Regular Types
and
Why Do I Care?

April, 2019
Bristol

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
Why Regular types?

Why are we talking about this?
Have we really exhausted all the cool C++ template<> topics 😜 ?

```cpp
auto sum = []<typename T>(T a, T b) {
    return a + b;
};

auto acc = sum(5, 6.3);
```

coming to a C++20 compiler near you...
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

Part 1
youtube.com/watch?v=xTdeZ4MxbKo

Part 2
youtube.com/watch?v=tn7oVNrPM8I
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?

https://github.com/CppCon/CppCon2018/tree/master/Presentations/modern_cpp_api_design_pt_1

https://github.com/CppCon/CppCon2018/tree/master/Presentations/modern_cpp_api_design_pt_2

Part 2
youtube.com/watch?v=tn7oVNrPM8I
Let's start with the basics...
A datum is a finite sequence of 0s and 1s
A **value type** is a correspondence between a species (abstract/concrete) and a set of *datums*.
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.
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*.
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.**
Type is a set of values with the same interpretation function and operations on these values.
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
• 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
Where am I going with this?
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

Greatest Common Measure: The Last 2500 Years
Mathematics Really Does Matter

“"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.

Richard Feynman
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

Objectives

- To show deep connections between Programming and Mathematics.
- To recognize well-known mathematical abstractions in regular programming tasks.
- To write reusable, efficient algorithms defined on these abstractions.
- To use a small number of algorithms in a variety of applications.

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.

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

Lectures presented at A9
2012
It all leads up to...
Generic programming depends on the *decomposition* of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined *interfaces*. 
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*.
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.
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**
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?
What Is Your Relationship With C++?

- Full Time
- Part Time
- Student
- It’s complicated

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
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


http://stepanovpapers.com/stl.pdf

Alexander Stepanov: STL and Its Design Principles

https://www.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 as the interface between algorithms and data structures
algorithms are associated with a set of common properties

Eg. \{ +, *, min, max \} => associative operations
=> reorder operands
=> parallelize + reduction (\texttt{std::accumulate})

natural extension of 4,000 years of mathematics

exists a generic algorithm behind every \texttt{while()} or \texttt{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.

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.

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.

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

**Compare Concept**

What are the requirements for a *Compare* type?

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

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

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

🤔

But what kind of ordering relationship is needed 🤔

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Irreflexivity</td>
<td>∀ a, comp(a,a)==false</td>
</tr>
<tr>
<td>Antisymmetry</td>
<td>∀ a, b, if comp(a,b)==true =&gt; comp(b,a)==false</td>
</tr>
<tr>
<td>Transitivity</td>
<td>∀ a, b, c, if comp(a,b)==true and comp(b,c)==true =&gt; comp(a,c)==true</td>
</tr>
</tbody>
</table>
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 strcmp(s1.c_str(), s2.c_str()) < 0;
});
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 \}
\[ x_1 < x_2; \; y_1 > y_2; \]
\[ x_1 < x_3; \; y_1 > y_3; \]
\[ x_2 < x_3; \; y_2 < y_3; \]

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

=>

P2 and P1 are unordered \( (P2 ? P1) \) | \( \text{comp}(P2,P1)==false \) && \( \text{comp}(P1,P2)==false \)
P1 and P3 are unordered \( (P1 ? P3) \) | \( \text{comp}(P1,P3)==false \) && \( \text{comp}(P3,P1)==false \)
P2 and P3 are ordered \( (P2 < P3) \) | \( \text{comp}(P2,P3)==true \) && \( \text{comp}(P3,P2)==false \)
Definition:

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

```cpp
class Point {
public:
    int x, y;
}

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:

\[
equiv(a,b) : \text{comp}(a,b)==\text{false} \land \text{comp}(b,a)==\text{false}
\]
### Strict weak ordering

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

<table>
<thead>
<tr>
<th>Irreflexivity</th>
<th>∀ a, comp(a,a)==false</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antisymmetry</td>
<td>∀ a, b, if comp(a,b)==true =&gt; comp(b,a)==false</td>
</tr>
<tr>
<td>Transitivity</td>
<td>∀ a, b, c, if comp(a,b)==true and comp(b,c)==true =&gt; comp(a,c)==true</td>
</tr>
<tr>
<td>Transitivity of equivalence</td>
<td>∀ a, b, c, if equiv(a,b)==true and equiv(b,c)==true =&gt; equiv(a,c)==true</td>
</tr>
</tbody>
</table>

where:

\[
\text{equiv}(a,b) : \text{comp}(a,b)==false \&\& \text{comp}(b,a)==false
\]
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
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?
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

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

http://wg21.link/p0898

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Irreflexivity</strong></td>
<td>∀ a, (a &lt; a)==false</td>
</tr>
<tr>
<td><strong>Antisymmetry</strong></td>
<td>∀ a, b, if (a &lt; b)==true =&gt; (b &lt; a)==false</td>
</tr>
<tr>
<td><strong>Transitivity</strong></td>
<td>∀ a, b, c, if (a &lt; b)==true and (b &lt; c)==true =&gt; (a &lt; c)==true</td>
</tr>
<tr>
<td><strong>Transitivity of equivalence</strong></td>
<td>∀ a, b, c, if equiv(a,b)==true and equiv(b,c)==true =&gt; equiv(a,c)==true</td>
</tr>
</tbody>
</table>

where:

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

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

<table>
<thead>
<tr>
<th>Reflexivity</th>
<th>∀ a, (a == a) == true</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetry</td>
<td>∀ a, b, if (a == b) == true =&gt; (b == a) == true</td>
</tr>
<tr>
<td>Transitivity</td>
<td>∀ a, b, c, if (a == b) == true and (b == c) == true =&gt; (a == c) == true</td>
</tr>
</tbody>
</table>

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

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 :(
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](http://stepanovpapers.com/DeSt98.pdf)
Mandatory Slide

Gauging the audience...

C++98/03  C++11  C++14  C++17
Three-way comparison

Bringing consistent comparison operations...

operator $\llneq$

\[(a \llneq b) < 0 \text{ if } a < b\]
\[(a \llneq b) > 0 \text{ if } a > b\]
\[(a \llneq b) == 0 \text{ if } a \text{ and } b \text{ are equal/equivalent}\]
C++20 ✓ Three-way comparison

The comparison categories for: operator `<=>

- weak_equality
- partial_ordering
- weak_ordering
- strong_ordering
- strong_equality

It's all about relation strength 💪
Three-way comparison

San Diego ISO C++ Committee Meeting
(November 2018)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/#mailing2018-10

11 papers to fix
operator<=>
Three-way comparison

San Diego ISO C++ Committee Meeting
(November 2018)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/#mailing2018-10

Performance Impacts on Using <=> for Equality

https://wg21.link/p1190
https://wg21.link/p1185
Three-way comparison

San Diego ISO C++ Committee Meeting
(November 2018)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/#mailing2018-10

When do you actually use <=> ?
https://wg21.link/p1186

<=> in generic code!
Three-way comparison

San Diego ISO C++ Committee Meeting
(November 2018)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/#mailing2018-10

Default Ordering

https://wg21.link/p0891
Three-way comparison

San Diego ISO C++ Committee Meeting
(November 2018)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/#mailing2018-10

Effect of operator<=> on the C++ Standard Library

https://wg21.link/p0790
Three-way comparison

Wish list for: `operator<=>`

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

```cpp
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...
San Diego ISO C++ Committee Meeting (November 2018)

Titus Winters
@TitusWinters

In other news, yesterday's developments on C++'s operator\(<=\) are going to force me to write an essay, "Type design over time: why you can't safely infer semantics from syntax."

7:37 AM - 9 Nov 2018

2 Retweets 11 Likes
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 waaaaaay less error checking code**

Do you see where this is going?
Using `std::optional` as *vocabulary type* allows us to simplify code and compose functions easily.

### The `M` word

`map()` / `and_then()` / `or_else()`

chaining

But, wait...

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

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

- rebind
- shallow compare
- operator==
  - over by dead body!
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

https://www.youtube.com/watch?v=xwP4YCP_0q0
std::string_view

Do you recognize this?
std::string_view

Brunel's SS Great Britain
std::string_view

Brunel's SS Great Britain
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/](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/
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.
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  \texttt{std::span<T>}

Think "\texttt{array_view}" as in \texttt{std::string_view},
but \texttt{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==
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
http://stepanovpapers.com/DeSt98.pdf

Alexander Stepanov
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

**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
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!

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

#5: Ordering Algorithms
https://foonathan.net/blog/2018/09/07/three-way-comparison.html
Regular Types and Why Do I Care?

April, 2019
Bristol

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](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