code::dive

A Short Life span< >
For a Regular Mess

November 21, 2019
Wrocław

CAPHYON

@ciura_victor

Victor Ciura
Principal Engineer
CAPHYON

Abstract

By now you probably heard about “Regular Types and Why Do I Care” :) This would be Part 2 of the journey we’ll take together, where we get a chance to explore std::span<T> through our Regular lens. Don’t worry if you’ve missed Part 1; we’ll have plenty of time to revisit the important bits, as we prepare to span our grasp into C++20.

“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. C++20 Concepts are based on precisely defined foundational type requirements such as Semiregular, Regular, EqualityComparable, etc.

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

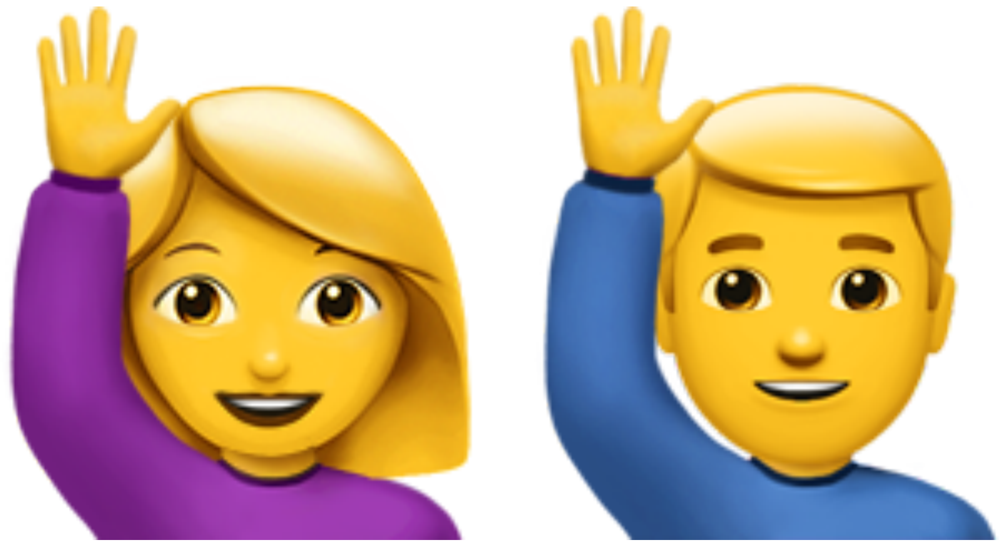
This talk will explore the relation between Regular types (and other concepts) and new STL additions like std::span<T> with examples, common pitfalls and guidance.

Who Am I ?

Advanced Installer

Clang Power Tools

@ciura_victor



Regular Types
and
Why Do I Care ?

CppCon 2018 | Meeting C++ 2018 | ACCU 2019

`Regular` types

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"

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

Whole-part semantics

C++17

Concepts

Ordering Relations

C++23~

C++20

Requirements

Lifetimes

Equality

Cpp Core Guidelines

`std::span`

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 ?

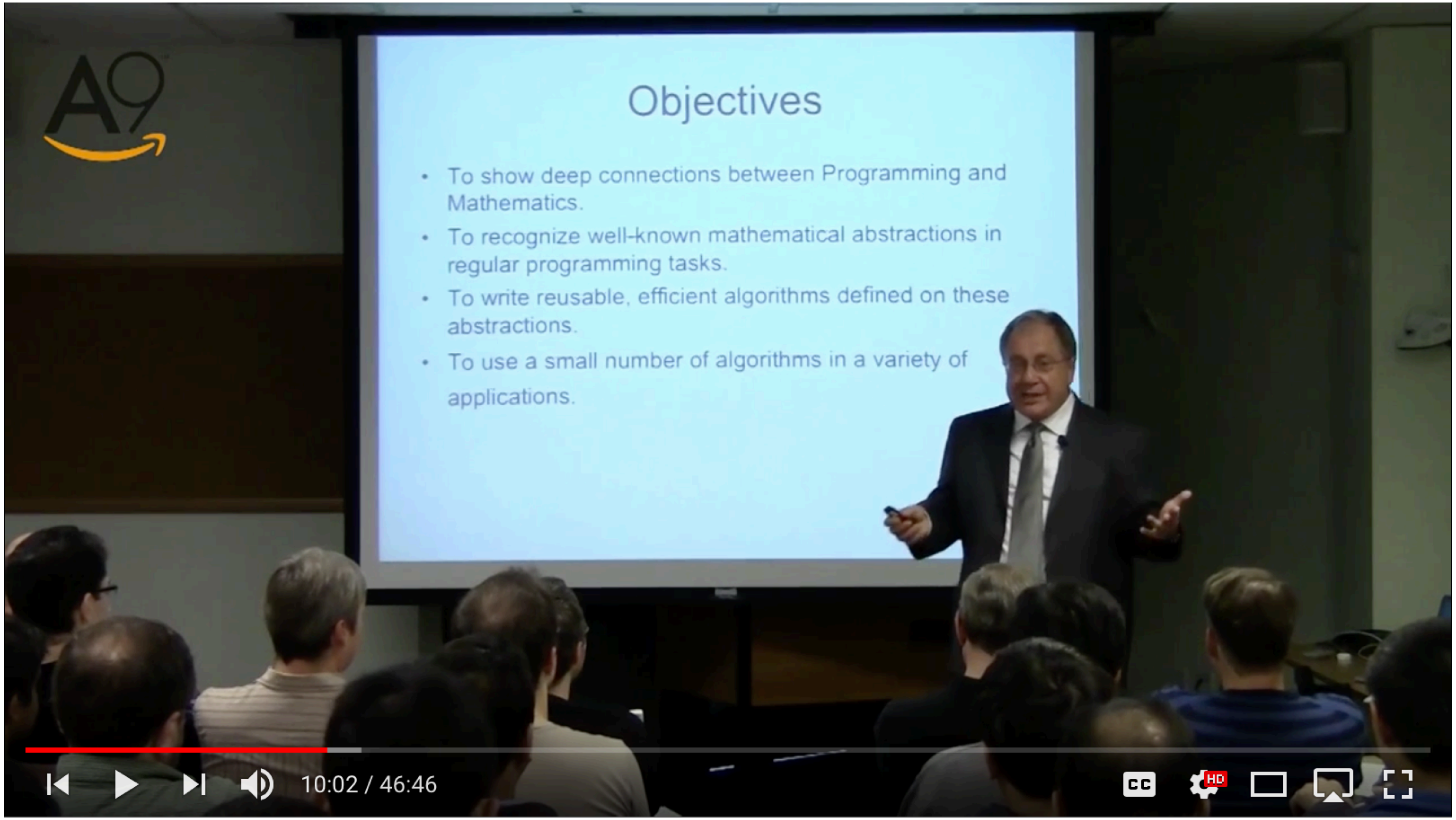
Titus Winters - Modern C++ API Design
youtube.com/watch?v=tn7oVNrPM8I

Let's start with the beginning...

2,000 BC

😁

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.

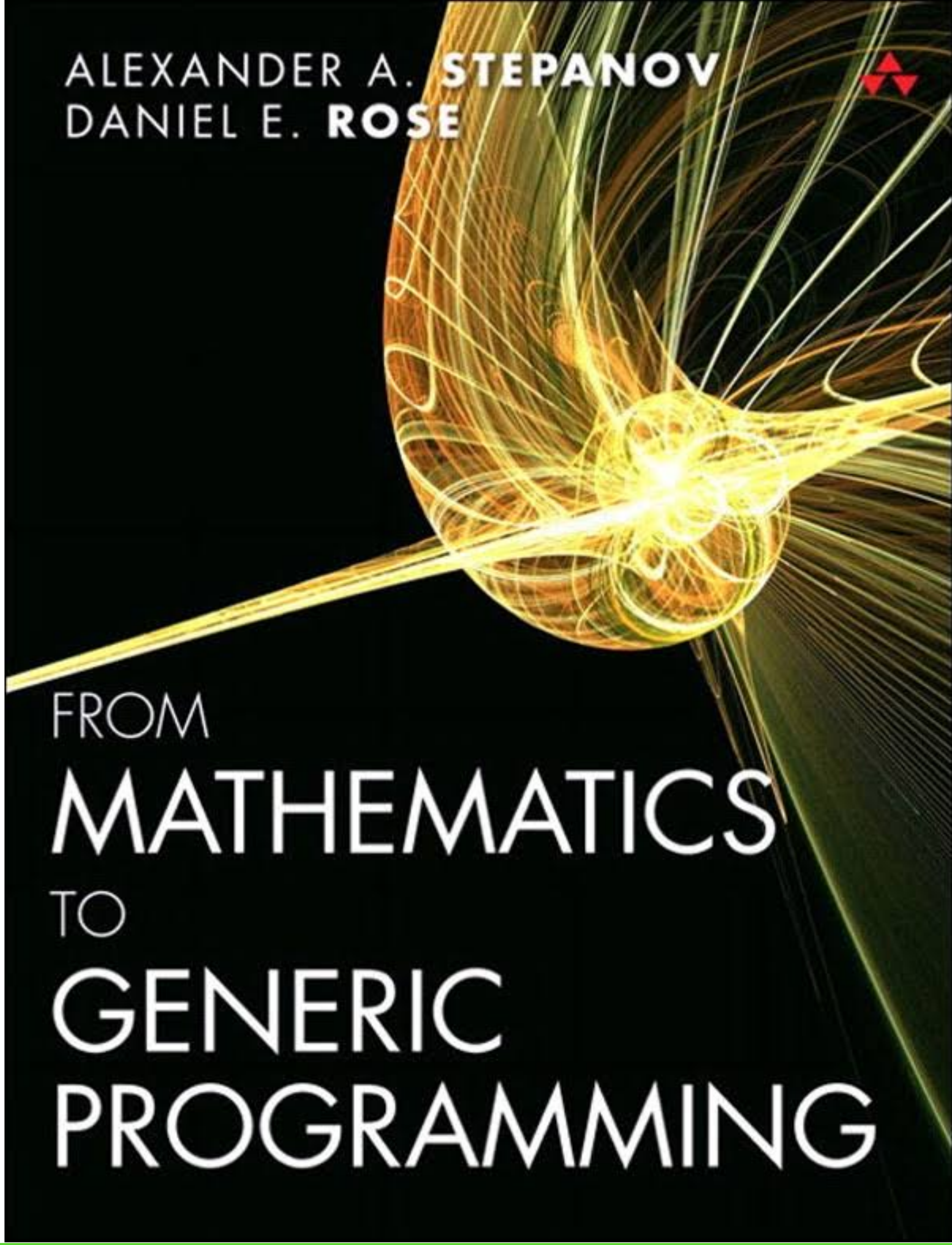
II. Heirs of Pythagoras (12h)
How division with remainder led to discovery of many fundamental abstractions.

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

Lectures presented at A9

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

A value cannot change.

#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).

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.

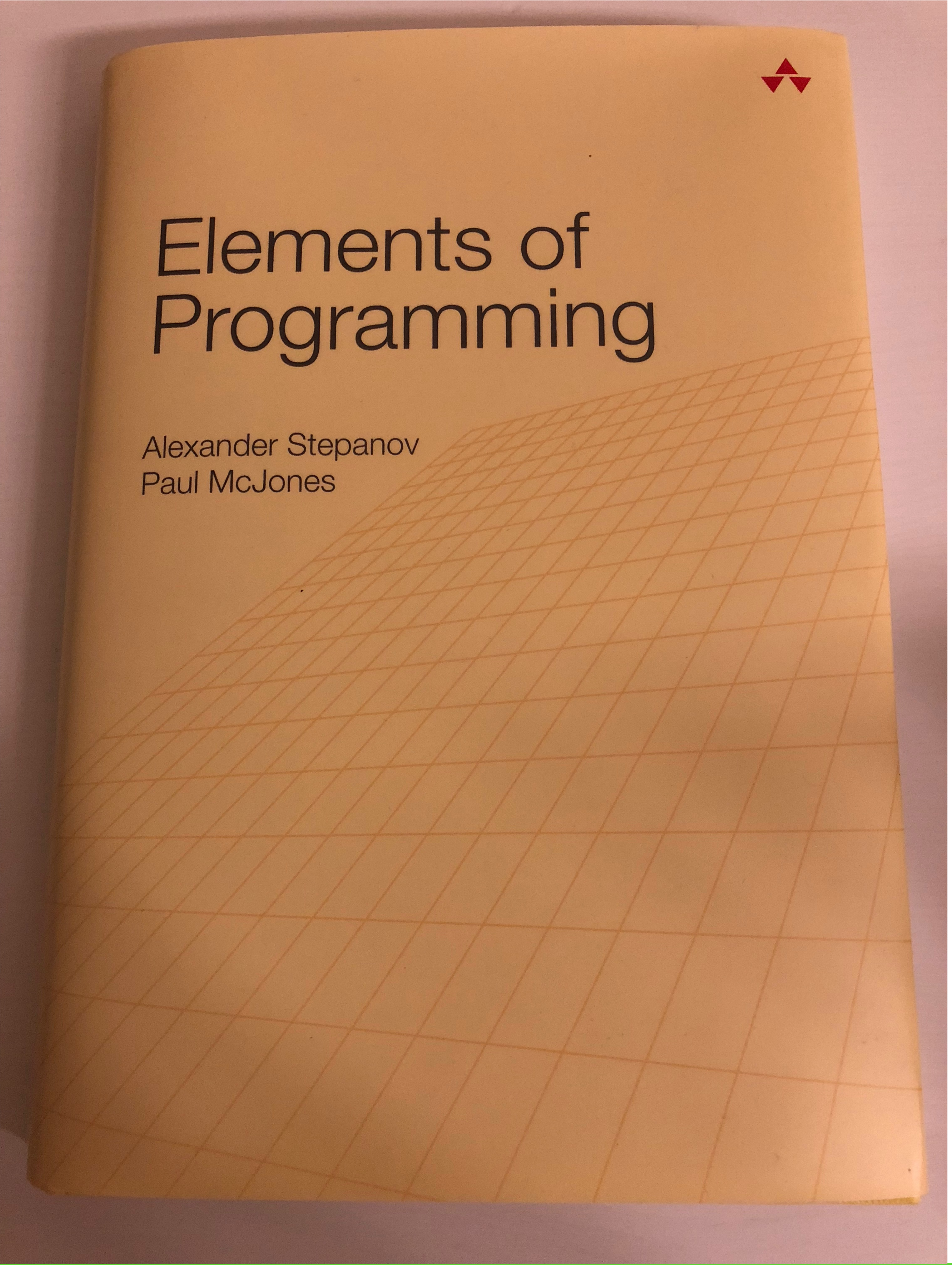
#EoP

#define

Concept

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

#EoP

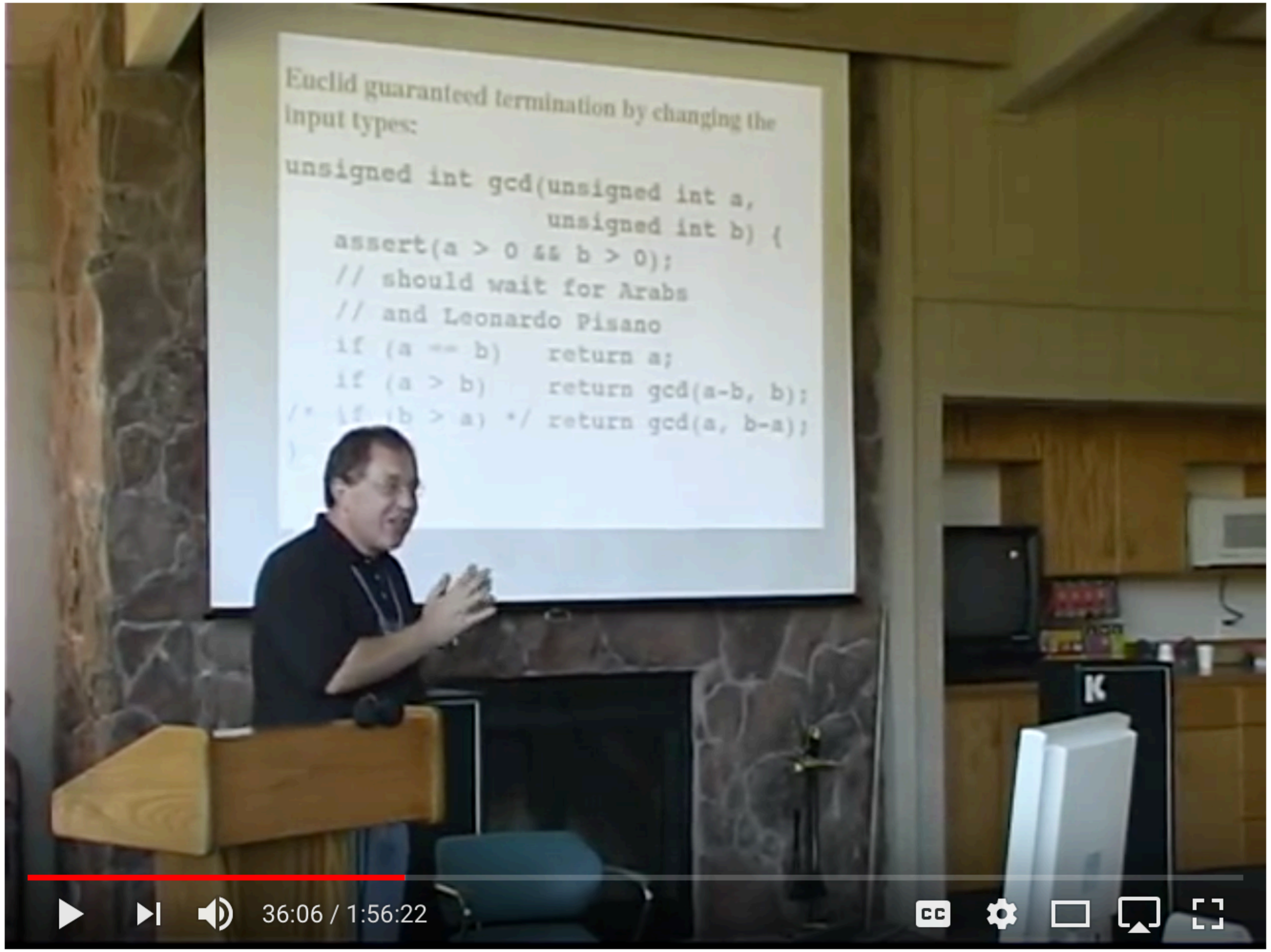


- **Foundations**
- Transformations and Their Orbits
- Associative Operations
- **Linear Orderings**
- Ordered Algebraic Structures
- Iterators
- Coordinate Structures
- Coordinates with Mutable Successors
- Copying
- Rearrangements
- Partition and Merging
- Composite Objects

Free PDF

http://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 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

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

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
(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 (**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

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

C++20 Renaming concepts from `Pascal/CamelCase` to `snake_case`

https://wg21.link/p1754

Boolean	boolean
EqualityComparable	equality_comparable
EqualityComparableWith	equality_comparable_with
StrictTotallyOrdered	totally_ordered
StrictTotallyOrderedWith	totally_ordered_with
Movable	movable
Copyable	copyable
Semiregular	semiregular
Regular	regular
Invocable	invocable
RegularInvocable	regular_invocable
Predicate	predicate

SizedSentinel	sized_sentinel_for
InputIterator	input_iterator
OutputIterator	output_iterator
ForwardIterator	forward_iterator
BidirectionalIterator	bidirectional_iterator
rator	random_access_iterator
r	contiguous_iterator
IndirectUnaryInvocable	indirectly_unary_invocable
IndirectRegularUnaryInvocable	indirectly_regular_unary_invoca-ble
IndirectUnaryPredicate	indirect_unary_predicate
IndirectRelation	indirect_relation

C++20 Renaming concepts from `PascalCase`/`CamelCase` to `snake_case`

https://wg21.link/p1754

Adults lack imagination...

Photo: @AdiShavit

With apologies to Antoine de Saint-Exupéry (The Little Prince)

C++20 Renaming concepts from `Pascal/CamelCase` to `snake_case`

https://wg21.link/p1754

I liked the original PascalCase because:

- it's desirable to make concepts **Stand Out** (they are ***policies*** rather than types)
- concepts are not types and should thus be named differently from standard types
- of consistency with standard template parameters
  eg.
  ```
  template<class CharT, class Traits, class Allocator>
  class basic_string;
  ```
- confusion with **type traits**:
  eg. having both `std::copy_constructible` and `std::is_copy_constructible`
  mean different things and give subtly different answers in some cases
  => creates user confusion and pitfalls

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)

Using STL - Compare Concept

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 ?

🤔

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

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
```

< `LessThanComparable`

https://en.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

http://wg21.link/p0898

Examples from STL:

`DefaultConstructible`, `MoveConstructible`, `CopyConstructible`

`MoveAssignable`, `CopyAssignable`, `Swappable`

`Destructible`

`LessThanComparable`, `EqualityComparable`

`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

The STL was written with *Regularity* as its basis

Also, see the **Palo Alto TR**
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

http://wg21.link/p0898

`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 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

Defining **equality** is hard 😞

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.

😓

http://stepanovpapers.com/DeSt98.pdf

C++20 Three-way comparison

Bringing consistent comparison operations…

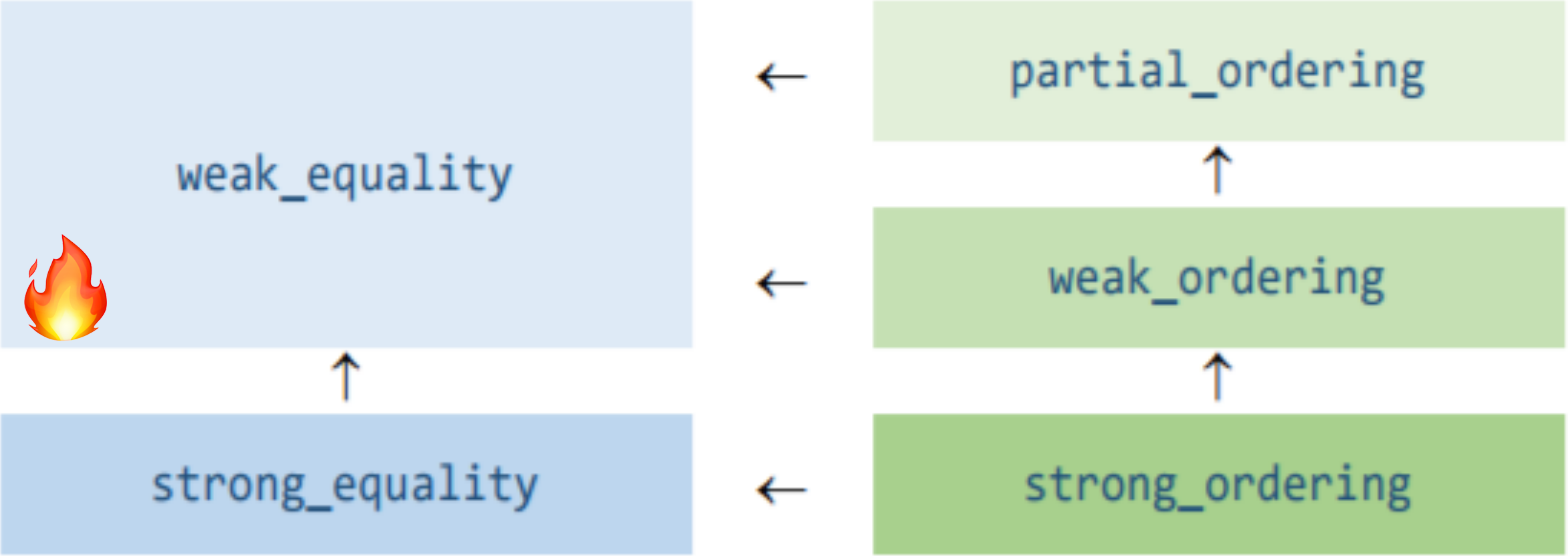
`operator <=>`

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

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

Convenience

`operator<=>`

`operator!=`

`operator==`

Efficiency ?

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

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

https://wg21.link/P1185

C++20 `Three-way comparison`

The Mothership Has Landed

Adding `operator<=>` to the whole STL

Barry Revzin
2019-07 Cologne ISO C++ Committee Meeting

https://wg21.link/P1614

C++20 Three-way comparison

https://www.youtube.com/watch?v=8jNXy3K2Wpk

Before we get too far with C++20

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

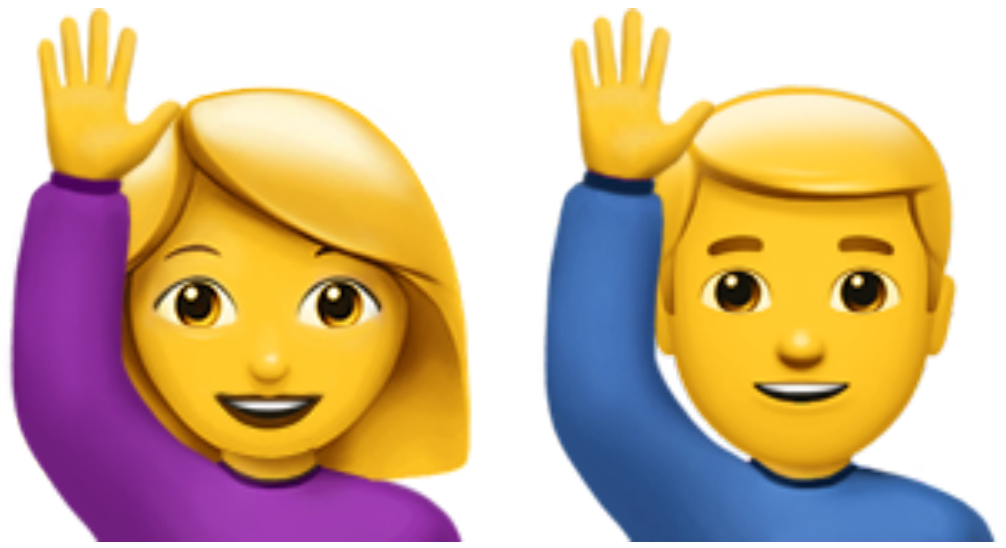
C++17

`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



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

Enough `string_view` to hang ourselves

CppCon 2018

https://www.youtube.com/watch?v=xwP4YCP_0q0 `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***

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

https://quuxplusone.github.io/blog/2018/03/27/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)

`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

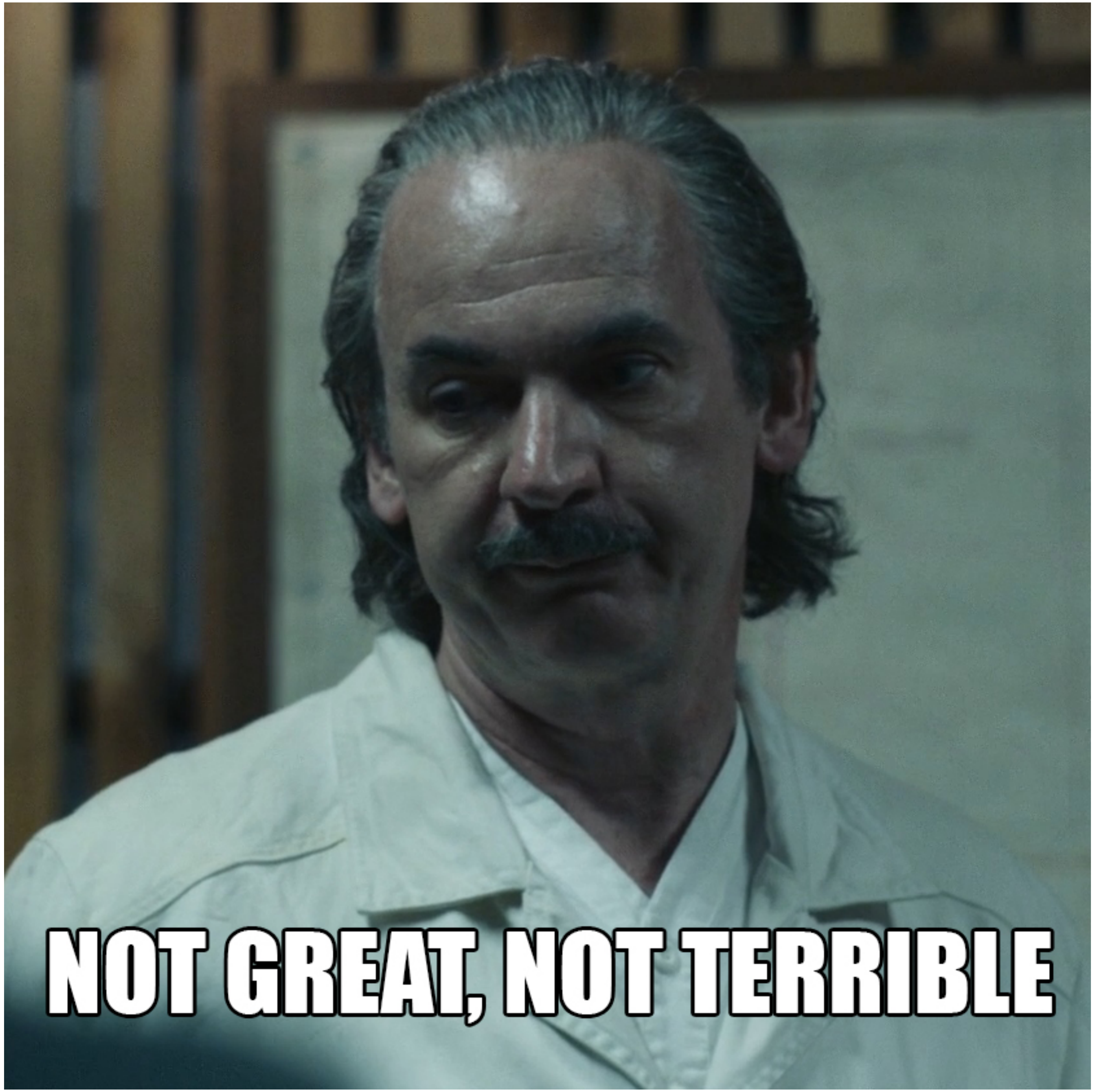
When the underlying data is **extant** and **constant**

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

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

`string_view` works well...

as if it is a **Regular** type


NOT GREAT, NOT TERRIBLE

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>

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

Historical Background

C++ Core Guidelines

github.com/isocpp/CppCoreGuidelines

Editors:
- Bjarne Stroustrup
- Herb Sutter

F.24: Use a span<T> or a span_p<T> to designate a half-open sequence

CppCoreGuidelines.md#Rf-range

Pro.bounds: Bounds safety profile

CppCoreGuidelines.md#SS-bounds

C++ Core Guidelines

F.24: Use a span<T> or a span_p<T> to designate a half-open sequence

Reason: Informal/non-explicit ranges are a source of errors

Ranges are extremely common in C++ code.

Typically, they are **implicit** and their correct use is very hard to ensure.

Given a pair of arguments `(p, n)` designating an array `[p:p+n)`,

it is in general impossible to know if there really are `n` elements to access following `*p`

GSL `span<T>` and `span_p<T>` were designed to solve this problem, by given an **explicit** context

C++ Core Guidelines

Pro.bounds: Bounds safety profile

- Don't use pointer arithmetic; use **span** instead
- Only index into arrays using constant expressions
- No array-to-pointer decay
- Don't use standard-library functions and types that are not bounds-checked

Pass pointers to single objects (only) and Keep pointer arithmetic simple

Use the standard library in a type-safe manner

Historical Background

GSL: Guidelines Support Library

github.com/microsoft/GSL

The library includes types like `span`, `string_span`, `owner` **and others**

github.com/Microsoft/GSL/blob/master/include/gsl/span

(circa 2017)

Historical Background

`std::span`

" Comes directly from the **C++ Core Guidelines**' **GSL** and is intended to be a replacement especially for unsafe C-style (pointer, length) parameter pairs.
We expect to be used pervasively as a vocabulary type for function parameters in particular.

span: bounds-safe views for sequences of objects

wg21.link/p0122 **Neil MacIntosh & Stephan T. Lavavej**

https://herbsutter.com/2018/04/02/trip-report-winter-iso-c-standards-meeting-jacksonville/

Automatic Checkers

Use the C++ Core Guidelines checkers

- core guideline checkers are installed by default in **Visual Studio 2017** and **Visual Studio 2019**

  docs.microsoft.com/en-us/visualstudio/code-quality/using-the-cpp-core-guidelines-checkers

- LLVM **clang-tidy** `-checks='-*,cppcoreguidelines-*'`

  clang.llvm.org/extra/clang-tidy/checks/list.html

- ClangPowerTools

  clangpowertools.com (powered by **clang-tidy**)

Automatic Checkers

- **LLVM clang-tidy**

  clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines-pro-bounds-array-to-pointer-decay.html

  This check flags all array to pointer decays. Pointers should not be used as arrays.
  span<T> is a bounds-checked, safe alternative to using pointers to access arrays.

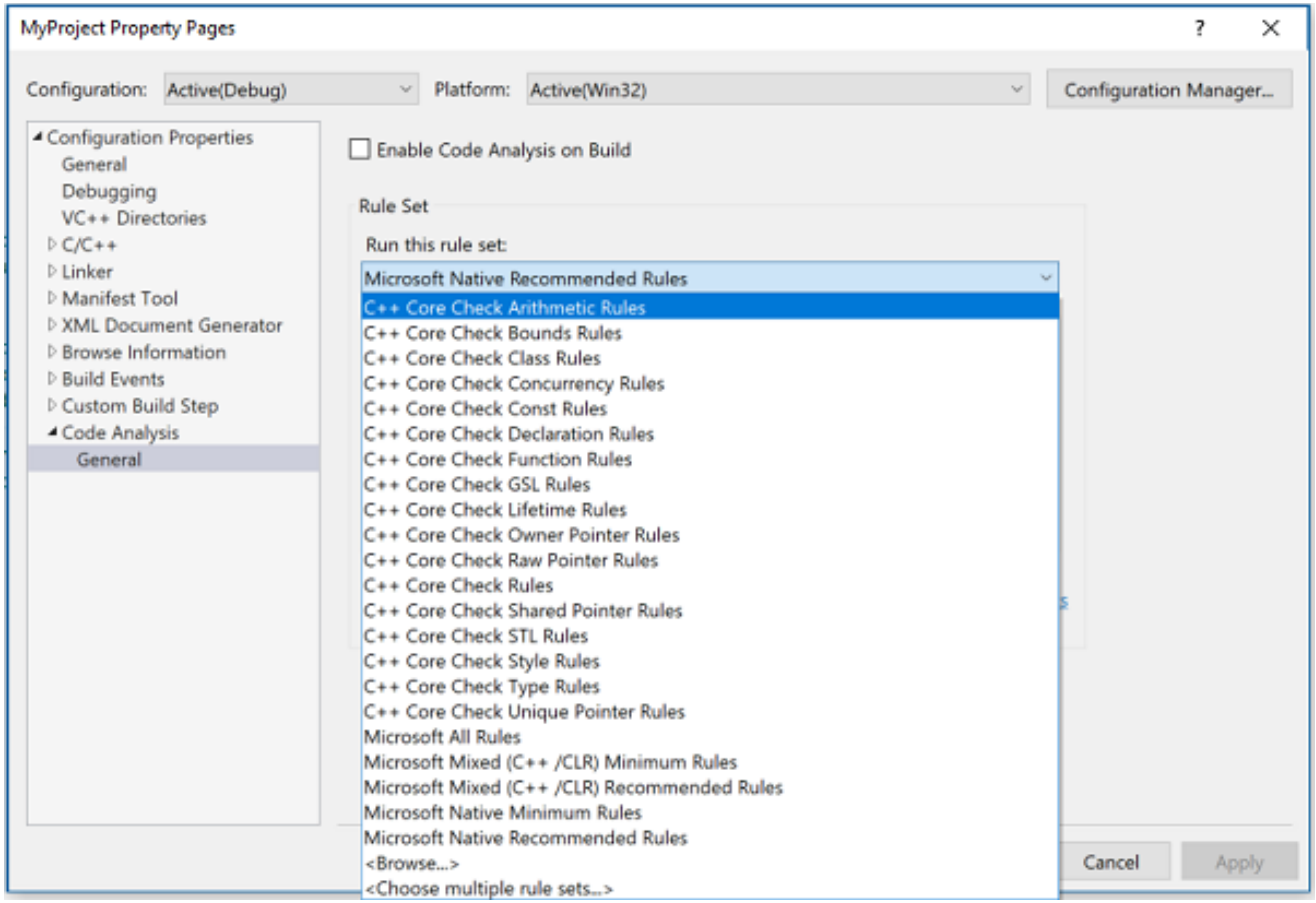
  clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines-pro-bounds-pointer-arithmetic.html

  This check flags all usage of pointer arithmetic, because it could lead to an invalid pointer.
  Subtraction of two pointers is **not flagged** by this check.

  Pointers should only refer to single objects, and pointer arithmetic is fragile and easy to get wrong.
  span<T> is a bounds-checked, safe type for accessing arrays of data.

Automatic Checkers

- **Visual Studio 2017/2019**



`C26485`
Bounds.3: No array-to-pointer decay.

`C26481`
Bounds.1: Don't use pointer arithmetic. Use span instead.

docs.microsoft.com/en-us/visualstudio/code-quality/using-the-cpp-core-guidelines-checkers

Automatic Checkers

- **Visual Studio 2017/2019**

C26494
Type.5: Always initialize an object

C26485
Bounds.3: No array-to-pointer decay

C26481
Bounds.1: Don't use pointer arithmetic
Use span instead

```
int arr[10];              // warning C26494
int * p = arr;            // warning C26485

[[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1
{
   int * q = p + 1;     // warning C26481 (suppressed)
   p = q++;             // warning C26481 (suppressed)
}
```

docs.microsoft.com/en-us/visualstudio/code-quality/using-the-cpp-core-guidelines-checkers

Automatic Checkers

```
static unsigned char* encodeBytesGroup(unsigned char* data, const unsigned char* buffer, int bits)
{
    assert(bits >= 1 && bits <= 8);

    if (bits == 1)
        return data;

    if (bits == 8)
    {
        memcpy(data, buffer, kByteGroupSize);
        return data + kByteGroupSize;
    }

    size_t byte_size = 8 / bits;
    assert(kByteGroupSize % byte_size == 0);

    // fixed portion: bits bits for each value
    // variable portion: full byte for each out-of-range value (using 1...1 as sentinel)
```

👉 26481: Don't use pointer arithmetic. Use span instead (bounds.1).

🤔

https://twitter.com/zeuxcg/status/1088686771037122560?s=21

Automatic Checkers

```
}

for (size_t i = 0; i < kByteGroupSize; ++i)
{
    if (buffer[i] >= sentinel)
    {
        *data
    }
}

return data;
```

const unsigned char *buffer

26481: Don't use pointer arithmetic. Use span instead (bounds.1).

https://twitter.com/zeuxcg/status/1088686771037122560?s=21

C++20

std::span

Defined in header <span>

```
template<
    class T,
    std::size_t Extent = std::dynamic_extent
> class span;
```

an object that can refer to a **contiguous** sequence of objects with the first element of the sequence at position zero

A typical implementation holds only two members:

- a pointer to T
- a size

A span can either have:

- a ***static extent*** (number of elements is known and encoded in the type)
- a ***dynamic extent***

Construct a span

Belfast, Nov 2019

```
constexpr span() noexcept;
constexpr span(pointer ptr, index_type count);
constexpr span(pointer first, pointer last);
template <std::size_t N>
constexpr span(element_type (&arr)[N]) noexcept;
template <std::size_t N>
constexpr span(std::array<value_type, N>& arr) noexcept;
template <std::size_t N>
constexpr span(const std::array<value_type, N>& arr) noexcept;
template <class Container>
constexpr span(Container& cont);
template <class Container>
constexpr span(const Container& cont);
template <class U, std::size_t N>
constexpr span(const std::span<U, N>& s) noexcept;
constexpr span(const span& other) noexcept = default;
```

It begin size_type

```
template<class It, class End>
constexpr span(It first, End last);
```

```
template<class R>
constexpr span(R && r);
```

https://wg21.link/p1394

Notable functions

```
constexpr reference front() const;

constexpr reference back() const;

constexpr reference operator[](size_type idx) const;

constexpr pointer data() const noexcept;
```

Notable functions

```
constexpr size_type size_bytes() const noexcept
{
  return size() * sizeof(element_type);
}

template<class T, std::size_t N>
auto as_bytes(std::span<T, N> s) noexcept
{
  return std::span(reinterpret_cast<const std::byte*>(s.data()), s.size_bytes());
}

template<class T, std::size_t N>
auto as_writable_bytes(std::span<T, N> s) noexcept
{
  return std::span(reinterpret_cast<std::byte*>(s.data()), s.size_bytes());
}
```

Subviews

```
template<size_t Count>
constexpr span<element_type, Count> first() const;

constexpr span<element_type, std::dynamic_extent> first(size_t Count) const;
```

```
template<size_t Count>
constexpr span<element_type, Count> last() const;

constexpr span<element_type, std::dynamic_extent> last(size_t Count) const;
```

```
template<size_t Offset, size_t Count = std::dynamic_extent>
constexpr span<element_type, CountOrDiff> subspan() const;

constexpr std::span<element_type, std::dynamic_extent> subspan(
  size_t Offset, size_t Count = std::dynamic_extent) const;
```

Usability Enhancements for `std::span`

wg21.link/p1024

- Add `front()` and `back()` member functions
  - improve consistency with standard library containers
- Mark `empty()` as `[[nodiscard]]`
- Remove `operator()`
  - vestigial traces from the `array_view` multidimensional genesis
- *Structured bindings* support for fixed-size spans
  - `std::get<N>()`
  - `tuple_element / tuple_size`

WWSD

What Would Stepanov Do?

Should Span be Regular?

wg21.link/p1085

Tony Van Eerd

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

- *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**)

https://herbsutter.com/2018/11/13/trip-report-fall-iso-c-standards-meeting-san-diego/

Should Span be Regular?

wg21.link/p1085

Tony Van Eerd

- `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***

https://herbsutter.com/2018/11/13/trip-report-fall-iso-c-standards-meeting-san-diego/

Should Span be Regular?

wg21.link/p1085 **Tony Van Eerd**

- **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

https://herbsutter.com/2018/11/13/trip-report-fall-iso-c-standards-meeting-san-diego/

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**

https://cor3ntin.github.io/posts/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==`

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

Have reference semantics,

but without the “magic” that can make references safer

(for example *lifetime extension*)

`std::string_view` cheatsheet

Lifetime with std::string_view (C++17)

std::string_view isn't a drop-in replacement
for const std::string&

```
std::string str() {
  return std::string("long_string_helps_to_detect_issues");
}
```

```
const std::string& s = str();
std::cout << s << '\n';
```

lifetime extended
prints the correct result

```
std::string_view sv = str();
std::cout << sv << '\n';
```

lifetime not extended
prints nonsense

const lvalue reference binds to rvalue and provides lifetime extension. But there is no lifetime extension for std:string_view.

For short strings this issue might be hard to detect due to short string optimization (SSO). The problem becomes obvious with longer (dynamically allocated) strings.

@walletfox

`clang-tidy` `bugprone-dangling-handle`

" Detect dangling references in value handles like `std::string_view`

These dangling references can be a result of constructing handles from ***temporary*** values, where the temporary is destroyed **soon** after the handle is created.

`std::span` 👉

Options:

`HandleClasses`

A semicolon-separated list of class names that should be treated as handles. By default only `std::string_view` is considered.

https://clang.llvm.org/extra/clang-tidy/checks/bugprone-dangling-handle.html

Lifetime profile v1.0

Lifetime safety: Preventing common dangling

This is important because it turns out to be ***easy*** to convert **[by design]**
a `std::string` to a `std::string_view`,
or a `std::vector/array` to a `std::span`,
so that dangling is almost the default behavior.

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Lifetime profile v1.0

Lifetime safety: Preventing common dangling

```
void example()
{
  std::string_view sv = std::string("dangling"); // A
  std::cout << sv;              // ERROR (lifetime.3): ‘sv’ was invalidated when
}                               // temporary was destroyed (line A)
```

clang -Wlifetime

Experimental

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Lifetime safety: Preventing common dangling

NEW

`[-Wdangling-gsl]` **diagnosed <u>by default</u> in Clang 10**

warning: initializing pointer member to point to a temporary object whose lifetime is shorter than the lifetime of the constructed object

```
void example()
{
  std::string_view sv = std::string("dangling");
                  // warning: object backing the pointer will be destroyed
                  // at the end of the full-expression [-Wdangling-gsl]
  std::cout << sv;
}
```

https://clang.llvm.org/docs/DiagnosticsReference.html#wdangling-gsl

NEW

Lifetime safety: Preventing common dangling

`[-Wdangling-gsl]` **diagnosed by default in Clang 10**

warning: initializing pointer member to point to a temporary object whose lifetime is shorter than the lifetime of the constructed object

```
void example()
{
  std::span sp = std::vector{1,2,3,4};  // warning: 🤞 WIP... (PR)

  for (auto e : sp)
    std::cout << e << " ";
}
```

https://clang.llvm.org/docs/DiagnosticsReference.html#wdangling-gsl

Simple rules for borrow types

Borrow types must appear ***only as function parameters*** or ***for-loop control variables***

We can make an **exception** for function ***return types***:

- a function may have a borrow type as its return type
  (the function must be explicitly `[annotated]` as returning a potentially dangling reference)

- the result returned ***must not be stored*** into any named variable,
  except passed along to a function parameter or for-loop control variable

Say What You Mean

If you decide to make an **exception** to these best practices,
strongly consider `explicitly annotating` your intent in code.

Custom attributes ?

😜

```
[[magic]]

[[trust_me_on_this_one]]

[[i_am_very_sorry]]

[[it_works_on_my_machine]]

[[beware_of_dangling_reference]]
```

https://en.cppreference.com/w/cpp/language/attributes

Credit: **Ólafur Waage**
@olafurw

What about compiler support ?

C++20

`std::span`

#godboltallthethings

GODBOLT ALL THE THINGS !!!

https://godbolt.org

COMPILER EXPLORER

Compiler Support

https://godbolt.org/z/FRHiPR

November, 2019

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

Span Evolution

Initial std::span spec	wg21.link/p0122	Clang libc++ **7.0**
Remove comparison operators of std::span	wg21.link/p1085	Clang libc++ **8.0**
Usability enhancements for std::span	wg21.link/p1024	Clang libc++ **9.0**
std::ssize() and unsigned extent for std::span	wg21.link/p1227	Clang libc++ **9.0**

Span Evolution (cont)

Belfast, Nov 2019

✅ https://wg21.link/p1394 — Range constructor for `std::span`

⛔ https://wg21.link/p1391 — Range constructor for `std::string_view` (Bonus)

❓ https://wg21.link/p448 — A `strstream` replacement using `span<charT>` as buffer

WIP...

github.com/cplusplus/papers/issues?q=span

Can't Wait ?

Want an implementation of std::span to match the **C++20** CD ?

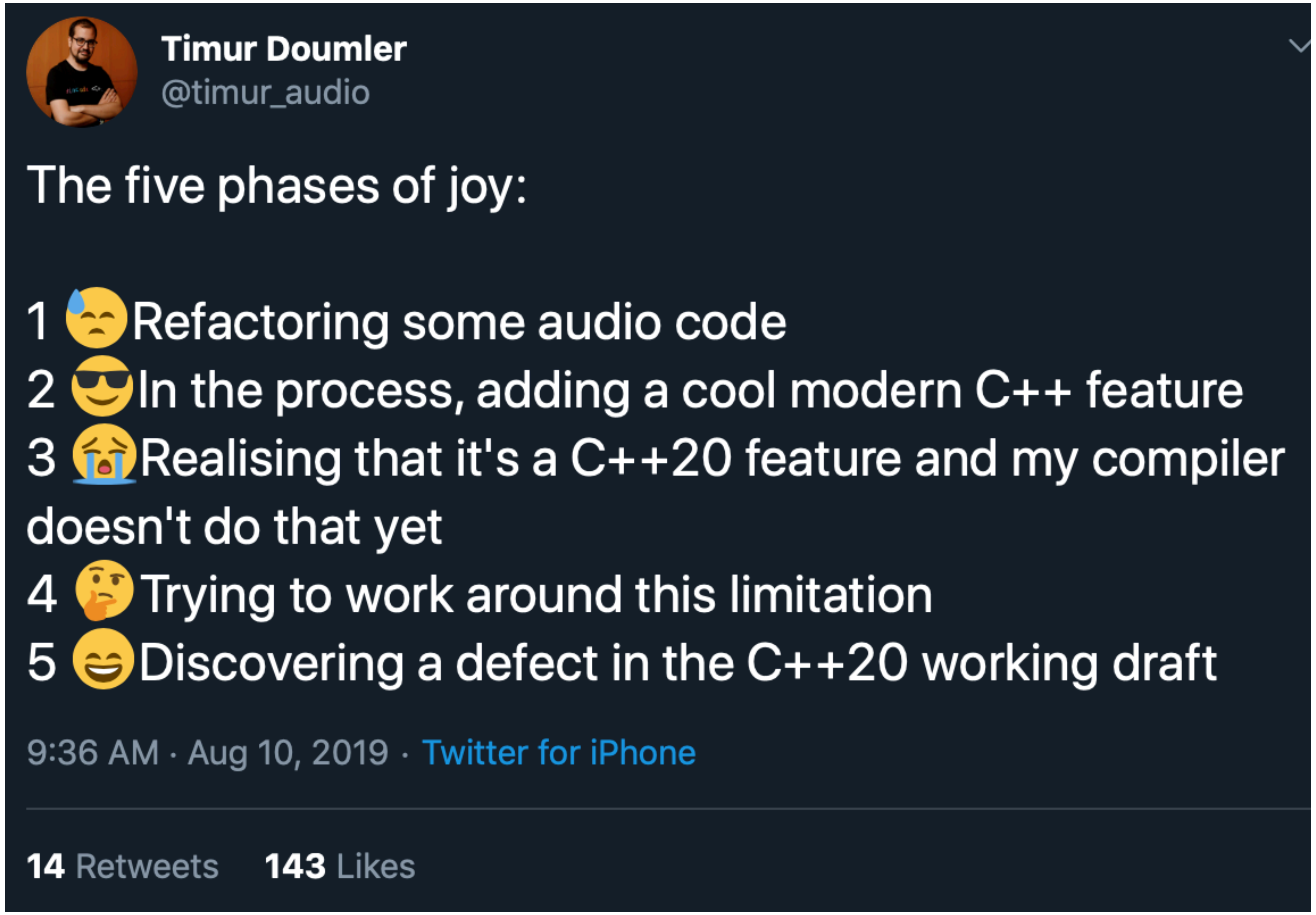
`Clang libc++ 9.0`

https://github.com/tcbrindle/span

by Tristan Brindle

github.com/chromium/chromium/base/containers/span.h

The Five Phases Of Joy


Timur Doumler
@timur_audio

The five phases of joy:

1 😓Refactoring some audio code
2 😎In the process, adding a cool modern C++ feature
3 😭Realising that it's a C++20 feature and my compiler doesn't do that yet
4 🤔Trying to work around this limitation
5 😄Discovering a defect in the C++20 working draft

9:36 AM · Aug 10, 2019 · Twitter for iPhone

14 Retweets **143** Likes

Can you guess what was the C++20 feature ?

`std::span`

lacks a feature test macro

https://twitter.com/timur_audio/status/1160092474259443712?s=21

The Five Phases Of Joy

`std::span` **lacks a feature test macro**

Can't you use `__has_include` **for that?**
new header: `<span>`

libc++ always has all the headers it's implemented,
but those headers are empty unless you have the right standard enabled.
So that doesn't work.

Why do I care ?

In case you want to use *another* **span** implementation,
until the *standard* one becomes available (same API)

https://twitter.com/timur_audio/status/1160092474259443712?s=21

Double or Nothing

```
int main(std::span<std::string_view> args);
```

Two of my favorite pet peeves, combined into one glorious disaster

What if the implementation expects a null-terminated string ?
(eg. calling some old system C API)

Beyond `std::span`

C++ 23-26

Possible areas of focus:

- `stride_view`
- `slice_view`
- `sliding_view`
- `cycle_view`
- `chunk_view`

It's all about ranges !

Other Dimensions...

C++ 23-26

`mdspan` A Non-Owning Multidimensional Array Reference wg21.link/p0009

`mdarray` An Owning Multidimensional Array Analog of mdspan wg21.link/p1684

#defining data layout in memory

Hear more about it:

https://cppcast.com/bryce-lelbach-mdspan/

Early implementation by **David Hollman**:

https://github.com/kokkos/mdspan

HP computing, graphics

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`

https://github.com/cplusplus/LEWG/blob/master/library-design-guidelines.md

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

Fin

A Short Life span< >
For a Regular Mess

November 21, 2019
Wrocław

@ciura_victor

Victor Ciura
Principal Engineer
CAPHYON

📖 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/
RangeOf: A better span
https://cor3ntin.github.io/posts/rangeof/

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

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

span: the best span
https://brevzin.github.io/c++/2018/12/03/span-best-span/

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

📖 References I encourage you to study

Sy 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

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

#5: Ordering Algorithms
https://foonathan.net/blog/2018/09/07/three-way-comparison.html