Add Concept Number to the Standard Library

Christopher Di Bella (cjdb.ns@gmail.com)

2017-03-26

Abstract

This document proposes a Standard `Number` concept family, which attempts to constrain types that claim to be numbers. It is also responsible for ensuring that `Number`s of different types are compatible.

Motivation

When introducing programmers to concepts, one of the first formal concepts that appears to be presented is a `Number` concept[1][3][6][5][1][4]. Unfortunately, there doesn't seem to be an agreed upon `Number` concept to introduce: differing audiences may require different `Number` specifications. For example, [4] provides a brief overview of an incomplete `Number` to make a point, whereas [1] relies on `Number`[2] for instruction regarding how a concept should be built, and is thus more comprehensive. A programmer that has taken the concept in [4] at face value would be shocked if they used the concept in [1]; and both programmers would be surprised if they used a version of `Number` that was written by someone who has read *both* articles, and combined the two to make a super `Number`. Even that concept can still be considered incomplete, since this paper provides a more mature `Number` concept *family*.

As people experiment, they may be inclined to add to their `Number` concepts, and some `Number`s might be stricter than others. This will cause problems later on, when programmers change companies, or try new libraries. As this is such a simply-named concept, it ought to be standardised to prevent this from happening: then, instructors are free to introduce `Number` however lightly or detailed as they need, and can point out that `Number` is actually a standardised concept that is used for education.

A `Number` concept, or a family of `Number` concepts will be useful, since many computations rely on different types of numbers; at present, these generic algorithms either risk being exposed to incompatible types (e.g. `vector`), or have a non-standard-but-commonly-required concept to stand at the function declaration. A `Number` concept is beyond the scope of the Ranges TS[2], and so there is no Standard `Number` (family) at present. We should strike while the iron is hot, and standardise a strict `Number` concept family before lots of programmers have the opportunity to craft their own incomplete versions.

Proposal

This proposal suggests to add:

7.2 Header `<experimental/ranges/concepts>` synopsis [concepts.lib.synopsis]

```
// 7.7.1, Number:
template <class N>
concept bool Number() {
  return see below;
}

template <class N1, class N2>
```

1) The author initially used `Artihmetic`, and suspects that `Numerical` would be a common alternative for the same concept.
2) Called `Arithmetic` in the actual article.

```
concept bool Number() {
  return see below;
}

template <class N1, class N2, class N3, class... Ns>
concept bool Number() {
  return see below;
}

// 7.7.2, RegularNumber:
template <class N>
concept bool RegularNumber() {
  return see below;
}

template <class N, class N2>
concept bool RegularNumber() {
  return see below;
}

template <class N1, class N2, class N3, class... Ns>
concept bool RegularNumber() {
  return see below;
}

// 7.7.3, OrderedNumber:
template <class N>
concept bool OrderedNumber() {
  return see below;
}

template <class N, class N2>
concept bool OrderedNumber() {
  return see below;
}

template <class N1, class N2, class N3, class... Ns>
concept bool OrderedNumber() {
  return see below;
}

// 7.7.4, RegularOrderedNumber:
template <class N>
concept bool RegularOrderedNumber() {
  return see below;
}

template <class N, class N2>
concept bool RegularOrderedNumber() {
  return see below;
}

template <class N1, class N2, class N3, class... Ns>
concept bool RegularOrderedNumber() {
  return see below;
```

```
}

// 7.7.5, BitwiseNumber:
template <class N, class I>
concept bool BitwiseNumber() {
  return see below;
}
```

7.7 Numerical concepts [concepts.lib.number.general]

7.7.1 Concept Number [concepts.lib.number]

```
template <class N>
concept bool Number() {
  return !Same<bool, N>() &&
    !Same<char, N>() &&
    !Same<wchar_t, N>() &&
    !Same<char16_t, N>() &&
    !Same<char32_t, N>() &&
    Regular<N>() &&
    requires(N n, const N cn) {
      {N{0}};
      {+cn} -> N;
      {-cn} -> N;
      {cn + cn} -> N; // not required to be associative or commutative
      {cn - cn} -> N;
      {cn * cn} -> N; // not required to be associative, commutative, or distributive
      {cn / cn} -> N;
      {n += cn} -> N&; // not required to be associative or commutative
      {n -= cn} -> N&;
      {n *= cn} -> N&; // not required to be associative, commutative, or distributive
      {n /= cn} -> N&;
    };
}
```

1 This is the base numerical concept, and shall represent any type that can be treated as a number.

2 The arithmetic types `char`, `wchar_t`, `char16_t`, and `char32_t` are character types, and should be excluded from `Number`, as they aren't typically used for arithmetic operations. Similarly for type `bool`.

[Editor's note: For the remainder of this proposal, the term "integral" shall exclude `bool`, `char`, `wchar_t`, `char16_t`, and `char32_t`, and `Integral` shall refer to any conforming Standard integral type.]

3 A `Number` must support all arithmetic operators common to both integral operations and floating-point operations, and all equivalent arithmetic compound operators.

4 A `Number` must be constructible from an `int`-literal[4].

5 To support parallelism, types conforming to the `Number` concept are not required to have associative or commutative addition and multiplication. Multiplication is also not required to be distributive.

```
template <class N1, class N2>
concept bool Number() {
  return Number<N1>() &&
    Number<N2>() &&
    Assignable<N1&, const N2&>() &&
    requires(N1 n1, const N1 cn, const N2 n2) {
      {cn + n2} -> common_type_t<N1, N2>; // not required to be associative or commutative
```

```
      {cn - n2} -> common_type_t<N1, N2>;
      {cn * n2} -> common_type_t<N1, N2>; // not required to be associative, commutative, or distributive
      {cn / n2} -> common_type_t<N1, N2>;
      {n2 + cn} -> common_type_t<N1, N2>; // not required to be associative or commutative
      {n2 - cn} -> common_type_t<N1, N2>;
      {n2 * cn} -> common_type_t<N1, N2>; // not required to be associative, commutative, or distributive
      {n2 / cn} -> common_type_t<N1, N2>;
      {n1 += n2} -> N1&;
      {n1 -= n2} -> N1&;
      {n1 *= n2} -> N1&;
      {n1 /= n2} -> N1&;
    };
}

template <class N1, class N2, class N3, class... Ns>
concept bool Number() {
  return Number<N1, N2>() &&
    Number<N1, N3>() &&
    Number<N1, Ns...>();
}
```

6 The first `Number` concept assumes that numerical operations only happen on types that are the same. In reality, this can be very different. [*Example:* Consider this program:

```
class Big_int {
public:
   Big_int() = default;
   explicit Big_int(int);

  Big_int& operator+=(const Big_int&);
  Big_int& operator-=(const Big_int&);
  Big_int& operator*=(const Big_int&);
  Big_int& operator/=(const Big_int&);

  // remaining implementation unspecified...
};

Big_int operator+(const Big_int&);
Big_int operator-(const Big_int&);
Big_int operator+(const Big_int&, const Big_int&);
Big_int operator-(const Big_int&, const Big_int&);
Big_int operator*(const Big_int&, const Big_int&);
Big_int operator/(const Big_int&, const Big_int&);
bool operator==(const Big_int&, const Big_int&);
bool operator!=(const Big_int&, const Big_int&);

static_assert(Number<Big_int>());

template <Number T, Number U>
Number some_operation(T t, U u) noexcept
{
   return t + u;
}

int main()
{
```

```
    some_operation(Big_int{}, 0);
  }
```

This example is not a well-formed program, however, the underlying diagnostic does manifests itself inside `some_operation`, rather than at the interface level.

This is because while `T` and `U` both conform to the `Number` concept, the concept makes no guarantees that `T` and `U` are compatible for operating on one another. *— end example*]

7 The latter two offer the solution to this dilemma, by requiring that arithmetic operations are compatible.

8 The second type does not participate in the assignment operations.

7.7.2 Concept RegularNumber [concepts.lib.regular.number]

```
template <class N>
concept bool RegularNumber() {
  return Number<N>();
}

template <class N1, class N2>
concept bool RegularNumber() {
  return Number<N1, N2>();
}

template <class N1, class N2, class N3, class... Ns>
concept bool RegularNumber() {
  return Number<N1, N2, N3, Ns...>();
}
```

1 A `RegularNumber` guarantees that addition and multiplication are both associative and commutative. Multiplication is also guaranteed to be distributive.

2 [*Note:* Floating-point numbers do not satisfy `RegularNumber`. *— end note*]

3 [*Note:* The distinction between `Number` and `RegularNumber` is purely semantic. *— end note*]

7.7.3 Concept OrderedNumber [concepts.lib.ordered.number]

```
template <class N>
concept bool OrderedNumber() {
  return Number<N>() &&
    StrictTotallyOrdered<N>() &&
    requires(N n) {
      {++n} -> N&;
      {--n} -> N&;
      {n++} -> N;
      {n--} -> N;
    };
}

template <class N1, class N2>
concept bool OrderedNumber() {
  return OrderedNumber<N1>() &&
    OrderedNumber<N2>() &&
    Number<N1, N2>() &&
    StrictTotallyOrdered<N1, N2>();
}
```

```
template <class N1, class N2, class N3, class... Ns>
concept bool OrderedNumber() {
  return OrderedNumber<N1, N2>() &&
    OrderedNumber<N1, N3>() &&
    OrderedNumber<N1, Ns...>();
}
```

1 An `OrderedNumber` is a `Number` that is also `StrictTotallyOrdered` and supports both pre-increment and post-increment operators.

2 [*Note:* All integral and floating-point types are `OrderedNumber`s. — *end note*]

3 [*Note:* `complex` does not meet the requirements for `OrderedNumber`. — *end note*]

7.7.4 Concept RegularOrderedNumber [concepts.lib.regular.ordered.number]

```
template <class N>
concept bool RegularOrderedNumber() {
  return RegularNumber<N>() &&
    OrderedNumber<N>();
}

template <class N1, class N2>
concept bool RegularOrderedNumber() {
  return RegularNumber<N1, N2>() &&
    OrderedNumber<N1, N2>();
}

template <class N1, class N2, class N3, class... Ns>
concept bool RegularOrderedNumber() {
  return RegularNumber<N1, N2, N3, Ns...>() &&
    OrderedNumber<N1, N2, N3, Ns...>();
}
```

1 A `RegularOrderedNumber` is both a `RegularNumber` and an `OrderedNumber`.

2 [*Note:* Floating-point numbers do not satisfy `RegularOrderdNumber`. — *end note*]

3 [*Note:* The distinction between `OrderedNumber` and `RegularOrderedNumber` is purely semantic. — *end note*]

[Editor's note: This should be the default `Number` concept. It was initially called `Number`, but no appropriate names could be matched with all of the preceding concepts, and so the naming scheme was rearranged to what is present.]

7.7.5 Concept BitwiseNumber [concepts.lib.bitwise.number]

```
template <class N, class I>
concept bool BitwiseNumber() {
  return UnsignedIntegral<I>() &&
    RegularOrderedNumber<N>() &&
    sizeof(I) <= sizeof(N) &&
    requires(N n, const N cn, const I i) {
      {cn & i} -> N;
      {cn | i} -> N;
      {cn ^ i} -> N;
      {~cn} -> N;
      {cn >> i} -> N;
```

```
    {cn << i} -> N;
    {n &= i} -> N&;
    {n |= i} -> N&;
    {n ^= i} -> N&;
    {n >>= i} -> N&;
    {n <<= i} -> N&;
  };
}
```

1 A `BitwiseNumber` is a `Number` that supports bitwise operations.

2 Whether or not a `BitwiseNumber` shall support `{n >> n} -> N`, etc. is undecided: this proposal states no preference, and would like to open the floor for discussion.

3 Whether or not the requirement for `I` is `Integral` or `UnsignedIntegral` is also left to be decided, although the proposal is strongly in favour of `UnsignedIntegral`. The outcome for *Proposal for C++ replacing unsigned integer types with signed integer types in the Standard Library* should, in the author's opinion, affect this decision.

4 [*Note:* Floating-point numbers do not satisfy `BitwiseNumber`. — *end note*]

Implementation

A working implementation can be found by cloning `git@github.com:cjdb/cmcstl2.git` and checking out branch `numerics`.

Acknowledgements

This document was initially inspired by (in order): [7], [3], [5], [1], [2]. Further inspiration was derived from [4] during the writing of this proposal. The author would like to thank Sergey Zubkov and Bjarne Stroustrup for their inspiration.

The author would like to thank Oswyn Brent, Casey Carter, Manuel Chakravarty, Daryl D'Sousa, Margret Steele, Andrew Sutton, and Sergey Zubkov for their review of [1], which was featured at CppCon 2016 in the open content session, "Start teaching the Concepts TS and ranges library now!".

Arien Judge proof read this document prior to its submission.

Special thanks to Eric Niebler and Casey Carter for their guidance (and patience) along the way, as well as their instruction on the workings of WG21 and proposals at large.

The LaTeXsource for this document was adapted from a revision of the Ragnes TS.

Finally, this proposal would not have been possible without you, the reader, and the biggest thanks of all is dedicated to you.

Bibliography

[1] Christopher Di Bella. Concepts ts and ranges ts extension notes. https://github.com/cjdb/cppcon/blob/master/concepts/concepts.asciidoc, 09 2016. Accessed: 2017-03-25. Last updated: 2016-09-28.

[2] Eric Niebler and Casey Carter. N4651 working draft, c++ extensions for ranges. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4651.pdf, 03 2017.

[3] Bjarne Stroustrup. Cppcon 2014: Bjarne stroustrup "make simple tasks simple!". https://www.youtube.com/watch?v=nesCaocNjtQ, 09 2014.

[4] Bjarne Stroustrup. Concepts: The future of generic programming or how to design good concepts and use them well. http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0557r0.pdf, 01 2017.

[5] Bjarne Stroustrup and Herb Sutter. Cppcoreguidelines. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, 2015. Accessed: 2017-03-23.

[6] Andrew Sutton. Introducing concepts. https://accu.org/index.php/journals/2157, 10 2015.

[7] Sergey Zubkov. Quora.com – sergey zubkov's answer to what is your opinion on herb sutter's advice for automatic type deduction? https://www.quora.com/What-is-your-opinion-on-Herb-Sutters-advice-for-automatic-type-deduction.