Document Number: P1934R0
Date: 2019-10-07
Audience: Library Evolution Working Group
Authors: Casey Carter
Christopher Di Bella
Eric Niebler
Reply to: Casey@Carter.net

`boolean` Considered Harmful

1 Abstract

The `std::boolean` concept strikes the perfect balance of being hard to validate, hard to teach, and hard to use - as is the subject of some US NB comments. This paper proposes replacing the failed concept with a formulation that is simple to validate, teach, and use.

1.1 Revision History

1.1.1 Revision 0

— Initial revision.

2 Problem description

The `std::boolean` concept is defined in [concept.boolean] (quoting the pre-Belfast working draft):

1 The `boolean` concept specifies the requirements on a type that is usable as a truth value.

```
template<class B>
  concept boolean =
    movable<remove_cvref_t<B>> &&
    requires(const remove_reference_t<B>& b1,
             const remove_reference_t<B>& b2, const bool a) {
      { b1 } -> convertible_to<bool>;
      { !b1 } -> convertible_to<bool>;
      { b1 && b2 } -> same_as<bool>;
      { b1 &&  a } -> same_as<bool>;
      {  a && b2 } -> same_as<bool>;
      { b1 || b2 } -> same_as<bool>;
      { b1 ||  a } -> same_as<bool>;
      {  a || b2 } -> same_as<bool>;
      { b1 == b2 } -> convertible_to<bool>;
      { b1 ==  a } -> convertible_to<bool>;
      {  a == b2 } -> convertible_to<bool>;
      { b1 != b2 } -> convertible_to<bool>;
      { b1 !=  a } -> convertible_to<bool>;
      {  a != b2 } -> convertible_to<bool>;
    };
```

2 For some type `B`, let `b1` and `b2` be lvalues of type `const remove_reference_t<B>`. `B` models `boolean` only if

(2.1) — `bool(b1) == !bool(!b1)`.

(2.2) — `(b1 && b2)`, `(b1 && bool(b2))`, and `(bool(b1) && b2)` are all equal to `(bool(b1) && bool(b2))`, and have the same short-circuit evaluation.

(2.3) — `(b1 || b2)`, `(b1 || bool(b2))`, and `(bool(b1) || b2)` are all equal to `(bool(b1) || bool(b2))`, and have the same short-circuit evaluation.

(2.4) — `bool(b1 == b2)`, `bool(b1 == bool(b2))`, and `bool(bool(b1) == b2)` are all equal to `(bool(b1) == bool(b2))`.

(2.5) — `bool(b1 != b2)`, `bool(b1 != bool(b2))`, and `bool(bool(b1) != b2)` are all equal to `(bool(b1) != bool(b2))`.

The Standard Library requires the results of evaluating predicate expresssions to model `boolean` which indicates when, for example, comparison expressions are true or false. The concept has accreted requirements over time in an attempt to provide a more complete approximation of the grammar and semantics of boolean expressions. It has in the process become somewhat complex, with fourteen direct expression requirements and quite a few semantic constraints to relate those expressions to each other. This level of definition complexity would be acceptable if the concept is easy to apply and reason about. Unfortunately, that is not the case.

2.1 Why the complexity?

There are two different kinds of binary expressions required by `boolean`: those that operate on two operands of the same type (an lvalue `const B`), and those that mix a `B` operand with a `bool` operand. Ideally, the concept would constrain how those expressions work when applied to any pair of operands whose types each model the concept: this would achieve the fully-general boolean grammar we want to express. There is no way to express such a universal quantification using C++ concepts, so we fall back to applying requirements to only the type in question and to `bool` itself.

It has been suggested that the universal quantification be applied outside of the concept syntax as a set of purely semantic requirements. This is feasible, but perhaps not necessarily desirable. Do we really want to have a concept that can only truly be verified by examining the entire program text? Shouldn't I be able to reason locally, say about whether `bool` satisfies the `boolean` concept? In the presence of universally quantified semantic constraints a type satisfies the concept only if there is - for example - no overloaded operator anywhere in the program that violates the semantic requirements.

2.2 Non-uniformity of application

So the formulation of `boolean` is necessarily a compromise; we must step outside the required expressions when forming boolean expressions with operands of multiple types - other than `bool` - that model the concept by converting those operands to `bool`. Consequently, most simple uses need not force any conversions:

```
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
O copy(I first, S last, O out) {
  for(; first != last; ++first, (void) ++out) {
    *out = *first;
  }
  return out;
}
```

which makes it easy to forget to force conversions when we venture nearer the edges of the design:

```
template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2>
  requires indirectly_comparable<I1, I2, ranges::equal>
bool equal(I1 first1, S1 last1, I2 first2, S2 last2) {
  for(; first1 != last1 && first2 != last2; // Oops; both need casts to bool
      ++first1, (void) ++first2) {
    if (*first1 != *first2)
      return false;
  }
  return first1 == last1 && first2 == last2; // Oops; here as well
}
```

Style guides will presumably "fix" this problem by mandating that all expressions whose types model `boolean` are converted to `bool`, implicitly or explicitly. That simple and universal rule makes it far easier to teach and use the concept without error, which begs the question of what we get from `boolean`'s current complex definition.

Certainly not ease of application and modeling. Quickly look back at the syntactic and semantic requirements; are they satisfied by `void*`? How about `int`? Is it readily apparent that integral types `T` model `boolean` only over the restricted domain `{T(0), T(1)}`? The authors' experience is that this formulation is far from transparent to new users.

2.2.1 Costs

Checking fourteen *compound-requirement*s has a non-negligible compile time cost. Overload resolution for each required expression and satisfaction checking of the *type-requirement* are not free. Given that other concepts require often several predicate expressions - e.g., `sentinel_for`, `equality_comparable`, and the mother of all comparison concepts `totally_ordered_with` - the checking costs are often greatly magnified.

It's unfortunate that checking `boolean` is so expensive, especially in the presence of a "always force a conversion" style guideline which suggests that very few of the concept's required expressions will be used in practice. The current formulation is hard to validate, hard to use, and hard to teach.

3 Proposal

Several alternative formulations of `boolean` have been suggested to the authors.

— Just use `bool`, i.e., require predicate expressions to be prvalues of type `bool`. This simple formulation notably doesn't need a named concept at all: we can replace uses of `boolean<T>` (and the *type-requirement* `-> boolean;`) with `same_as<T, bool>` (respectively `-> same_as<bool>;`).

— Use `decays_to<T, bool>`, where `decays_to` is:

```
template <class T, class U>
concept decays_to = same_as<decay_t<T>, U>;
```

A slight generalization of the above, this additionally allows predicate expressions that yield an lvalue (possibly-`const`) `bool`.

— Use `one_of<T, bool, true_type, false_type>` where `one_of` is:

```
template <class T, class... Us>
concept one_of = (same_as<decay_t<T>, Us> && ...);
```

This formulation cleverly covers the types for which `boolean` support is most often requested.

These formulations notably never[1] require a user to force a conversion to `bool` when forming binary expressions, which is a nicely consistent rule for usage.

The other consistent rule is "always force a conversion," and there are at least two suggestions that use that rule:

— "Contextually convertible to `bool`." There's no convenient short spelling for contextual conversion to `bool`, so we'd need to keep a named concept:

```
template <class T>
concept boolean = requires(const remove_reference_t<T>& t) {
  t ? 0 : 0; // The conditional operator contextually converts its first operand to bool.
};
```

This is the rule the [algorithms] have used for predicates for decades.

— "explicitly and implicitly convertible to `bool`" which is similar to the above - contextual conversion to `bool` can trigger explicit conversions - with the additional requirement for implicit conversion. This has the short spellings `convertible_to<T, bool>` and `-> convertible_to<bool>;`. While this formulation is slightly more restrictive than "contextually convertible," the direct reuse of the `convertible_to` concept is attractive.

The authors propose this last formulation be used to replace the `boolean` concept for C++20. `convertible_to<bool>` seems to strike a balance between the concerns favored by the other formulations. There's a short spelling, allowing the named concept to be removed. It admits a variety of models including all the examples currently listed as satisfying `boolean`. It's relatively cheap to check (as are all of the noted alternatives). It's quite close to the "old" notion of what a predicate should yield. It uses the other library concepts well:

1) Ignoring for the moment that a user could e.g. define an operator overload that takes `bool` and `true_type` in the global namespace.

`convertible_to<T>` *should* be what we teach programmers to reach for when they want to capture the set of types that convert to `T`.

4 Technical Specifications

Change [cmp.concept] as follows:

```
template<class T, class U>
concept partially-ordered-with = // exposition only
  requires(const remove_reference_t<T>& t, const remove_reference_t<U>& u) {
    { t <  u } -> ~~boolean~~convertible_to<bool>;
    { t >  u } -> ~~boolean~~convertible_to<bool>;
    { t <= u } -> ~~boolean~~convertible_to<bool>;
    { t >= u } -> ~~boolean~~convertible_to<bool>;
    { u <  t } -> ~~boolean~~convertible_to<bool>;
    { u >  t } -> ~~boolean~~convertible_to<bool>;
    { u <= t } -> ~~boolean~~convertible_to<bool>;
    { u >= t } -> ~~boolean~~convertible_to<bool>;
  };
```

Change [concepts.syn] as follows:

```
namespace std {
  [...]
  // 18.5, comparison concepts
  ~~// 18.5.2, concept boolean~~
  ~~template<class B>~~
    ~~concept boolean = see below;~~
  [...]
}
```

Strike the subclause [concept.boolean].

Change [concept.equalitycomparable] as follows:

```
template<class T, class U>
  concept weakly-equality-comparable-with = // exposition only
    requires(const remove_reference_t<T>& t,
             const remove_reference_t<U>& u) {
      { t == u } -> ~~boolean~~convertible_to<bool>;
      { t != u } -> ~~boolean~~convertible_to<bool>;
      { u == t } -> ~~boolean~~convertible_to<bool>;
      { u != t } -> ~~boolean~~convertible_to<bool>;
    };
```

[...]

Change [concept.totallyordered] as follows:

```
template<class T>
  concept totally_ordered =
    equality_comparable<T> &&
    requires(const remove_reference_t<T>& a,
             const remove_reference_t<T>& b) {
      { a <  b } -> ~~boolean~~convertible_to<bool>;
      { a >  b } -> ~~boolean~~convertible_to<bool>;
      { a <= b } -> ~~boolean~~convertible_to<bool>;
      { a >= b } -> ~~boolean~~convertible_to<bool>;
    };
```

[...]

```
template<class T, class U>
  concept totally_ordered_with =
    totally_ordered<T> && totally_ordered<U> &&
    common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
    totally_ordered<
      common_reference_t<
        const remove_reference_t<T>&,
        const remove_reference_t<U>&>> &&
    equality_comparable_with<T, U> &&
    requires(const remove_reference_t<T>& t,
             const remove_reference_t<U>& u) {
      { t <  u } -> ~~boolean~~convertible_to<bool>;
      { t >  u } -> ~~boolean~~convertible_to<bool>;
      { t <= u } -> ~~boolean~~convertible_to<bool>;
      { t >= u } -> ~~boolean~~convertible_to<bool>;
      { u <  t } -> ~~boolean~~convertible_to<bool>;
      { u >  t } -> ~~boolean~~convertible_to<bool>;
      { u <= t } -> ~~boolean~~convertible_to<bool>;
      { u >= t } -> ~~boolean~~convertible_to<bool>;
    };
```

[...]

Change [concept.predicate] as follows:

```
template<class F, class... Args>
  concept predicate =
    regular_invocable<F, Args...> &&
    ~~boolean~~convertible_to<invoke_result_t<F, Args...>, bool>;
```