Adobe

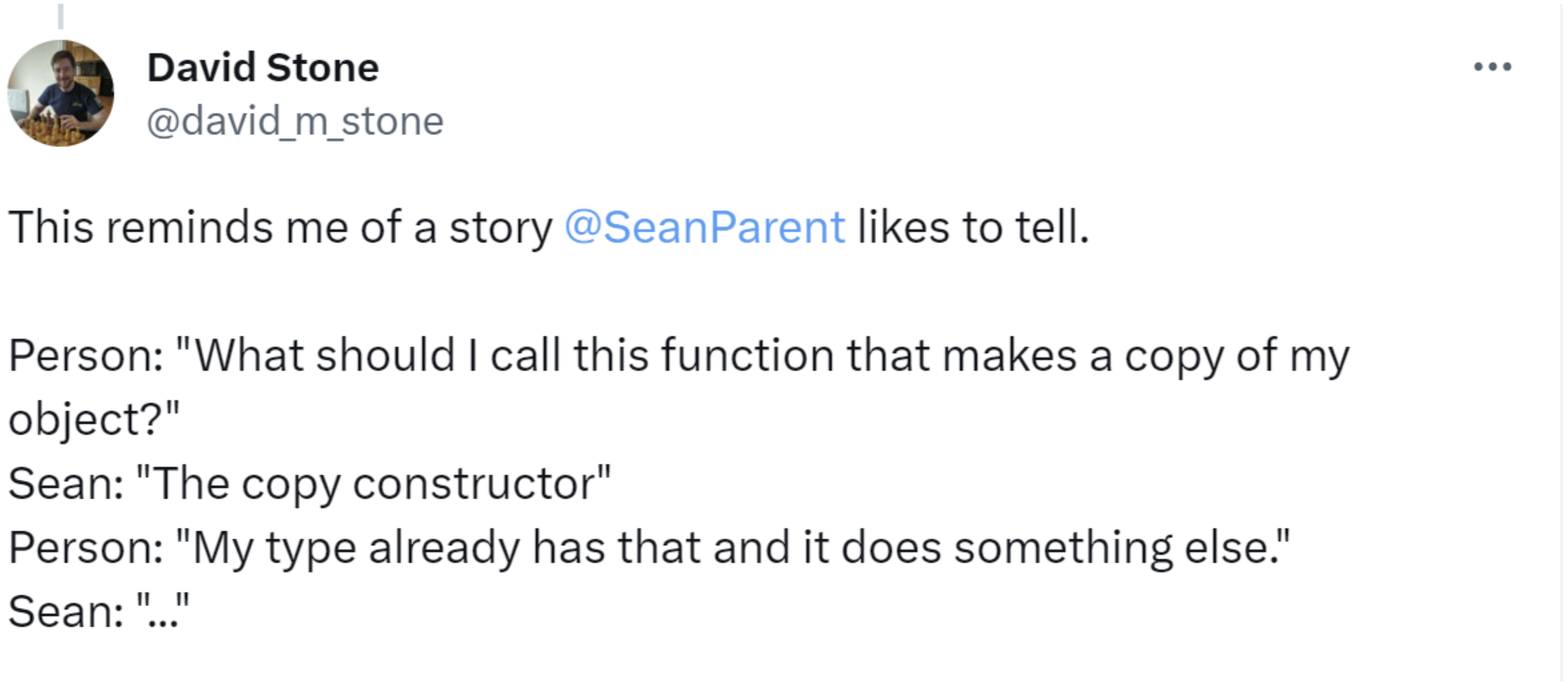
std::simd types should be regular

P2892R0

David Sankel | Adobe's Software Technology Lab

Joe Jevnik | Jump Trading

June 2023, WG21 Varna Meeting



David Stone
@david_m_stone

This reminds me of a story @SeanParent likes to tell.

Person: "What should I call this function that makes a copy of my object?"
Sean: "The copy constructor"
Person: "My type already has that and it does something else."
Sean: "..."

This Presentation

- Background
- Arguments for mask-returning == operator
- Our position
- Answers to objections
- Alternatives considered

Background

- std::simd types represent multiple numeric values
- Arithmetic operators are element-wise. This isn’t contested
- Current proposal has comparison operations element-wise as well. That is the subject of this discussion

Arguments for mask-returning == operator

Mask-returning == is consistent with the other SIMD operators

```
std::simd x, y, z;
//…
z = x + y;
y = z * x;
auto mask = y == z;
```

Bool-returning == is easy to misuse (courtesy Zach Laine/Lane)

```
std::simd x, y, z;
// …
auto result = (x == y) * z; // result is all 1s or 0s. Not the author’s intent
```

std::simd is not a value semantic type

It is in a different category. It is parallel value semantic.

You never want bool-returning == with SIMD

- It is always the wrong operation
- In real code you never see this used
- Therefore, we should use the == operator to be something useful

Every SIMD library out there does this

- We should standardize existing practice
- Users will reject this library if we make it unfamiliar

std::valarray

- std::valarray's == operator doesn't return bool
- This is existing standard library convention we should follow

Our Position

The meaning of == is taken. It should not be contradicted

1. T a = b; assert(a==b);
2. T a; a = b; ⇔ T a = b;
3. T a = c; T b = c; a = d; assert(b==c)
4. T a = c; T b = c; zap(a); assert(b==c && a!=b) where zap always changes its operand’s value.

-- James C. Dehnert and Alexander Stepanov. 1998. Fundamentals of Generic Programming.

http://stepanovpapers.com/DeSt98.pdf

Value semantic types should be regular

- Value semantic types represent mathematical values
- Regular rendering of value semantic types enables generic programming
  - Creation and use of general-purpose algorithms
  - Consider operator=='s usage in std::find

```
class PixelGrid64x64 {
   using Color = std::fixed_size_simd<std::uint32_t, 4>;
   std::array<Color, 64*64> data;
public:
   bool has_black() const {
     auto i = std::find(data.begin(), data.end(), C{}); // ERROR
     return i != data.end();
   }
};
```

Regularity is not only a library convention, it is a language convention

```
struct Pixel {
  std::uint32_t red = 0;
  std::uint32_t green= 0;
  std::uint32_t blue = 0;
  std::uint32_t alpha = 0;
  bool operator==(const Pixel&) const = default;
};
```

```
struct Pixel {
  std::fixed_size_simd<std::uint32_t, 4> value{};
  std::uint32_t red() const;
  std::uint32_t green() const;
  std::uint32_t blue() const;
  std::uint32_t alpha() const;
  bool operator==(const Pixel&) const = default;
};

// Equality comparison of Pixels errors out on first
// usage.
```

std::simd types should be regular

- Value semantic type should be regular as this enables generic programming
- This is something that will have practical impact on our users
- Regular simd types reduces the complexity of the standard library
- The consistency of the whole outweighs familiarity in a particular domain

Answering Objections

Mask-returning == is consistent with the other SIMD operators

```
std::simd x, y, z;
//…
z = x + y;
y = z * x;
auto mask = y == z;
```

- External consistency is traded for internal consistency
- Code remains readable with suggested change

```
std::simd x, y, z;
//…
z = x + y;
y = z * x;
auto mask = mask_equals(y, z);
```

Bool-returning == is easy to misuse (courtesy Zach Laine)

```
std::simd x, y, z;
// …
auto result = (x == y) * z;
// result is all 1s or 0s. Not the author’s intent
```

- Easily mitigated with documentation
- It is not clear this would be a prevalent bug
- People making the opposite assumption would have the same issue

std::simd is not a value semantic type

It is in a different category. It is parallel value semantic.

- This objection is based on a misunderstanding of value semantic types
- Value semantic types represent an entity in the platonic world of mathematical forms
- std::simd objects represent a sequence of numbers
- We don't get to change math because we prefer a particular function identifier ☺

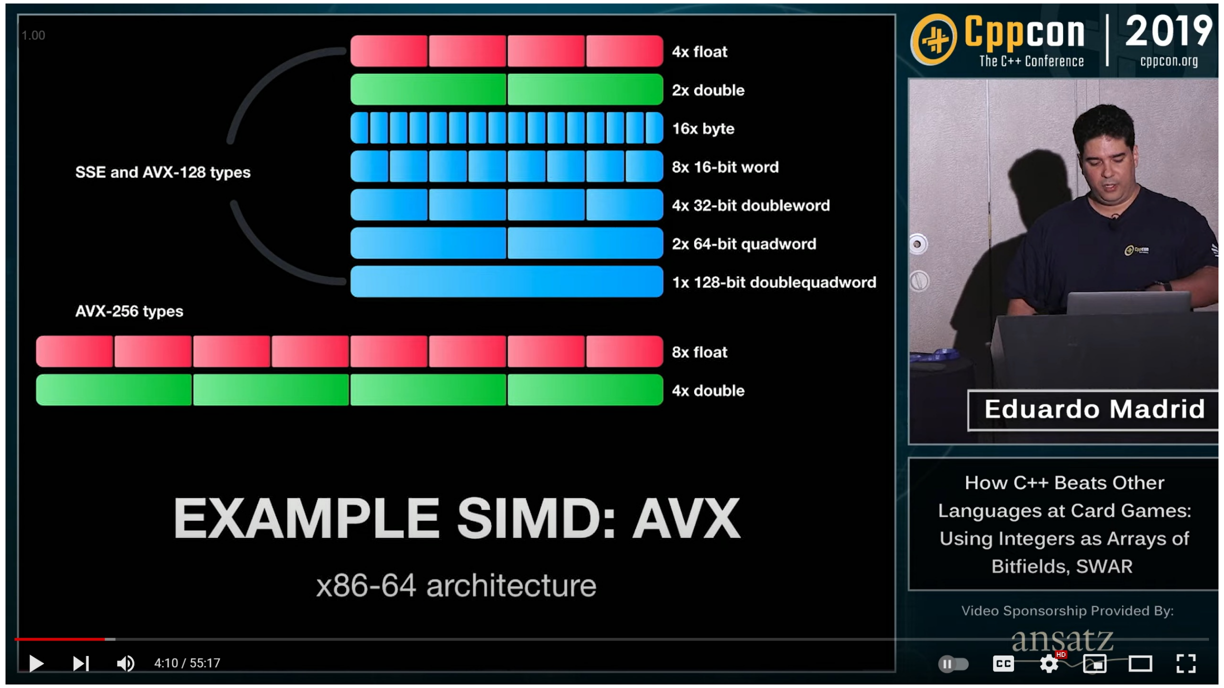
You never want bool-returning == with SIMD

- It is always the wrong operation
- In real code you never see this used
- Therefore, we should use the == operator to be something useful

- Unit testing is a good practice
- Comparing results of simd operations using equality is a straightforward way to test
- SIMD is not only used for bulk data processing. It can be used for fixed-width vectors, color representations, and other things. Equality makes sense in these domains.

Every SIMD library out there does this

- We should standardize existing practice
- Users will reject this library if we make it unfamiliar



- Not every SIMD library does this
- Eduardo Madrid's SWAR library is a good counter-example
- See paper for other examples
- Standardizing existing practice in general would result in an inconsistent and inferior standard library (consider the STL)

std::valarray

- std::valarray's == operator doesn't return bool
- This is existing standard library convention we should follow

- std::valarray was a failure
- It is not a good standard library to look at for best practices for this and other reasons

Alternatives Considered

Make masks convertible to bool

- Idea: make operator== return a mask and have that mask convertible to bool
  - Get the best of both worlds?
- Concerns
  - Bool conversion for == result is “all of”, but bool conversion for != result is “any of”
  - May introduce too much complexity
  - May result in contradictions

Remove/rename comparison operators for SIMD types

- Benefits
  - Remove the contradiction
  - Remove possibility of run-time errors due to false assumptions
- Drawbacks
  - Missed opportunities to take advantage of regular-type machinery

About the artist

Dan Zucco

London-based 3D art and motion director Dan Zucco creates repeating 2D patterns and brings them to life as 3D animated loops. Inspired by architecture, music, modern art, and generative design, he often starts in Adobe Illustrator and builds his animations using Adobe After Effects and Cinema 4D. Zucco's objective for this piece was to create a geometric design that felt like it could have an infinite number of arrangements.

Made with

Ai Adobe Illustrator   Ae Adobe After Effects

Adobe
Bē
Artwork by Dan Zucco