# `std::simd` Types Should be Regular


| | |
|---|---|
| Document #: | P2892R0 |
| Date: | 2023-05-18 |
| Project: | Programming Language C++ |
| Audience: | Library Evolution |


| | |
|---|---|
| Reply-to: | Joe Jevnik<br><joejev@gmail.com><br>David Sankel<br><dsankel@adobe.com> |


## 1 Abstract


SIMD Types from the Parallelism TS are proposed for adoption in C++26. Deviating from standard value-semantic types, the `==` operator for SIMD types is not regular and returns a mask instead of `bool`. This inconsistency detrimentally effects user experience. We instead recommend that `==` and related operators return `bool` with masked variants provided as free functions.


**Before**

```
using uint32_4v = std::fixed_size_simd<
                            std::uint32_t, 4>;

class Color {
public:
  bool operator==(const Color &) const =
                                      default;
private:
  uint32_4v data_;
};

void f() {
  Color a, b;
  // ...
  if( a == b )    // ERROR: use of deleted
                  // function
                  // 'Color::operator=='
    //...
}
```

**After**

```
using uint32_4v = std::fixed_size_simd<
                            std::uint32_t, 4>;

class Color {
public:
  bool operator==(const Color &) const =
                                      default;
private:
  uint32_4v data_;
};

void f() {
  Color a, b;
  // ...
  if( a == b )    // OKAY


    //...
}
```

## 2 Introduction

The Parallelism TS 2, [N4808], includes data-parallel types that enable the portable utilization of "single instruction, multiple data" (SIMD) CPU capabilities. Matthias Kretz proposes standardizing these types with some modifications (hereinafter referred to as `simd` types) in [P1928R3], "Merge Data-Parallel Types From the Parallelism TS 2". This initiative prompted working sessions at C++Now 2023 that analyzed `simd` types from a usability perspective.

Notably, the behavior and return type of `==` for `simd` types was discussed. As currently defined, `==` performs

element-wise equality and returns a `simd_mask` of the result. Given `simd` objects `v1` and `v2` with values `1,2,3,4` and `1,4,3,6`, `v1 == v2` has value `1,0,1,0`. This is consistent with other `simd` operators such as `+`, `-`, `*`, and `/`.

While this behavior appears promising, it runs contrary to long established convention with value semantic types in C++. The departure results in increased complexity and unexpected behavior in user code. After lengthy discussions, a poll was taken at C++Now indicating a strong preference for conventional `==` semantics for `simd` types: "operator== should return a scalar bool" — SF: 9, WF: 10, N: 0, A: 1, SA: 0

The following sections outline the arguments leading to this result by reviewing value semantics, considering its application to `std::simd` types, and looking at existing practices.

# 3 Value semantics and regularity

Objects of value types correspond to mathematical entities, enabling equational reasoning. This correspondence is the basis of generic programming, which allows algorithms to be composed and reused without modification in different scenarios. For a full treatment, see [EOP].

To take full advantage of value types, they need to be consistently rendered in code. In C++, properly encoded value types are called regular. [DeSt98] calls out the following relationships for the copy constructor, assignment, and equality:

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.

Due to the consistent rendering, regular types are easily composed into more sophisticated regular types. In the following snippet, `S` is, by construction, a regular type if `T1` and `T2` are:

```
struct S {
  T1 o1;
  T2 o2;
  bool operator==(const S&) const = default;
}
```

The standard library also heavily depends on value semantic types being regular. `std::find(first, last, value)`, for example, uses the `==` operator to search for `value` in the `[first,last)` range.

Due to readability and productivity benefits, regular rendering of value semantic types are a best practice for C++ as reflected by several C++ Core Guidelines rules (See [CppCoreGuidelines]):

— C.11 "Regular types are easier to understand and reason about than types that are not regular…Concrete classes without assignment and equality can be defined, but they are (and should be) rare."
— C.61 "After `x=y`, we should have `x == y`"
— C.160 "Define operators primarily to mimic conventional usage"

# 4 std::simd, value semantics, and regularity

`std::simd` types *are* value types. A `std::simd` object corresponds to a sequence of values such as `255,0,0,0`.[1] The current implementation of `std::simd` objects is not regular, however, due to semantics of their comparison operators.

Would this lack of regularity be a problem in practice? Say a user has written the following class and would like to accelerate it using `std::simd`:

[1] Formally, with denotational semantics we say $\mu[\![$ `std::simd<` $T$`, std::simd::abi::fixed_size<`$N$`>` $]\!] = \mu[\![T]\!] \times \mu[\![T]\!] \cdots \mu[\![T]\!]$ ($\mu[\![N]\!]$ times)

```
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;
};
```

A straight forward modification would be as follows:

```
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;
};
```

This code will compile and work *until* the `==` operator is called. In that case an error message like the following is emitted:

```
<source>: In function 'void f()':
<source>:43:14: error: use of deleted function 'bool Pixel::operator==(const Pixel&) const'
   43 |     if( a == b );
      |              ^
<source>:22:10: note: 'bool Pixel::operator==(const Pixel&) const' is implicitly deleted because the defa
   22 |     bool operator==(const Pixel &) const = default;
      |          ^~~~~~~~
<source>:22:10: error: could not convert 'std::operator==(((const Pixel*)this)->Pixel::data, <anonymous>.
   22 |     bool operator==(const Pixel &) const = default;
      |          ^~~~~~~~
      |          |
      |          std::simd<unsigned int, std::simd_abi::_Fixed<4> >::mask_type {aka std::simd_mask<unsign
ASM generation compiler returned: 1
<source>: In function 'void f()':
<source>:43:14: error: use of deleted function 'bool Pixel::operator==(const Pixel&) const'
   43 |     if( a == b );
      |              ^
<source>:22:10: note: 'bool Pixel::operator==(const Pixel&) const' is implicitly deleted because the defa
   22 |     bool operator==(const Pixel &) const = default;
      |          ^~~~~~~~
<source>:22:10: error: could not convert 'std::operator==(((const Pixel*)this)->Pixel::data, <anonymous>.
   22 |     bool operator==(const Pixel &) const = default;
      |          ^~~~~~~~
      |          |
      |          std::simd<unsigned int, std::simd_abi::_Fixed<4> >::mask_type {aka std::simd_mask<unsign
Execution build compiler returned: 1
```

Indeed, element-wise comparisons undo the benefits of defaulted `==` and `<=>` operators. Users are forced to write trivial implementations of these functions for every structure that embeds a `std::simd` object.

Consider the following snippet where a `std::array` represents a 64x64 pixel grid. Each element of the array is a color represented by a `std::fixed_size_simd<std::uint32_t, 4>`.

```
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();
  }
};
```

The compilation error produced by the `std::find` algorithm is, again, due to the non-`bool` result of the `==` operator. The user is forced to use a custom comparison to get this code to compile.

## 5 Existing practice in SIMD libraries

Many libraries in the wild provide SIMD types. Some explicitly wrap a single SIMD register, like VCL or xsimd, while others express larger arrays or mathematical objects. The following table associates some open source libraries with their choice with respect to comparison operators.

| Library | Comparison Behavior |
|---|---|
| Armadillo | Element-wise |
| Blaze | Regular |
| EVE | Element-wise |
| Eigen::Array | Element-wise |
| Eigen::{Matrix, RowVector, Vector} | Regular |
| Fastor | Element-wise |
| VCL | Element-wise |
| xsimd | Element-wise |
| xtensor | Regular `operator==` and `operator!=`; rest are element-wise[2] |

These libraries exist broadly in two categories: wrappers around SIMD operations and DSLs (domain specific languages) for doing math and multidimensional array operations. For libraries that are conceptually DSLs: users are unlikely to switch away from these high level tools. If end users want to use a DSL, they will be shielded from the particular syntactic choices used by `std::simd` and this is an irrelevant decision for them.

For users who want to introduce a small amount of data parallelism in their code, or are implementers of a DSL, having regular comparison will fit more naturally into rest of the C++ language for the reasons stated above. Non-DSL users will still see a considerable benefit with respect to conciseness and clarity over using platform specific intrinsic operations, even with value-semantics. For example if we compare one of the algorithms in [P1928R3] which used element-wise `operator>`, the new proposal is not considerably more verbose nor unclear:

```
 using floatv = std::simd<float>
 using intv = std::rebind_simd_t<int, floatv>;

 int count_positive(const std::vector<floatv>& xs) {
     // simplify generated assembly:
     if (x.size() == 0) std::unreachable();
     intv counter = {};
     for (std::simd x : xs) {
-        counter += x > 0;
```

[2] This option is only used in xtensor, not even the related project xsimd. The inconsistency seems more likely to cause bugs than selecting either behavior by itself.

```
+         counter += greater(x, 0);
      }
      return reduce(counter);
  }
```

### 5.1 Existing practice in the standard

While the standard does not yet include SIMD types, it has several that represent sequences of values. These are `std::bitset`, `std::array`, `std::vector`, `std::list`, and `std::valarray`. All of these have regular `==` operators with the exception of `std::valarray`, which is rarely used due to its poor design.[3]

## 6 Conclusion

Value semantic types should be regular in C++ and `std::simd` is no exception. We've looked at the benefits of regularity, illustrated the impact it has on `std::simd`, and surveyed existing practice.

## 7 Acknowledgments

Thanks to Jeff Garland and all who participated in C++Now 2023's library in a week for engaging in these interesting discussions. Thanks to Darius Neațu and Dave Abrahams for reviewing drafts of this document. A special thanks to Zach Lane for a never-ending supply of friendly opposition.

## 8 References

[CppCoreGuidelines] Bjarne Stroustrup and Herb Sutter. 2020. C++ Core Guidelines.
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[CppStandardLibrary] Nicolai M. Josuttis. 2012. The C++ Standard Library: A Tutorial and Reference.

[DeSt98] James C. Dehnert and Alexander Stepanov. 1998. Fundamentals of Generic Programming.
http://stepanovpapers.com/DeSt98.pdf

[EOP] Alexander Stepanov and Paul McJones. 2009. Elements of Programming.
http://elementsofprogramming.com/eop.pdf

[N4808] Hoberock Jared. 2019. Working Draft, C++ Extensions for Parallelism Version 2.
http://wg21.link/N4808

[P1928R3] Matthias Kretz. 2023. Merge Data-Parallel Types From the Parallelism TS 2.
http://wg21.link/P1928R3

[3] See [CppStandardLibrary] for a historical note on valarray's unfortunate design.