Rename concepts to standard_case for C++20, while we still can

Document Number: **P1754 R0**
Date: 2019-06-16
Audience: EWG, LEWG, LWG

Reply-to: Herb Sutter (hsutter@microsoft.com)
Casey Carter
Gabriel Dos Reis
Eric Niebler
Bjarne Stroustrup
Andrew Sutton
Ville Voutilainen

1 Motivation: Why we need to do this, and why it's a bug-fix

Before C++20, standard identifiers have always used standard_case, without exception.[1] However, current draft C++20 creates a new inconsistency by making concept names use PascalCase.

Importantly, the standard's consistent use of standard_case has always made it possible for programmers to create a clear delineation between standard names and domain-specific names, by using PascalCase for domain-specific names. The current draft C++20 PascalCase naming scheme for library concepts breaks that by doing a land grab into that swamp, making it murkier than before. The ambiguity is more than just a naming clash – it is about the standard style now conflicting with styles that it didn't conflict with before, which is a readability problem, a mental-model-compartmentalization problem, and a whole host of other problems.

Secondarily, this new taking of PascalCase names does also create potential new ambiguities with user-defined names like `Integral` and `Common` and `OutputRange` and `Boolean` which previously could never conflict with the standard's names. For example, a quick look at *codesearch.isocpp.org* for "`Integral`" shows over 2,200 uses in frameworks, and in domain-specific libraries for chemistry, mathematics, and other domains. In the past these names were safe and known to be immune from clashes with the standard library, even in the presence of `using namespace std`, so despite the current issues with `using namespace` directives, a library could use PascalCase and know it was safe to be used in programs that did `using namespace std` specifically, which is by far the most common using-directive.[2]

Why now: We can still change these names before we publish C++20, but whatever names we ship with C++20 are the ones we will live with for decades.

Why a bug-fix: This could be a NB comment on the CD, if needed. It is a straight renaming, with no technical semantic change and no impact on existing conforming Standard C++ code. There is minor impact on code that uses pre-standard Ranges TS names, which can do a global replace (or macro); note that such code already needs to make other changes to use the standardized version of Ranges.

---

[1] The only normative names in the standard that are not standard_case are MACRO_NAMES, which are rightly visually distinct because they are fundamentally different – they're not just "not identifiers," they're outside the language entirely.

[2] in *codesearch.isocpp.org*'s code corpus, of all 311,000 hits for `using namespace`, over 18% are specifically for `using namespace std`, compared to 3% for all `using namespace boost` which includes subnamespaces `boost::*`.

1.1 Alternatives and objections considered

The following alternatives and objections were considered:

1.1.1 Retain status quo PascalCase, on the principle that it's desirable to make concepts stand out because they're new.

We think this would be a shortsighted choice, because soon they won't be new and then would look different forever. Some of the authors think there are similarities to the rationale used by some C++ developers in the 1990s for naming classes starting with `C`, or templates starting with `T`, and in hindsight we think it is good that the standard never followed those conventions (although individual libraries are free to do so and some are quite happy with them).

1.1.2 Retain status quo PascalCase, because (for example) 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.

We think this concern already exists with `std::is_copy_constructible_v<T>` and `std::CopyConstructible<T>`, because new users don't know that PascalCase means something magical any more than they know that the prefix `is_` and suffix `_v` mean something magical. Novice users will conflate the trait and the concept regardless of the transformation we apply to the words "copy" and "constructible" if both the trait and the concept contain some variation of those words. One of the authors who initially preferred PascalCase concept naming "found that after a while `std::copy_constructible`, `std::is_copy_constructible`, and `std::CopyConstructible` are equally similar, equally different, and if you see any two of them you have to head for *cppreference.com* or equivalent to find out the difference." Finally, in this particular example and others like it of similarly-named concepts and type traits, in the examples we've looked at the difference very minor (here, primarily in explicit copy constructors, which should be rare and are discouraged) and we conjecture they are unlikely to be actually noticed by most users.

1.1.3 Retain status quo PascalCase, on the principle of consistency with standard template parameters (e.g., *`template<class T, auto Size>`*).

We think there is a precedent here, but not with the PascalCase names which are not identifiers, they are only expository and so not subject to standard_case for all actual library names; rather, the "concepts" precedents in those examples are `class` and `auto` which are lowercase. So we think that there is some guiding precedent here, but that it argues in favor of changing concept names to standard_case.

1.1.4 Retain status quo PascalCase, on the principle that concepts are not types, and are thus named differently from standard types.

We think this is a variation of 'make the new things look different' so similar rationale applies. Some of the authors think there are similarities to the rationale used by some C++ developers in the 1990s for naming enums with `E` because they're not classes, or templates with `T` because they're not types (the instantiations are types), and in hindsight we think it is good that the standard never followed those conventions (although individual libraries are free to do so and some are quite happy with them). It is also contrary to the intent of several of the

designers of concepts that concepts are (or should be) like types, and who want to further blur that distinction rather than accentuate it.

1.1.5 Put concepts in a sub-namespace.

We think we should not do this because it is probably ugly (commonly we would type `concept::` before standard concepts) and too late to experiment (we don't have time to verify whether there may be unintended usability consequences with name lookup, such as if users common do/don't `using namespace std::concepts;`).

2 Proposal

This paper proposes that we should continue to follow our standard identifier naming style consistently also for concept names. This is nearly our last opportunity to revisit that before casting the current names in stone in a published standard. We should:

- Rename standard library concept names to standard_case.
- Use a name that is an adjective or abstract noun. (Prefer names like "regular" and "swappable" and "color." Avoid names like "has constructor" and "red.")
- Occasionally, use a "_type" suffix as a concession to avoiding name collisions, usually for very general concepts with very common names.

Notes:

- Concepts that are similar to the existing traits often just drop the "is_" prefix, which feels both nicely consistent and nicely nonconflicting.
- None of the proposed names conflict with existing names.
- It would be nice if the X_with concepts could be merged with their X variants, but that is independent of the name change.

2.1 Impact

No impact on portable standard-conforming code, and low impact on code that uses Ranges TS concepts:

- This proposal has no technical (semantic) impact, it is only renaming.
- No existing portable standard-conforming code is affected because the names are not yet in a published standard.
- Existing code that uses Ranges TS concepts can be updated to use the new names by a global edit or a transient header loaded with macros (e.g., #define CopyConstructible copy_constructible). Note that Ranges TS code already needs to make other changes to use the standardized version of Ranges.

2.2 Examples

Some names are harder to read with PascalCase, notably if they start with the letter I (EYE, not ELL):

```
// status quo
template <std::Integral T> void foo(T);

// proposed
template <std::integral T> void foo(T);
```

Ville Voutilainen notes: “I can instantly see that that’s not a lower-case l after std. Curiously, none of the concepts seem to start with an l, but plenty of them start with an I.” (Note: At least one of the other authors had to read that comment three times to see it was written correctly, which highlights the problem.)

Many names are unchanged except for case and underscores:

```
// status quo
void f(SignedIntegral auto x);

// proposed
void f(signed_integral auto x);
```

Some have minor changes:

```
// status quo
template <Assignable<Foo> T> void foo(T);

// proposed
template <assignable_from<Foo> T> void foo(T);
```

The latter is clearer about the direction, so also an improvement on the name.

2.3 Comprehensive list of current/proposed names

Here is the complete proposed renaming.

Current	Proposed	Notes
Same	same_as	Consistent with derived_from and convertible_to
DerivedFrom	derived_from	
ConvertibleTo	convertible_to	
CommonReference	has_common_reference	“common_reference” might be better, but that’s a struct (however, it’s a struct added in C++20, so it could be possible to rename it if we want)
Common	has_common_type	“common_type” is already in use since C++11
Integral	integral	
SignedIntegral	signed_integral	
UnsignedIntegral	unsigned_integral	
Assignable	assignable_from	
Swappable	swappable	Casey notes: Swappable<T> is ***almost*** equivalent to SwappableWith<T&, T&> -

Current	Proposed	Notes
		the CommonReference requirement introduces some squirrely differences - and it will be equivalent if I can get LWG3175 properly resolved. The differences in usage syntax mirror the differences in the type traits: is_swappable_v<T> is equivalent to is_swappable_with_v<T&, T&>. The two exist to support different uses; Swappable<T>/is_swappable_v<T> is a convenient shorthand for “lvalues of type T can be swapped” vs. SwappableWith<T, U>/is_swappable_with_v<T, U>‘s meaning “expressions E and F such that decltype(E) and decltype(F) are T and U can be swapped.” I don’t think the benefit of having fewer concepts would outweigh the convenience of the shorthand version.
SwappableWith	swappable_with	See swappable
Destructible	destructible	Consistent with is_destructible, but no conflict because traits use is_*
Constructible	constructible	Consistent (but no conflict) with is_constructible and with descriptive uses
DefaultConstructible	default_constructible	Consistent (but no conflict) with is_default_constructible
MoveConstructible	move_constructible	Consistent (but no conflict) with is_move_constructible
CopyConstructible	copy_constructible	Consistent (but no conflict) with is_copy_constructible
Boolean	boolean_type	Suffixed because “boolean” is likely common in user code (“boolean” has no conflict in the standard itself), and we can squint a little to say it fits the rule of using _type for a broad category
EqualityComparable	equality_comparable	
EqualityComparableWith	equality_comparable_with	
StrictTotallyOrdered	totally_ordered	Shouldn’t we drop the “strict” here?
StrictTotallyOrderedWith	totally_ordered_with	Ditto, see also weakly_ordered

Current	Proposed	Notes
Movable	movable	Used as a descriptive word (only 3 places)
Copyable	copyable	Used consistently as a descriptive word
Semiregular	semiregular	Note that the "*semiregular* [italics] exposition-only" is a known poor name that is being actively proposed to be renamed (and isn't a collision even if not renamed)
Regular	regular	No conflict, including with file_type::regular
Invocable	invocable	
RegularInvocable	regular_invocable	
Predicate	predicate	
Relation	relation	
StrictWeakOrder	weakly_ordered	For consistency with StrictTotallyOrdered[With], or should these really be spelled differently? Q (Andrew): Why is there we have a StrictWeakOrder concept but not an EquivalenceRelation concept? Both were in the Palo Alto TR and used for (at least) equal() and mismatch(). It looks like the committee weakened all of the EquivalenceRelation requirements to simple binary predicates. That means you can parameterize equal() in a way that the algorithm doesn't compute equality? A (Casey): Relation was roughly EquivalenceRelation before P1248 removed the semantics; now it is "these four totally unrelated Predicates must be valid" and therefore a meaningless concept. It's on my huge list of things to fix (most likely by incorporating it into StrictWeakOrder and replacing uses of IndirectRelation with a new IndirectPredicate concept).
Readable	readable	More consistent with readable_traits
Writable	writable	More consistent with writable_traits

Current	Proposed	Notes
WeaklyIncrementable	weakly_incrementable	
Incrementable	incrementable	More consistent with incrementable_traits
Iterator	iterator_type	“iterator” could work except that std::iterator is still alive in [depr.iterator.basic]
Sentinel	sentinel_for	Casey notes: I’ve been considering “sentinel_for” / “sized_sentinel_for” which is quite readable in type-constraint usage which is typical for these concepts: “template<frob_iterator I, sentinel_for<I> S>”. Despite that we don’t really ***need*** to change the name, it has caused some confusion that we use “sentinel” as a name for “the thing that denotes the end of a range” and “Sentinel” as the name of the concept that describes the relationship between those things and iterators.
SizedSentinel	sized_sentinel_for	See sentinel_for
InputIterator	input_iterator	More consistent with input_iterator_tag
OutputIterator	output_iterator	More consistent with output_iterator_tag
ForwardIterator	forward_iterator	More consistent with forward_iterator_tag
BidirectionalIterator	bidirectional_iterator	More consistent with bidirectional_iterator_tag
RandomAccessIterator	random_access_iterator	More consistent with random_access_iterator_tag
ContiguousIterator	contiguous_iterator	More consistent with contiguous_iterator_tag
IndirectUnaryInvocable	indirect_unary_invocable	
IndirectRegularUnaryInvocable	indirect_regular_unary_invocable	
IndirectUnaryPredicate	indirect_unary_predicate	
IndirectRelation	indirect_relation	
IndirectStrictWeakOrder	indirect_strict_weak_order	
IndirectlyMovable	indirect_movable	
IndirectlyMovableStorable	indirect_movable_storable	

Current	Proposed	Notes
IndirectlyCopyable	indirect_copyable	
IndirectlyCopyableStorable	indirect_copyable_storable	
IndirectlySwappable	indirect_swappable	
IndirectlyComparable	indirect_comparable	
Permutable	permutable	
Mergeable	mergeable	No conflict, but appears as a function name once in an example in [expr.new], might want to rename that one example even though we don't have to
Sortable	sortable	
Range	range_type	For symmetry with view_type (which can't be just "view")
SizedRange	sized_range	More consistent with disable_sized_range
View	view_type	"view" is not available, it's a namespace alias for std::ranges::view
OutputRange	output_range	No conflict, but is used as a formal parameter name in uninitialized_copy and uninitialized_move, so probably want to rename those parameters (4 occurrences total) if we take this name just to avoid any potential reader confusion
InputRange	input_range	Same as output_range (same 4 occurrences)
BidirectionalRange	bidirectional_range	
RandomAccessRange	random_access_range	
ContiguousRange	contiguous_range	
CommonRange	common_range	
ViewableRange	viewable_range	
UniformRandomBitGenerator	uniform_random_bit_generator	

3 Proposed wording

In the C++ working paper:

- change each “Current” name to its corresponding “Proposed” name in the foregoing table

Additionally, to avoid confusion with the new concept names (these changes are not necessary, just nice):

In [expr.new]/12’s Example:

- change `mergeable` to `can_merge`
- change `unmergeable` to `cannot_merge`

In [uninitialized.copy]/3, [uninitialized_move]/2, and [memory.syn]:

- change `input_range` to `in_range` (4 occurrences)
- change `output_range` to `out_range` (4 occurrences)