

weak_equality considered harmful

Document number: P1307

Date: 2018-10-08

Audience: EWG

Reply-to: Tony Van Eerd. regular at forecode.com

If I had more time, I would have written a shorter letter. - Blaise Pascal (and others)

I hope that most programmers will learn the fundamental semantic properties of fundamental operations. What does assignment mean? What does equality mean? How to construct data structures.

At present C++ is the best vehicle for this style of programming.

-- Alex Stepanov

Synopsis (*and about that title*)

This paper will explain why `weak_equality` should not lead to the generation of `==` (nor `!=`). (See P0515 for in the introduction of `weak_equality` and `<=>` into C++.)

The title of this paper is obviously "click-bait" like the goto paper it alludes to, but it is also accurate - an `==` operator that is "weakly equal" is an oxymoron and is *harmful* to quality software; it will break std algorithms and goes against Concepts, the Palo Alto paper (N3351), the Lakos "value paper" (N2479), Stepanov and the *Elements of Programming* ("EoP") book, and the very fundamentals of C++.

P.S. same goes for generating `==` from `partial_ordering` or `weak_ordering`. Also not good.

Background

To ensure everyone clearly understands the terms `strong_equality` and `weak_equality` from P0515.

The *strong* in `strong_equality` refers to *substitutability*, the idea that $x == y$ implies $f(x) == f(y)$, for all (Regular) f if f only accesses the *salient* attributes of x and y .

What is *salient*? See Lakos. Either N2479, or see *actual John Lakos*. But basically "salient" means "the important parts" and the parts that are *guaranteed* to be copied/moved via construction and assignment. The parts that define the *value* of the type. For example, the elements of a vector are

salient, and form the value of the vector, but capacity is not salient.

(It sounds like "salient" is arbitrary. To an extent, it is. It must be defined by each particular class. But whatever the class defines as salient, it is important for it to be consistent about it - copy, move, assign, and ==, should all agree. Again, see Lakos. And later examples in this paper.)

The *weak* in `weak_equality` means, unsurprisingly, *not strong*. ie a weak == does NOT imply substitutability. As noted in P0515 it still forms an *equivalence relation* in the mathematical sense (ie reflexive, symmetric, transitive - see Wikipedia), but not "equal" in the mathematical or substitutable sense. In fact, not "equal" in the *equal* (common English) sense. Like 2 and 7 are *equivalent* mod 5, but 2 and 7 are not equal, they do not represent the same *value*.

So fundamentally, `weak_equality` generates an == such that == does not mean "equal". Thus it is an oxymoron. :-)

Whereas, as this paper will explain, `strong_equality` means equality, as it is commonly known.

More on "equal"

What *does* equal mean?

The C++ Working Draft:

There are approximately 700 occurrences of "equal" in the Standard. Besides technical terms like *brace-or-equal-initializers*, the Standard mostly uses "equal" in the English and mathematical sense. When referring to syntactic ==, it will use more specific terms, like "compare equal".

The word "equal" is used in the sense of "`weak_equality`" 0 times.

The Standard also uses *equivalent* (550+ occurrences) - "Effects: equivalent to..." - ie substitutable - "equivalence of keys" is carefully explained as specific term, to be an equivalence relation (not equality) on keys in map and set. Note that == is not used for key-equivalence.

The Palo Alto Paper (N3351)

There is an entire section: **3.1.1 Equality**

First sentence:

"Reasoning about computer programs is facilitated by *equational reasoning*, which allows us to substitute equals for equals."

Next:

"If two values represent the same entity, then they are equal."

But what is meant by "same entity"? It is not in Palo Alto. We need to turn to *Elements of*

Programming. For equality, EoP is (unsurprisingly) similar:

"Two values of a value type are equal if and only if they represent the same abstract entity"

But EoP is more thorough. Page 1:

"An *abstract entity* is an individual thing that is eternal and unchangeable,... Blue and 13 are examples of abstract entities"

And Page 2 explains that values represent abstract entities (via datum - sequence of 0s and 1s). The important part being the abstract thing, like 13 - this is the key to "value".

This is also explained similarly in Lakos (N2479) which Palo Alto references in the **Equality** section.

And more directly, Palo Alto, Page 50:

So, EqualityComparable is true if T 1. has == and != with result types that can be converted to bool 2. == compares for true equality 3. == is reflexive, symmetric, and transitive 4. != is the complement of ==

Note the existence of #2, implying that #3 isn't sufficient. "true equality" isn't defined anywhere, but you can look at their definition of eq(), or the definition of equality and the quotes above.

Concepts (in the C++ Working Draft)

The Ranges/Concepts papers (now in the working paper) lean heavily on Palo Alto.

EqualityComparable<T> requires the expected syntactic constraints of ==, but also "is satisfied only if bool(a == b) is true when a is equal to b, and false otherwise". The "equal" in that sentence is "English equal" not "syntactic equal" (else it would be redundant) and is further explained in the [concepts.equality] section as strong equality (substitution).

To be clear: **Concepts and Ranges assume == means strong equality**

Thus some Concept-based std algorithms may break on types with weak equality. (As strong equality is an assumption that an implementation may assume, which algorithms will break may differ per implementation.)

But standard algorithms tend to only need partial ordering?

It was pointed out to me that

Alex Stepanov (or someone quoting him) would likely stomp on the notion of equivalence being problematic. He's retired now but famously stomps of people about mathematics, and he designed STL to use equivalence for associative containers and I've never heard him regret it.

(equivalence there meaning weak equality)

It is important to understand that *algorithms* tend to (and should) set minimal requirements *for that algorithm*, whereas *types* are expected to be useful in a larger set of algorithms, and thus tend to support a super-set of requirements. In particular, *Regular* is the concept that captures all these common and expected requirements.

EoP page 7: "A type is *regular* if and only if its basis includes equality, assignment, destructor, default constructor, copy constructor, total ordering, and underlying type."

Note *total ordering*. Not partial or weak. Even though most of Stepanov's STL only requires partial ordering, a type is expected to have total ordering, (to be clear - the difference between the two is strong equality).

So to answer the Stepanov question, agreed, equivalence is not problematic *for an algorithm*, but Stepanov would find it problematic for a *type*.

Why?

It goes back to Palo Alto's "Reasoning about computer programs is facilitated by *equational reasoning*, which allows us to substitute equals for equals."

Every important optimization technique is affiliated with some abstract property of programming objects. Optimization, after all, is based on our ability to reason about programs and to replace one program with its faster equivalent.

-- Alexander Stepanov, *Notes on Programming*

Another way to consider it is what Stepanov called "optimizing programmers":

The operations we have discussed here, equality and copy, are central because they are used by virtually all programs. They are also critically important because they are the basis of many optimizations... ..Such optimizations include, for example, common subexpression elimination, constant and copy propagation, and loop-invariant code hoisting and sinking. These are routinely applied today by optimizing compilers to operations on values of built-in types. Compilers do not generally apply them to operations on user types because language specifications do not place the restrictions we have described on the operations of those types.

However, users do apply such optimizations by hand. They often do so without thinking because they intuitively expect the conditions to apply.

If they are to produce efficient generic components without seeing the underlying type definitions, they must be able to make the assumptions which allow such optimizations. Our axioms, then, are necessary to allow users to reliably make the optimizations commonly made both by optimizing compilers and by optimizing programmers. Ultimately, we would like compilers to be able to perform such optimizations at a high semantic level as well as they do at the built-in type level.

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

ie we, as programmers, assume equality means substitution. We do this regularly.

The "algorithms" that require strong equality are the everyday lines of code we write.

The motivation *for* `weak_equality`

Why was `weak_equality` proposed in the first place, and is its motivation compelling *enough* to include it in the standard? (spoiler alert: No.)

I see 2 motivations for `weak_equality` in P0515 (the original paper proposing `<=>`)

1. "completeness"
2. `CaseInsensitiveString` (and/or filenames)
3. building on Lawrence Crowl's comparison work (P0474, P0100)

Motivation 1: Completeness - ie it makes the table of the relationship between equality and ordering more complete (and teachable):

```
+-----+-----+
|          | partial_ordering |
| weak_equality +-----+
|          | weak_ordering |
+-----+-----+
| strong_equality | strong_ordering |
+-----+-----+
```

Note however that it is not actually complete - `partial_equality` is missing. ie An equality that is strong where defined, but not defined over the full set of values - ie this could be used for `<=>` over floating point types (ie with strong equality everywhere except NaN). Rust, for example, has the `PartialEq` trait for this.

In fact, strong vs weak (ie "is it substitutable") and partial vs total ("does it cover all values") are orthogonal axes, conflated by P0515.

Motivation 2: `CaseInsensitiveString` (and/or case insensitive filenames)

The original `<=>` paper (P0515) used `CaseInsensitiveString` as a motivating example of a class that might want to use `weak_equality` or `weak_ordering`.

Basically I don't find the example sufficiently motivating. It is an anti-pattern:

I feel that a class like this needs to decide whether case is *salient* or not. A typical litmus test is "if we put a bunch of CISs into a set, would you be surprised when some disappeared?". Alternatively, would the rest of the code (and user-base) be OK if the string was converted to lowercase in the constructor?

- If case is not salient, not important, then equality can ignore it, and it is actually *strong equality* (like `vector` ignores capacity - it is not part of the value).
- Alternatively, if case is salient, make it part of `==`.

Neither case results in a weak `==`.

A weak `==` is *harmful* in that it can lead to simple mistakes, reminiscent of Stepanov's "optimizing programmers":

```
void set(X const & newX)
{
    if (oldX == newX)
        return;

    ... do stuff ...
}
```

A programmer would need to know not to use this pattern with `CaseInsensitiveString`, since "important(?)" case information would be lost.

Now you are free to disagree with me when I say that `CaseInsensitiveString` is a bad class, and that you will use it with a weak equality anyhow. Or that there are examples here and there that might use weak equality.

Fine.

The real question here is not whether we allow you to write bad code. We always do. The question is whether we encourage it, enable it, make it easy.

True, the standard isn't a guideline, but be very clear - the whole `<=>` feature is about making it easier to be more correct. We don't add things to the language without *sufficient motivation*.

C++ as a language does not impose any constraints. You can define your equality operator to do multiplication. But equality should be equality.

-- Alexander Stepanov

Motivation 3: Building on Crowl's P0474 and P0100

Much of the categorization in P0515 is based on Lawrence Crowl's very detailed comparison papers, P0474 and P0100. Lawrence suggested a `weak_equality()` function (which makes sense on types like `CaseInsensitiveString` that have weak or partial ordering), but never suggested promoting that to `==`. His papers are very clear that `==` should always be actual equality. (In fact, an earlier version of his paper called it `weak_equal`, but he (correctly :-)) later changed it to `weak_equality`).

It seems `weak_equality`, and the generation of a `weak ==`, was new in P0515 and did not get enough scrutiny.

Suggested Actions

- do not generate `==` from anything except `strong_ordering` and `strong_equality`
- rename `strong_equality` to `equality`, `weak_equality` to `weak_equality`,
- better, remove `weak_equality` completely, `weak_ordering` as well, as it becomes moot
- consider introducing `partial_equality`, which may actually be useful

References

- *A shorter letter*, <https://quoteinvestigator.com/2012/04/28/shorter-letter/>
- *Goto Considered Harmful* is not the original title
https://en.wikipedia.org/wiki/Considered_harmful
- Dr Dobb's interview with Alex Stepanov, March 1995
<http://stepanovpapers.com/drdoobbs-interview.html>
- *Notes on Programming*, Alexander Stepanov, <http://stepanovpapers.com/notes.pdf>
- *Fundamentals of Generic Programming*, James C. Dehnert and Alexander Stepanov,
<http://stepanovpapers.com/DeSt98.pdf>
- *Elements of Programming* (EoP book), Alexander Stepanov and Paul McJones,
<http://elementsofprogramming.com>
- *A Concept Design for the STL* (N3351, the "Palo Alto paper"), B. Stroustrup and A. Sutton (Editors), <http://wg21.link/N3351>
- *Normative Language to Describe Value Copy Semantics* (N2479), John Lakos,
<http://wg21.link/N2479>
- *Salient*, actual John Lakos, personal communication
- *Consistent comparison*, Herb Sutter, Jens Maurer, Walter E. Brown, <http://wg21.link/P0515>
- *Comparison in C++*, Lawrence Crowl, <http://wg21.link/P0100>
- *Comparison in C++: Basic Facilities*, Lawrence Crowl, <http://wg21.link/P0474>