-in replacement* for `std::vector | std::array`
- And that happens to be mostly the case…
- Until of course, you try to **copy** it or change its **value**, then it stops acting like a container **:(**

A Strange Beast

`std::span` - a case of unmet expectations...

- Users of the STL can reasonably expect span to be a *drop-in replacement* for `std::vector | std::array`
- And that happens to be mostly the case…
- Until of course, you try to **copy** it or change its **value**, then it stops acting like a container :(

`std::span` **is** ~~**Regular**~~ **SemiRegular**

C++20 `std::span<T>`

Photo credit: Corentin Jabot

cor3ntin.github.io/posts/span/

Non-owning reference types
like `string_view` or `span`

You need more **contextual** information when working on an instance of this type

Non-owning reference types like `string_view` or `span`

You need more **contextual** information when working on an instance of this type

Things to consider:

- shallow copy ?
- shallow / deep compare ?
- const / mutability ?
- `operator==`

Non-owning reference types like `string_view` or `span`

Have reference semantics,

but <u>without</u> the "magic" that can make references safer

(for example *lifetime extension*)

Lifetime

```cpp
std::string Name() {
  return std::string("some long runtime value string");
}

const string & str = Name();
std::print("{}", str);

string_view sv = Name();
std::print("{}", sv);
```

- const lvalue ref binds to rvalue and provides lifetime extension
- string_view doesn't extend the lifetime of the rvalue

👉 For short strings this issue might be hard to detect due to SSO.
Problem becomes obvious with longer dynamically allocated strings.

Call To Action

Call To Action

Make your value types **Regular**

Call To Action

Make your value types **Regular**

The best Regular types are those that model `built-ins` most closely and have no dependent preconditions.

Call To Action

Make your value types **Regular**

The best Regular types are those that model `built-ins` most closely and have no dependent preconditions.

Think `int` or `std::string` or `std::vector`

Call To Action

Call To Action

For non-owning reference types like `string_view` or `span`

Call To Action

For non-owning reference types like `string_view` or `span`

You need more **contextual** information when working on an instance of this type

Call To Action

For non-owning reference types like `string_view` or `span`

You need more **contextual** information when working on an instance of this type

Try to restrict these types to **SemiRegular** to avoid confusion for your users

Regular, Revisited

ACCU

April 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
M365 Substrate

BONUS SLIDES

Object Relocation

One particularly sensitive topic about handling C++ **values**
is that they are all conservatively considered **non-relocatable**.

https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation

Object Relocation

In contrast, a **relocatable value** would preserve its invariant, even if its bits were moved arbitrarily in memory.

For example, an `int32` is relocatable because moving its 4 bytes would preserve its actual value, so the address of that value does not matter to its integrity.

https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation

Object Relocation

facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation

Object Relocation

C++'s assumption of non-relocatable **values** hurts everybody for the benefit of a few questionable designs.

https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation

Object Relocation

Only a *minority* of objects are genuinely non-relocatable:

- objects that use internal **pointers**
- objects that need to update **observers** that store pointers to them

https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation