cppcon the c++ conference

Regular Types and Why Do I Care ?

September, 2018

CAPHYON

Victor Ciura
Technical Lead, Advanced Installer
www.advancedinstaller.com

Abstract

“Regular” is not exactly a new concept (pun intended). If we reflect back on STL and its design principles, as best described by Alexander Stepanov in his 1998 “Fundamentals of Generic Programming” paper or his lecture on this topic, from 2002, we see that regular types naturally appear as necessary foundational concepts in programming.

Why do we need to bother with such taxonomies ? Well, the STL now informally assumes such properties about the types it deals with and imposes such conceptual requirements for its data structures and algorithms to work properly. The new Concepts Lite proposal (hopefully part of C++20) is based on precisely defined foundational concepts such as Semiregular, Regular, EqualityComparable, DefaultConstructible, LessThanComparable (strict weak ordering), etc. Formal specification of concepts is an ongoing effort in the ISO C++ Committee and these STL library concepts requirements are being refined as part of Ranges TS proposal (<experimental/ranges/concepts>).

Recent STL additions such as string_view, tuple, reference_wrapper, as well as new incoming types for C++20 like std::span raise new questions regarding values types, reference types and non-owning “borrow” types.

Designing and implementing regular types is crucial in everyday programing, 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 (and other concepts) and STL containers & algorithms with examples, common pitfalls and guidance.

Who Am I ?

Advanced Installer

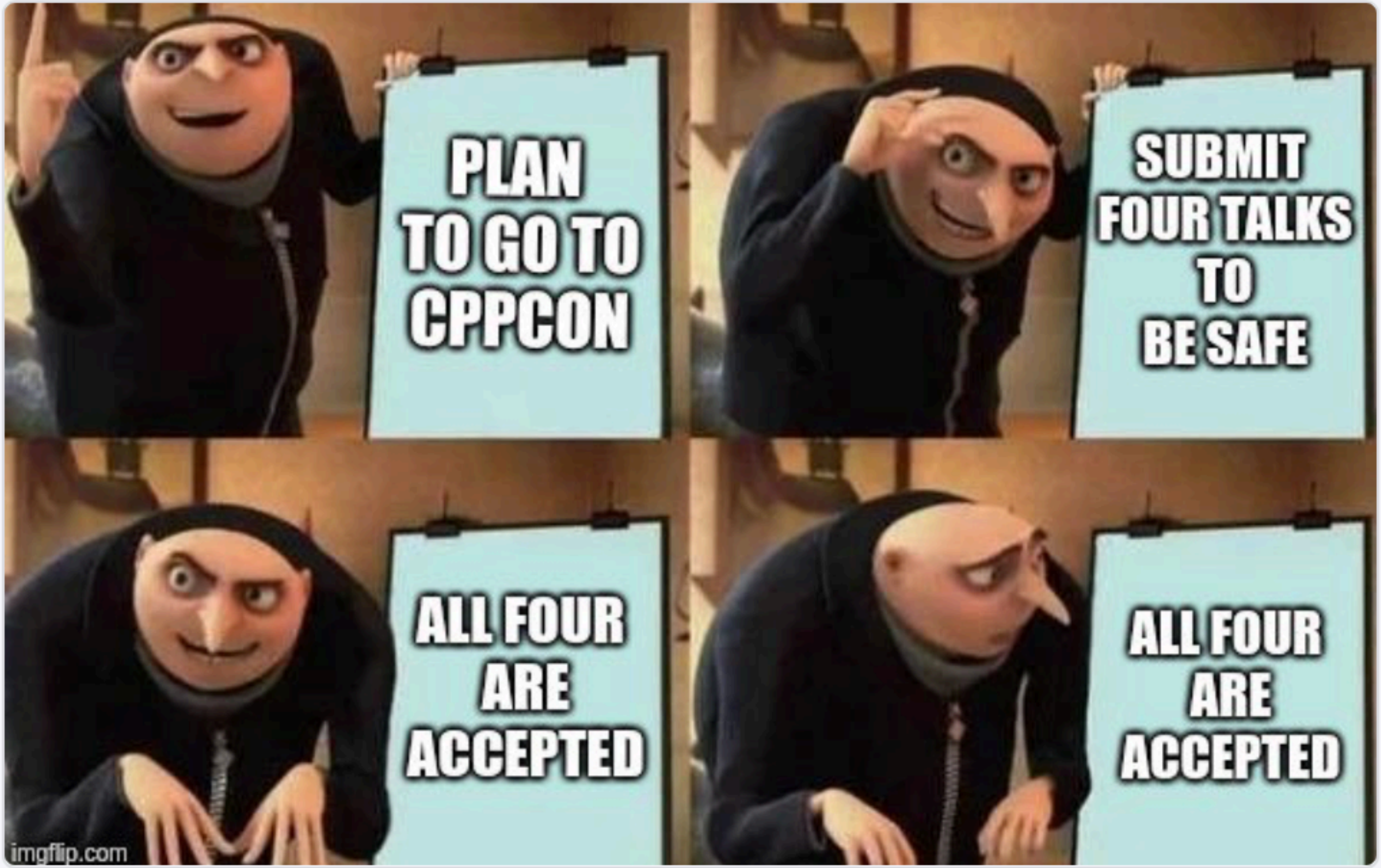
Clang Power Tools

@ciura_victor

Simon Brand @TartanLlama · Jul 6

So this happened

#CppCon




Thaaaats

Monday, September 24

11:00

Enough string_view to Hang Ourselves
Victor Ciura

Manage

Tuesday, September 25

You are here --->

09:00

Regular Types and Why Do I Care ?
Victor Ciura

Manage

Wednesday, September 26

09:00

These Aren't the COM Objects You're Looking For
Victor Ciura

Manage

Thursday, September 27

09:00

Better Tools in Your Clang Toolbox: Extending clang-tidy With Your Custom Checks
Victor Ciura

Regular Types
and
Why Do I Care ?

`Part 1 of N`

Why `Regular types` ?

Why are we talking about this ?

Have we really exhausted all the cool C++ `template<>` topics 😜 ?

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 *"Fundamentals of Generic Programming"* paper or his lecture on this topic, from 2002.

This talk is not just about `Regular types`

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.

This talk is not just about `Regular types`

`Values`

`Objects`

`Concepts`

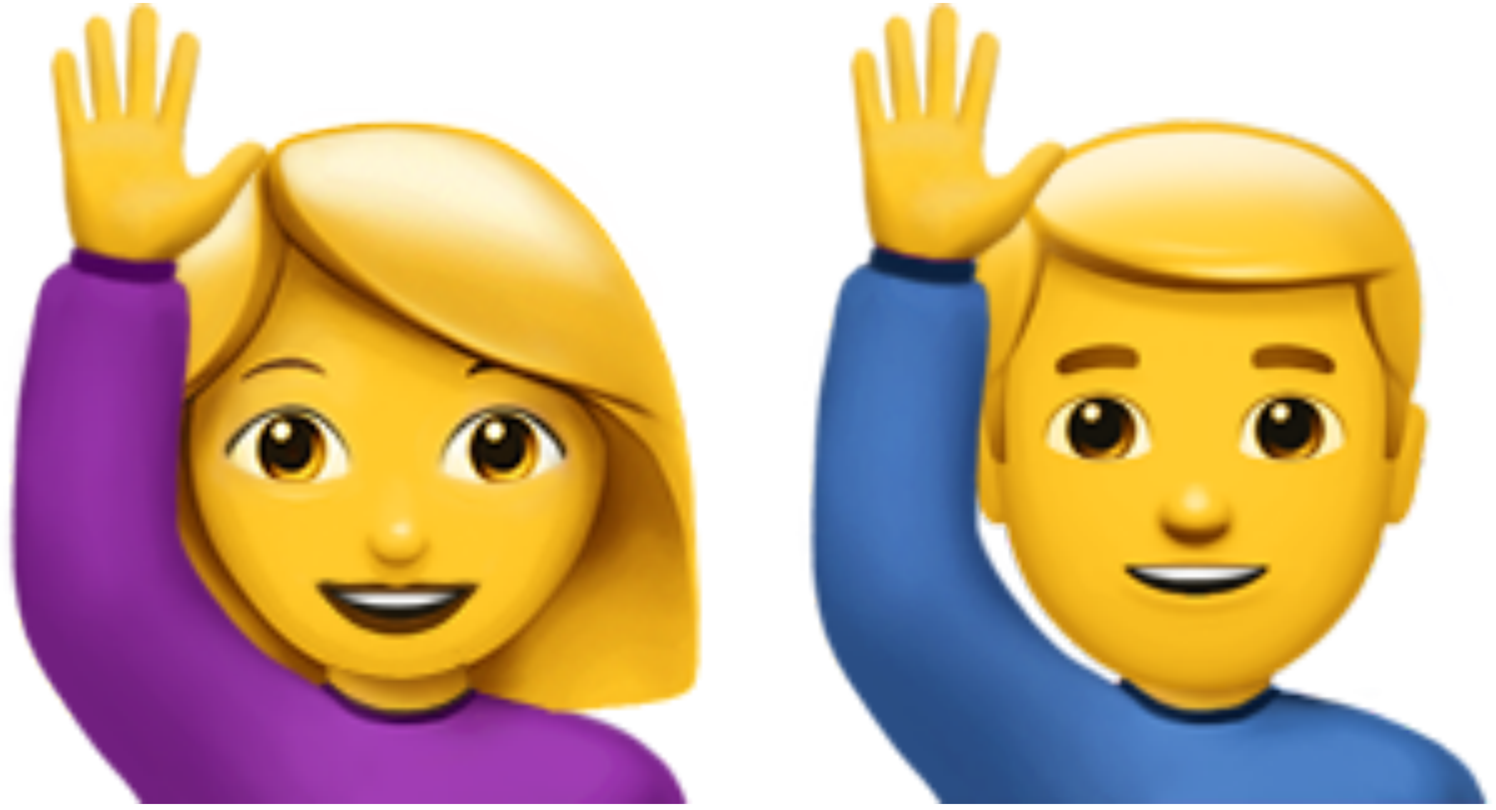
`Ordering Relations`

`Requirements`

Titus Winters

Modern C++ API Design

http://sched.co/FnLO



Titus Winters
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 ?

http://sched.co/FnLO

Let's start with the basics...

`#define`

`Datum`

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

`#define`

`Value Type`

A **value type** is a correspondence between

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

`#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.

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*.

`#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.

`#define`

Type

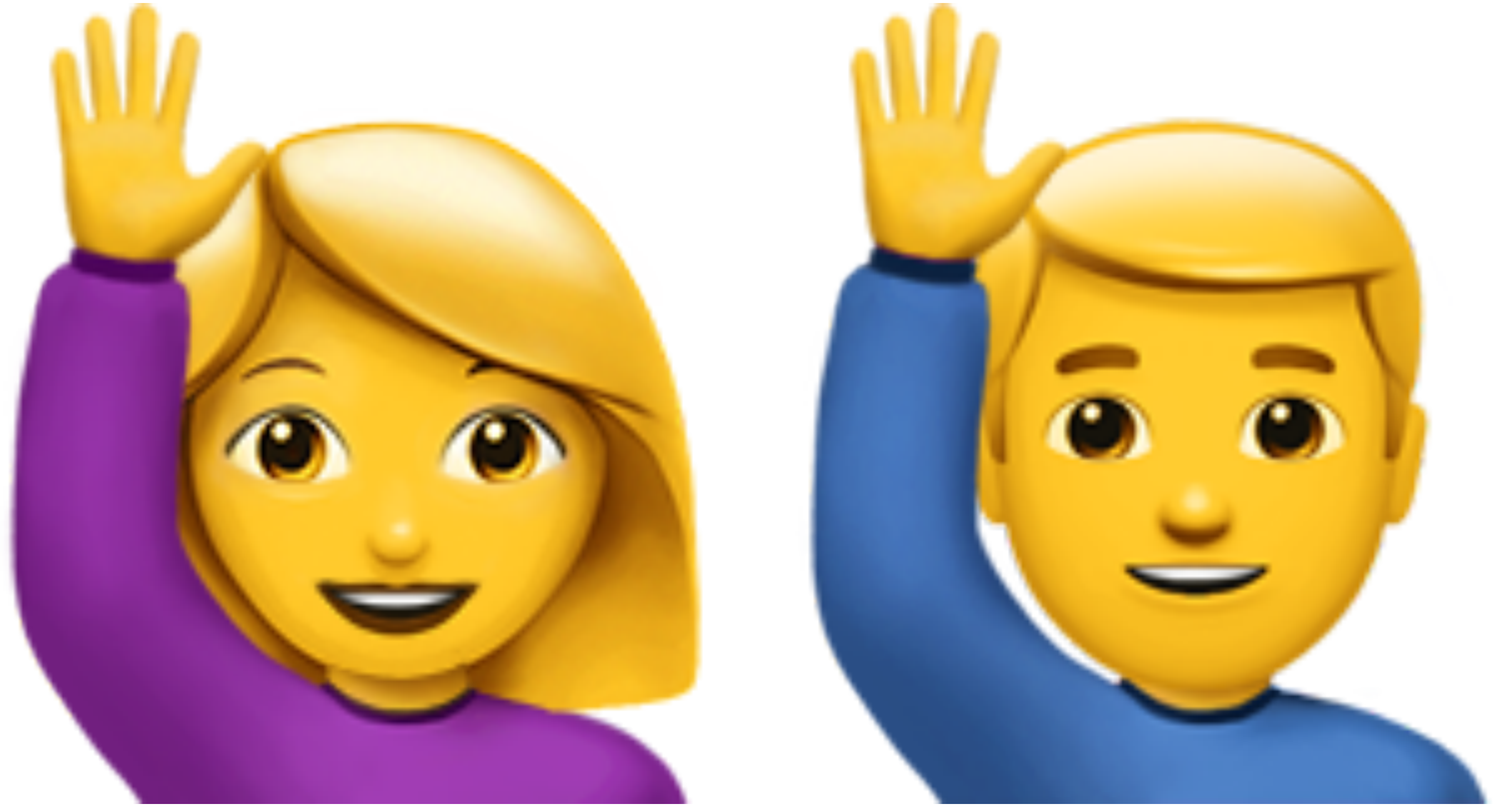
Type is a *set of values* with the same interpretation function and operations on these values.

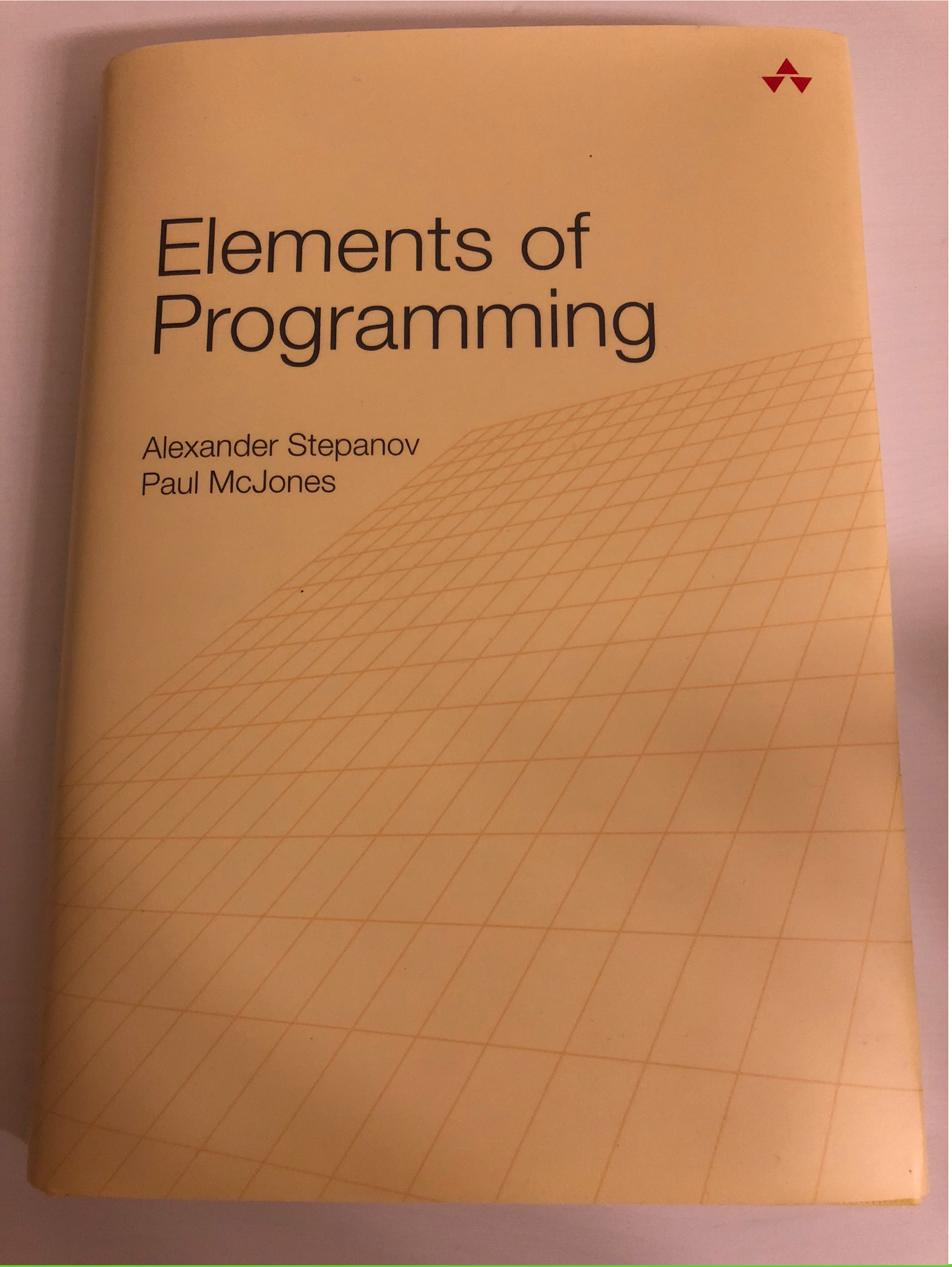
`#define`

`Concept`

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

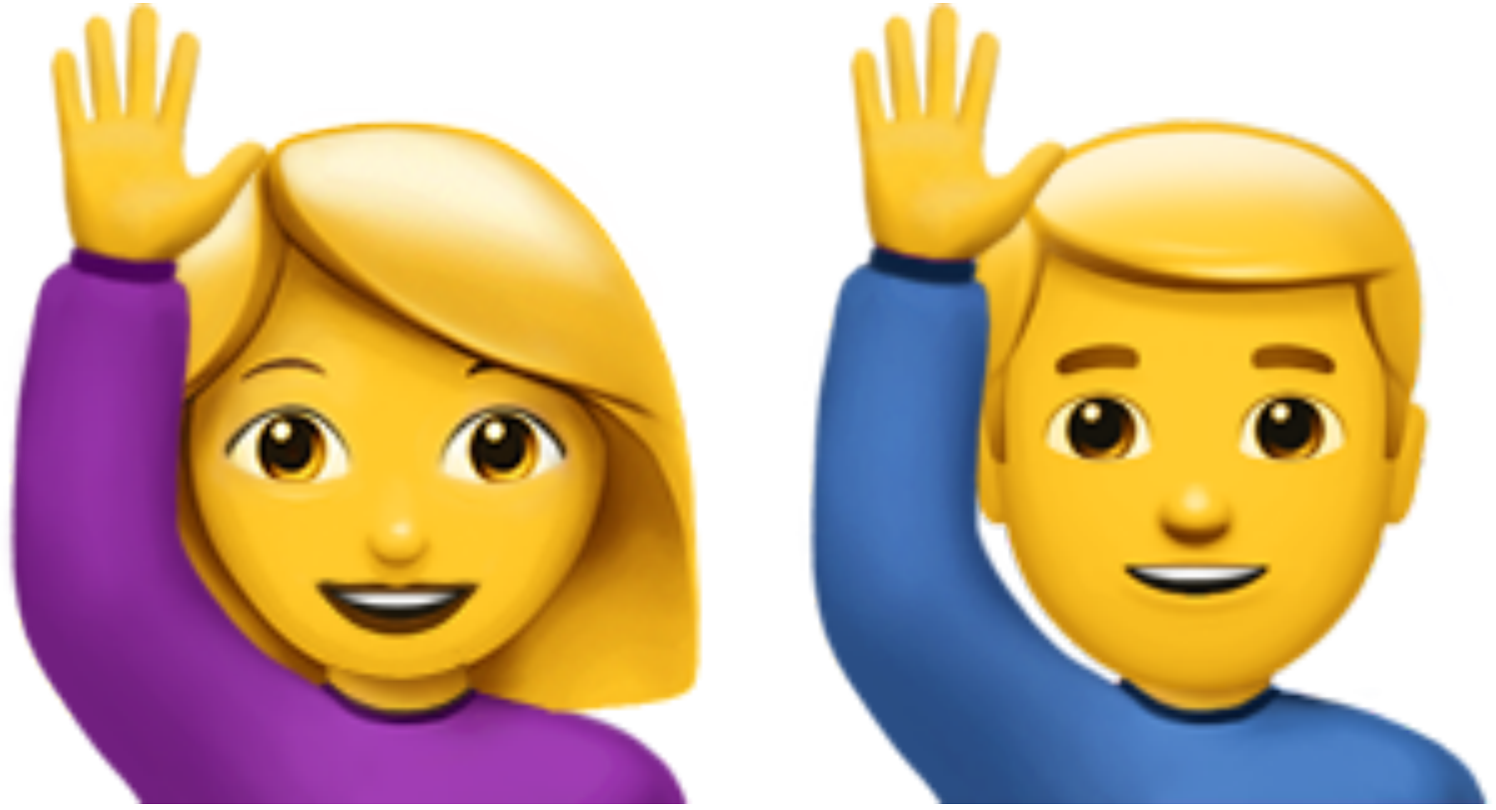
EoP



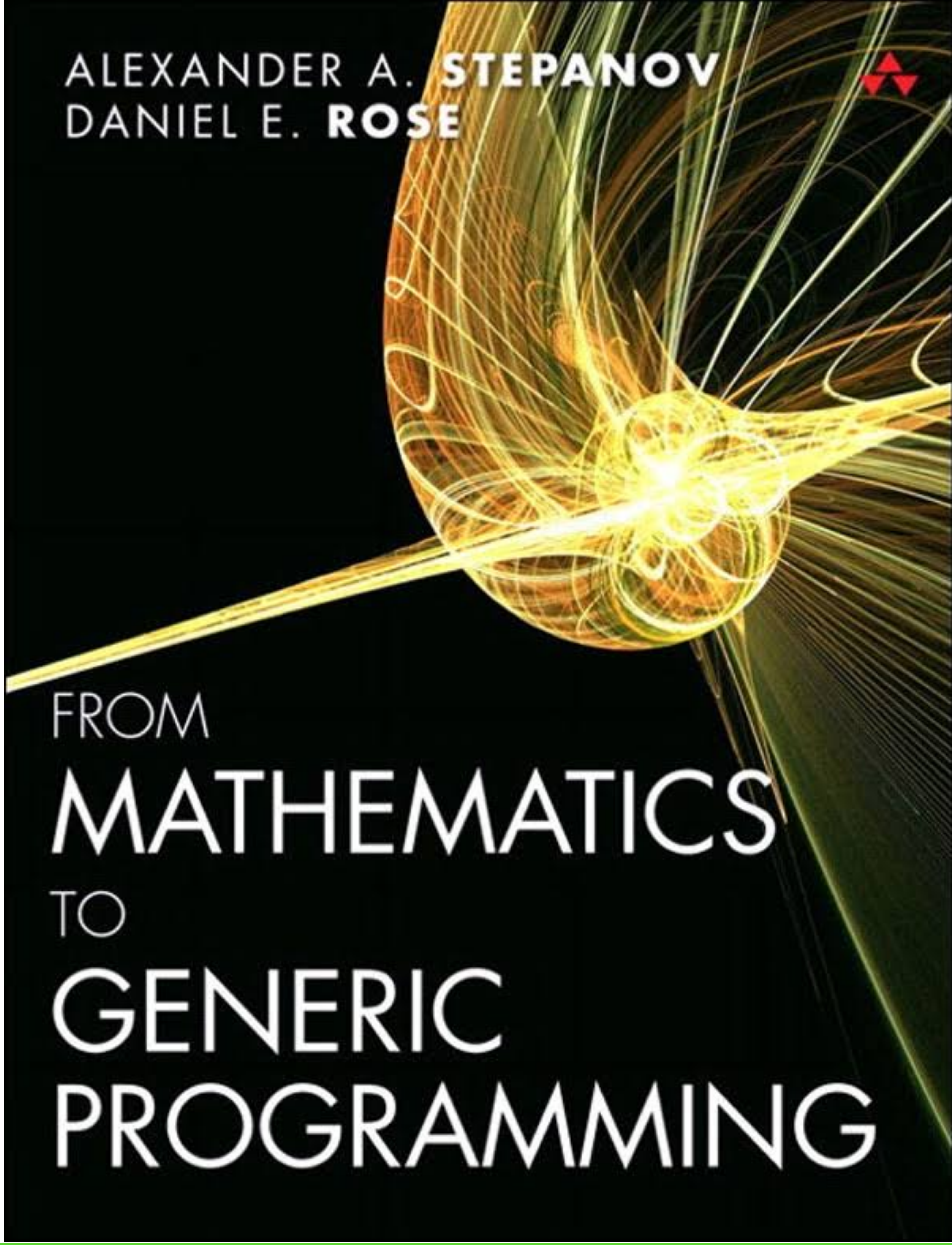


- **Foundations**
- Transformations and Their Orbits
- Associative Operations
- **Linear Orderings**
- Ordered Algebraic Structures
- Iterators
- Coordinate Structures
- Coordinates with Mutable Successors
- Copying
- Rearrangements
- Partition and Merging
- Composite Objects

FM2GP



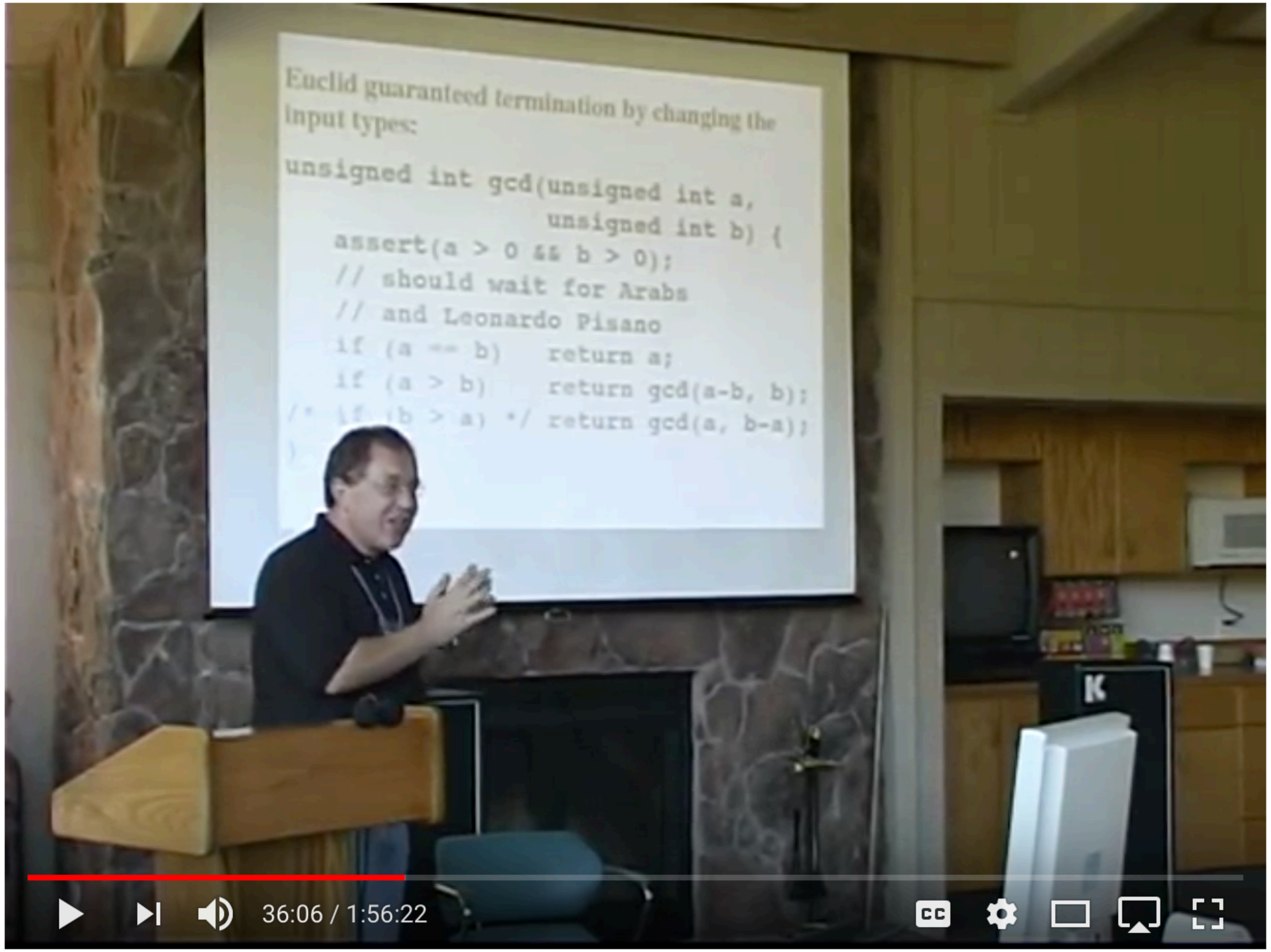
- Egyptian multiplication ~ **1900-1650 BC**
- Ancient Greek number theory
- Prime numbers
- Euclid's GCD algorithm
- Abstraction in mathematics
- Deriving generic algorithms
- Algebraic structures
- Programming concepts
- Permutation algorithms
- Cryptology (RSA) ~ **1977 AD**




cppcon
the c++ conference
2014
A. WEIL Number Theory
The Theory of Equations
Theory of Groups of Finite Order
BOURBAKI

Where am I going with this ?

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

Mathematics Really Does Matter

Richard Feynman

" To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature ...

If you want to learn about nature, to appreciate nature, it is necessary to understand the language that she speaks in.

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."

First of all: *I don't believe you* 😏

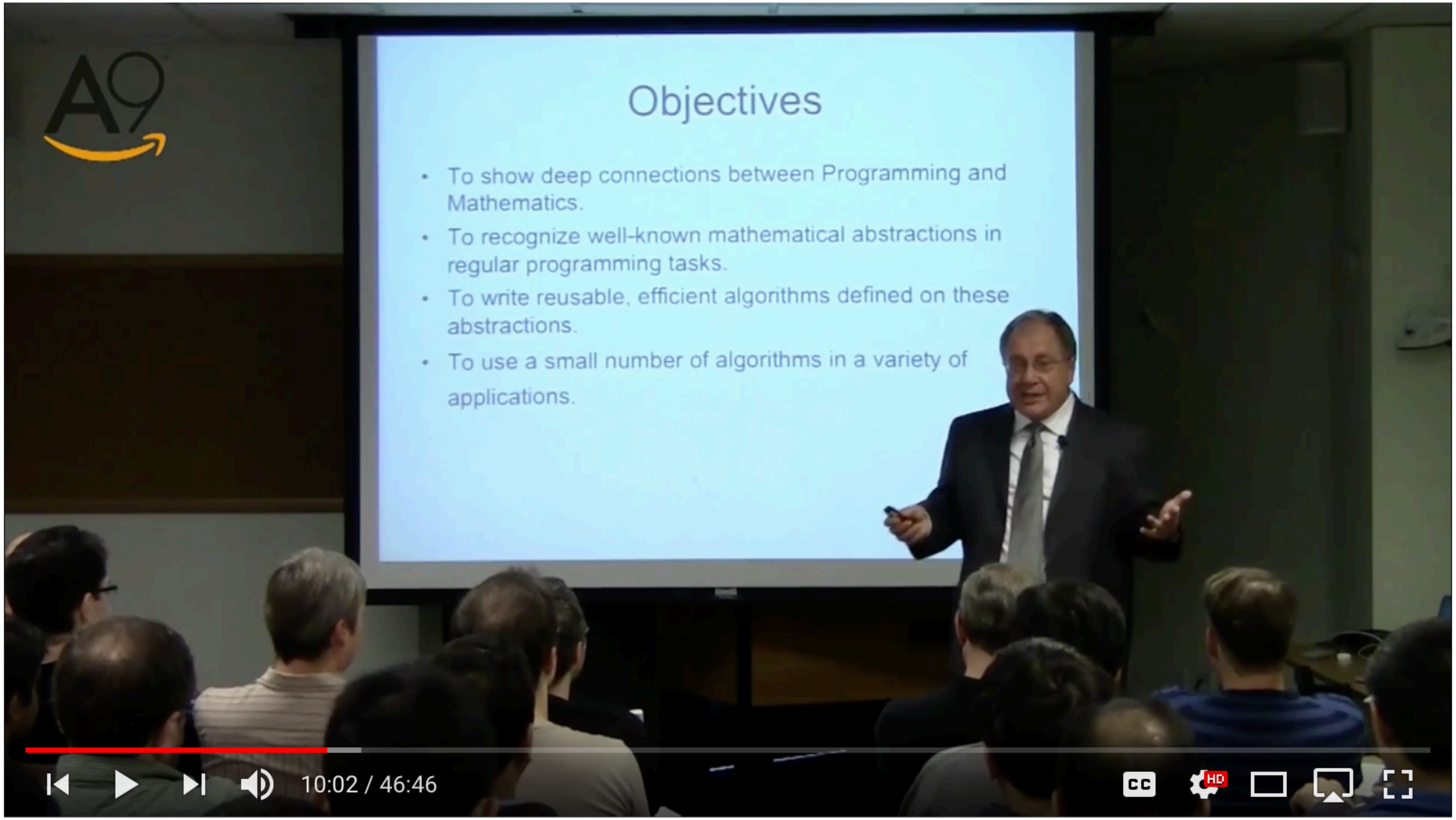
The reason things **just worked** for you
is that other people thought long and hard
about the details of the type system
and the libraries you are using

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

Stay with me !

I'm going somewhere with this...

Three Algorithmic Journeys



Spoils of the Egyptians: Lecture 1 Part 1 https://www.youtube.com/watch?v=wrmXDxn_Zuc

Lectures presented at

A9

2012

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
2012

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

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**, e.g. *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

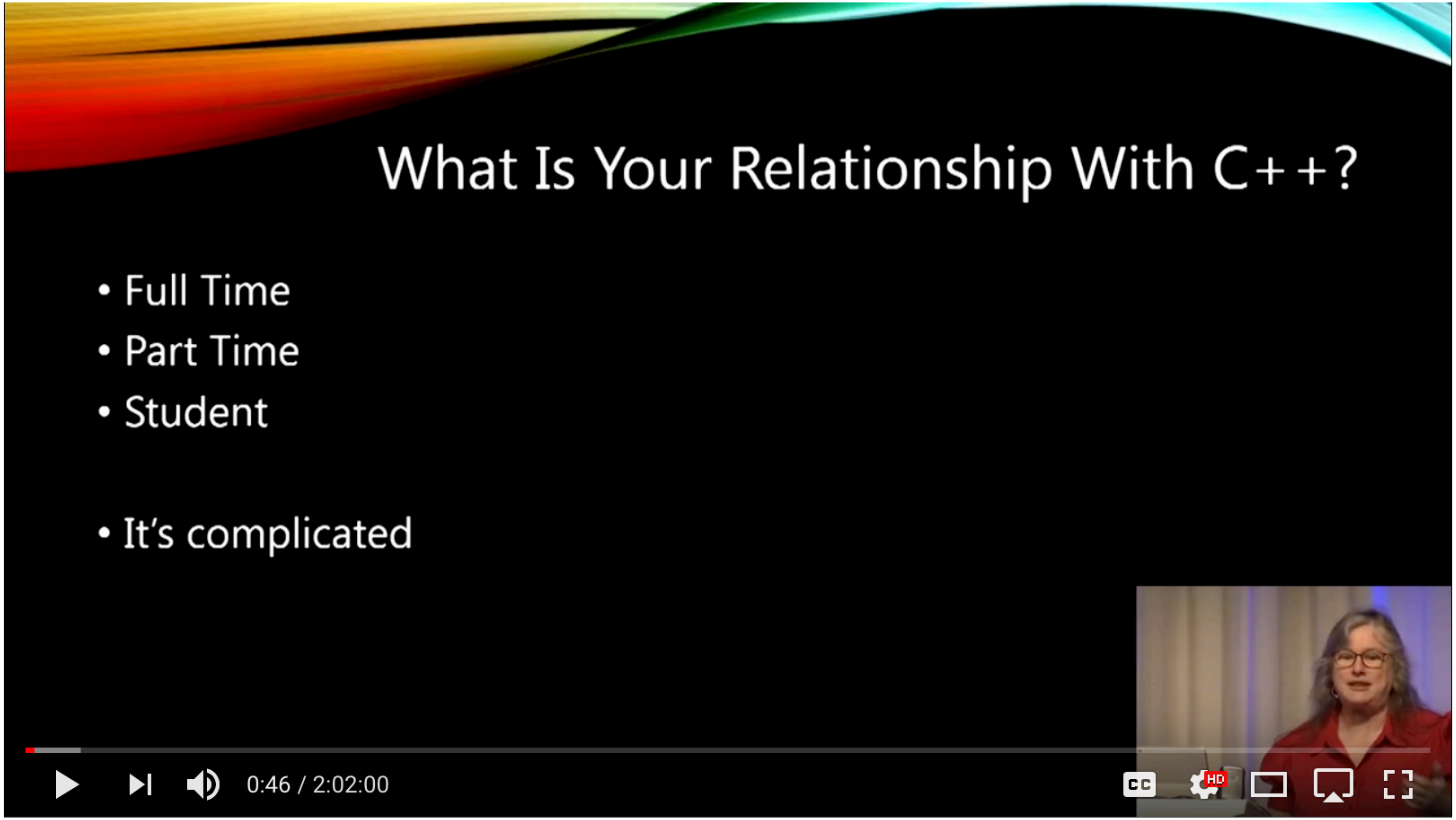
any sophisticated programming tasks, but **simple** and **intuitive** to reason about.

Fundamentals of Generic Programming

Is simplicity a good goal ?

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

🤓



Kate Gregory - It's Complicated - Meeting C++ 2017 Keynote

https://www.youtube.com/watch?v=tTexD26jIN4

Is simplicity a good goal ?

- Simpler code is more readable code
- Unsurprising code is more maintainable code
- Code that moves complexity to abstractions often has less bugs (eg. vector, RAII)
- Compilers and libraries are often much better than you

Kate Gregory, *“It’s Complicated”*, Meeting C++ 2017

Simplicity is Not Just for Beginners

- Requires knowledge (language, idioms, domain)
- Simplicity is an act of generosity (to others, to future you)
- Not about skipping or leaving out

Kate Gregory, *“It’s Complicated”*, Meeting C++ 2017

Revisiting Regular Types
(after 20 years)

https://abseil.io/blog/20180531-regular-types

Titus Winters, 2018

Evokes the **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

Revisiting Regular Types

(after 20 years)

https://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

Alexander Stepanov: STL and Its Design Principles

https://www.youtube.com/watch?v=COuHLky7E2Q

Talk presented at Adobe Systems Inc.
January 30, 2002

http://stepanovpapers.com/stl.pdf

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 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 (`std::accumulate`)

- 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

STL and Its Design Principles

Generic Programming Drawbacks

- abstraction penalty (rarely)
- implementation in the interface
- early binding
- horrible error messages (no formal specification of interfaces, yet)
- duck typing
- algorithm could work on some data types, but fail to work/compile
  on some other new data structures (different iterator category, no copy semantics, etc)

👉 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`

https://en.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 are being formalized in C++20 using **concepts**.

Until then, the burden is on the programmer to ensure that library templates are instantiated with template arguments that satisfy these requirements.

https://en.cppreference.com/w/cpp/named_req

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.

https://en.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

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

I need to tell you a story...

Let's explore one popular STL algorithm

... and its requirements

`std::sort()`

Compare Concept

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

Why is this one special ?
Because ~50 STL facilities (algorithms & data structures) expect some Compare type.

Eg.

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

https://en.cppreference.com/w/cpp/named_req/Compare

`Compare` Concept

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 ?

🤔

https://en.cppreference.com/w/cpp/named_req/Compare

`Compare` Concept

But what kind of *ordering* relationship is needed 🤔

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`

`{ Partial ordering }`

https://en.wikipedia.org/wiki/Partially_ordered_set

Compare Examples

```
vector<string> v = { ... };

sort(v.begin(), v.end());

sort(v.begin(), v.end(), less<>());

sort(v.begin(), v.end(), [](const string & s1, const string & s2)
{
  return s1 < s2;
});

sort(v.begin(), v.end(), [](const string & s1, const string & s2)
{
  return stricmp(s1.c_str(), s2.c_str()) < 0;
});
```

Compare Examples

```
struct Point { int x; int y; };
vector<Point> v = { ... };

sort(v.begin(), v.end(), [](const Point & p1, const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
});
```

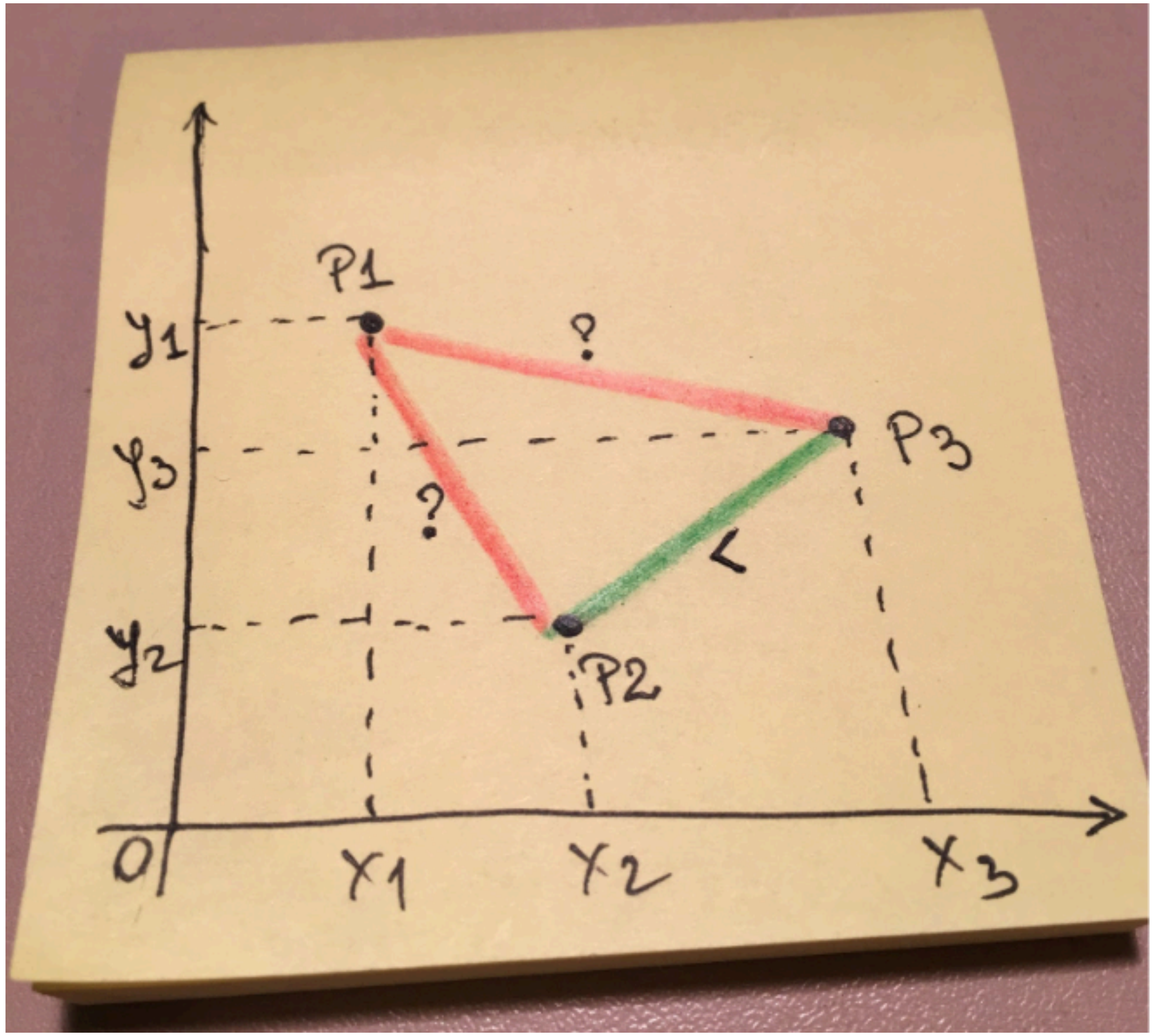
Is this a good Compare predicate for 2D points ?

Compare Examples

```
Let { P1, P2, P3 }
x1 < x2; y1 > y2;
x1 < x3; y1 > y3;
x2 < x3; y2 < y3;

auto comp = [](const Point & p1,
               const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
}

=>
```




```
P2 and P1 are unordered (P2 ? P1) | comp(P2,P1)==false && comp(P1,P2)==false
P1 and P3 are unordered (P1 ? P3) | comp(P1,P3)==false && comp(P3,P1)==false
P2 and P3 are ordered   (P2 < P3) | comp(P2,P3)==true  && comp(P3,P2)==false
```

Compare Examples

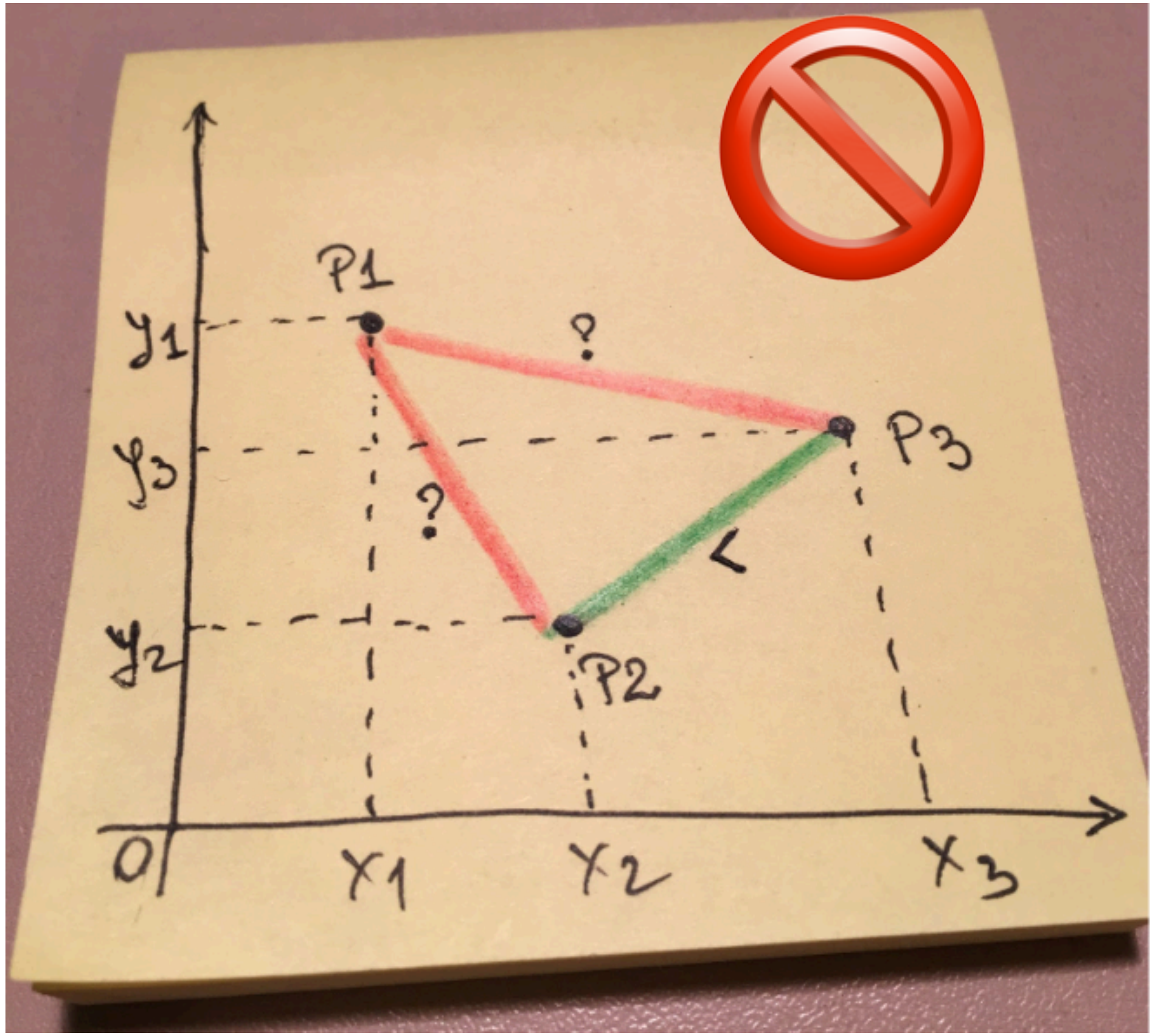
Definition:

```
if comp(a,b)==false && comp(b,a)==false
=> a and b are equivalent
```

```
auto comp = [](const Point & p1,
               const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
}
```

```
=>

P2 is equivalent to P1
P1 is equivalent to P3
P2 is less than     P3
```



`Compare` Concept

Partial ordering **relationship is not enough** 🤔

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

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

where:

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

Strict weak ordering

https://en.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
```

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 **<**

https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

Compare Examples

```
struct Point { int x; int y; };
vector<Point> v = { ... };

sort(v.begin(), v.end(), [](const Point & p1, const Point & p2)
{
  // compare distance from origin
  return (p1.x * p1.x + p1.y * p1.y) <
         (p2.x * p2.x + p2.y * p2.y);
});
```

Is this a good Compare predicate for 2D points ?

Compare Examples

```
struct Point { int x; int y; };
vector<Point> v = { ... };

sort(v.begin(), v.end(), [](const Point & p1, const Point & p2)
{
  if (p1.x < p2.x) return true;
  if (p2.x < p1.x) return false;

  return p1.y < p2.y;
});
```

Is this a good Compare predicate for 2D points ?

`Compare` Examples

The general idea is to pick an order in which to compare elements/parts of the object.

(we first compared by **X** coordinate, and then by **Y** coordinate for *equivalent* **X**)

This strategy is analogous to how a dictionary works,

so it is often called **dictionary order** or **lexicographical order**.

`std::pair<T, U>` defines the six comparison operators
in terms of the corresponding operators of the pair's ***components***

Named Requirements

http://wg21.link/p0898

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`

https://en.cppreference.com/w/cpp/named_req

#define

SemiRegular

DefaultConstructible, MoveConstructible, CopyConstructible

MoveAssignable, CopyAssignable, Swappable

Destructible

http://wg21.link/p0898

#define

Regular

(aka "Stepanov Regular")

SemiRegular

DefaultConstructible, MoveConstructible, CopyConstructible

MoveAssignable, CopyAssignable, Swappable

Destructible

+

EqualityComparable

http://wg21.link/p0898

`Regular`

`(aka "Stepanov Regular")`

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

STL algorithms assume `Regular` data structures

http://wg21.link/p0898

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
```

https://en.cppreference.com/w/cpp/named_req/LessThanComparable

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*.

https://en.wikipedia.org/wiki/Equivalence_relation https://en.cppreference.com/w/cpp/named_req/EqualityComparable

`Equality vs. Equivalence`

For the types that are both `EqualityComparable` and `LessThanComparable`,
the C++ standard library 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*.

Equality vs. Equivalence

Logicians might define **equality** via the following equivalence:

$$a == b \iff \forall \text{ predicate } P,\ P(a) == P(b)$$

But this definition is not very practical in programming :(

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.

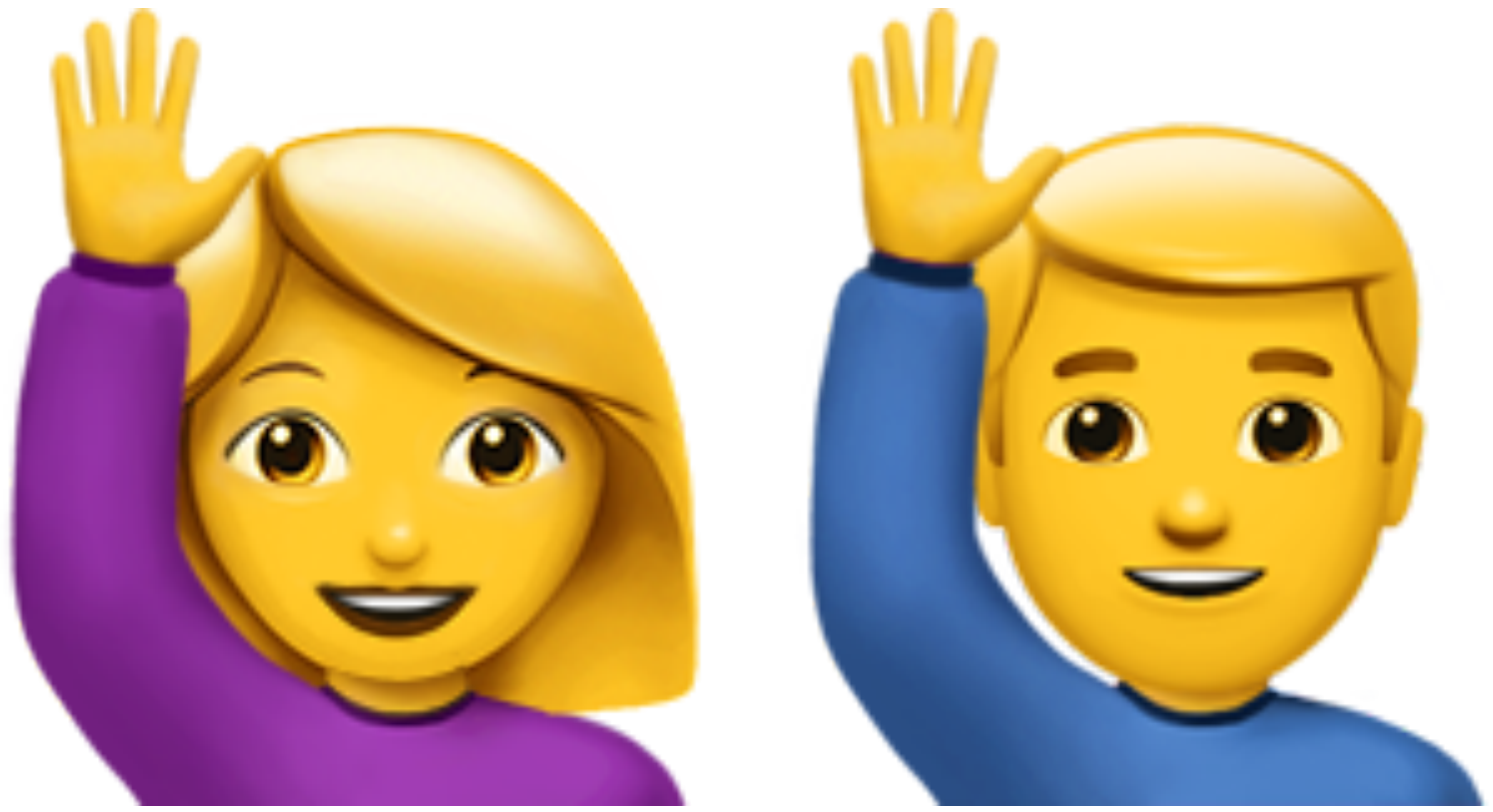
* *“although it still leaves room for judgement”*

http://stepanovpapers.com/DeSt98.pdf

Mandatory Slide

Gauging the audience...

C++98/03 C++11 C++14 C++17



C++20 Three-way comparison

Bringing consistent comparison operations…

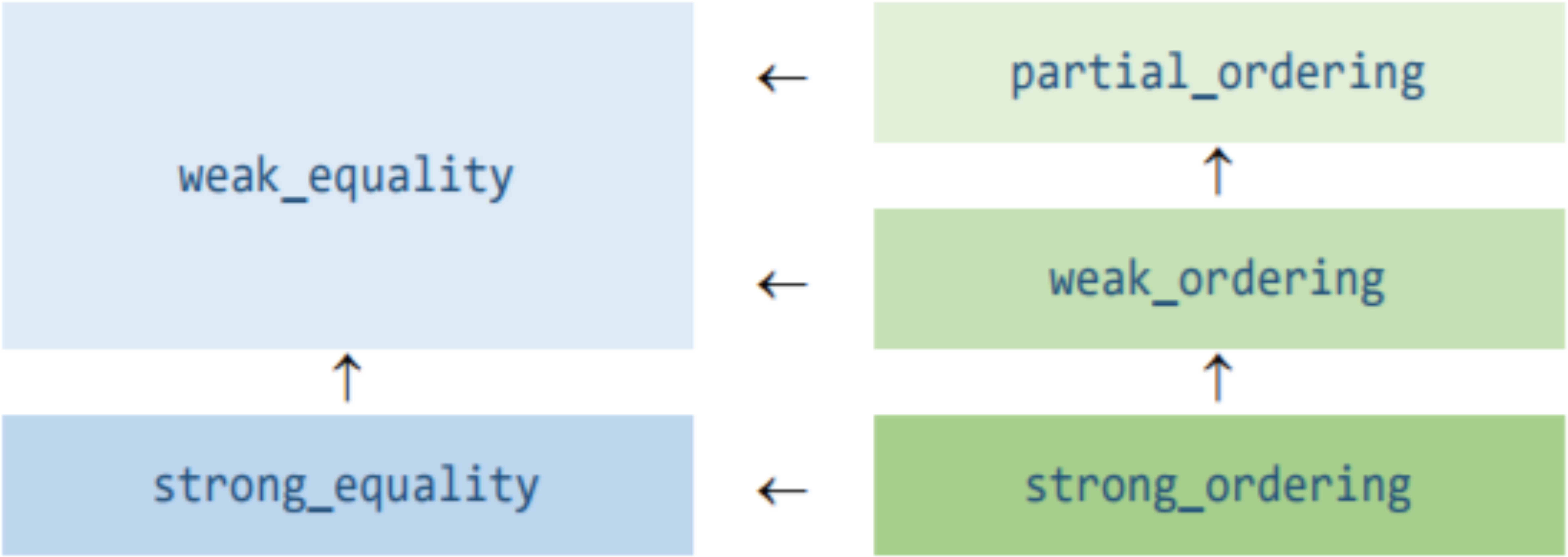
operator <=>

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

http://wg21.link/p0515

C++20 `Three-way comparison`

The comparison categories for: `operator <=>`



It's all about *relation strength* 💪

C++20 `Three-way comparison`

Wish list for: `operator<=>`

I would like to see `<=>` implemented for all STL vocabulary types.

```
std::string
std::string_view
std::optional
std::span
...
```

But, we need to let the dust settle a bit,
so that we have time to really get practical experience with it...

`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

Do you see where this is going ?

`std::optional<T>`

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

The `M` word

`map() / and_then() / or_else()` >>=

chaining

https://wg21.tartanllama.xyz/monadic-optional

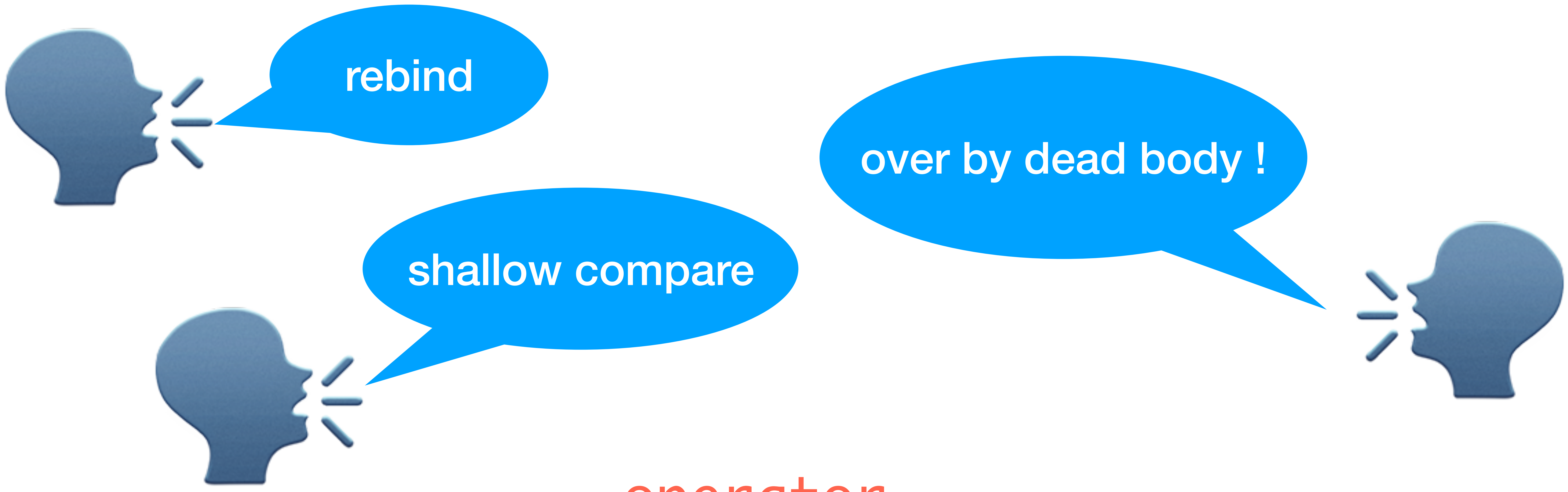
But, wait...

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

😱

```
operator==
```

`std::optional<T &>`



`operator==`

`std::string_view`

"The class template `basic_string_view` describes 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 (caller).

I have a whole talk just on C++17 `std::string_view`

Enough `string_view` to hang ourselves

CppCon 2018

http://sched.co/FnL6

`std::string_view`

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

- Arthur O'Dwyer

https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/

`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***.

https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/

`std::string_view` is a borrow type

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* safely from functions (no ownership semantics)

https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/

`std::string_view` is a borrow type

`string_view` is perhaps the first "mainstream" ***borrow type.***

BUT:

`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.

https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/

`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**

Generally, when used properly (as function parameter),
`string_view` works well..., **as if** a **Regular** type

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

I give you `std::span`

the very confusing type that the world's best C++ experts are not quite sure what to make of

🤦‍♂️

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

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>`

Photo credit: Corentin Jabot

https://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

Things to consider:

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

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`

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

This was the most fun talk I had to write 🤓

📖 Mainly because of some **wonderful people**, that wrote excellent articles about this topic

I want to thank all of them 👏

and encourage **you** to read their work

📖 References I encourage you to study

Alexander Stapanov, Paul McJones
Elements of Programming (2009)
http://elementsofprogramming.com

Alexander Stapanov, James C. Dehnert
Fundamentals of Generic Programming (1998)
http://stepanovpapers.com/DeSt98.pdf

Alexander Stepanov
STL and Its Design Principles - presented at Adobe Systems Inc., January 30, 2002
https://www.youtube.com/watch?v=COuHLky7E2Q
http://stepanovpapers.com/stl.pdf

Bjarne Stroustrup, Andrew Sutton, et al.
A Concept Design for the STL (2012)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

📖 References I encourage you to study

Titus Winters
Revisiting Regular Types
https://abseil.io/blog/20180531-regular-types

Corentin Jabot (cor3ntin)
A can of span
https://cor3ntin.github.io/posts/span/

Christopher Di Bella
Prepping Yourself to Conceptify Algorithms
https://www.cjdb.com.au/blog/2018/05/15/prepping-yourself-to-conceptify-algorithms.html

Tony Van Eerd
Should Span be Regular?
http://wg21.link/P1085

📖 References I encourage you to study

Simon Brand

Functional exceptionless error-handling with optional and expected
https://blog.tartanllama.xyz/optional-expected/

Spaceship Operator
https://blog.tartanllama.xyz/spaceship-operator/

Monadic operations for std::optional
https://wg21.tartanllama.xyz/monadic-optional

📖 References I encourage you to study

Arthur O'Dwyer

Default-constructibility is overrated
https://quuxplusone.github.io/blog/2018/05/10/regular-should-not-imply-default-constructible/

Comparison categories for narrow-contract comparators
https://quuxplusone.github.io/blog/2018/08/07/lakos-rule-for-comparison-categories/

std::string_view is a borrow type
https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/

📖 References I encourage you to study

Barry Revzin

Non-Ownership and Generic Programming and Regular types, oh my!
https://medium.com/@barryrevzin/non-ownership-and-generic-programming-and-regular-types-oh-my

Should Span Be Regular?
https://medium.com/@barryrevzin/should-span-be-regular-6d7e828dd44

Implementing the spaceship operator for optional
https://medium.com/@barryrevzin/implementing-the-spaceship-operator-for-optional-4de89fc6d5ec

📖 References I encourage you to study

Jonathan Müller

Mathematics behind Comparison

#1: Equality and Equivalence Relations
https://foonathan.net/blog/2018/06/20/equivalence-relations.html

#2: Ordering Relations in Math
https://foonathan.net/blog/2018/07/19/ordering-relations-math.html

#3: Ordering Relations in C++
https://foonathan.net/blog/2018/07/19/ordering-relations-programming.html

#4: Three-Way Comparison
https://foonathan.net/blog/2018/09/07/three-way-comparison.html

C++ Slack is your friend

https://cpplang.slack.com

CppLang Slack auto-invite:
https://cpplang.now.sh/

Cpplang
cpplang.slack.com

CppCast

```
auto CppCast = pod_cast<C++>("http://cppcast.com");
```

Rob Irving @robwirving

Jason Turner @lefticus

http://cpp.chat

https://www.youtube.com/channel/UCsefcSZGxO9lTBqFbsV3sJg/

https://overcast.fm/itunes1378325120/cpp-chat

Jon Kalb	@_JonKalb
Phil Nash	@phil_nash

Regular Types and Why Do I Care ?

September, 2018

@ciura_victor

Victor Ciura
Technical Lead, Advanced Installer
www.advancedinstaller.com

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

https://github.com/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

Questions

@ciura_victor