Constrained Numbers

Document #:	P2993R0
Date:	2024-3-21
Project:	Programming Language C++
Audience:	Numerics, Safety

Reply-to:	Luke Valenty <lvalenty@gmail.com>

Contents

1 Introduction

This paper proposes the template type `constrained_number<C, T>`. It allows variables and functions to advertise and enforce numerical constraints at compile-time. The constraints are guaranteed to be held true at run-time. Operations on instances of `constrained_number` are applied to the value of the underlying numerical type and produce a new type with appropriate constraints for the result. Additional library support is proposed to complete an ergonomic interface.

2 Motivation

In the five year span from 2018 to 2022 there are 1,515 CVE records of security vulnerabilities related to integer overflow. [CVE-2023] On June 4, 1996 the V88 flight of the Ariane 5 rocket ended in catastrophic failure with the loss of all satellites on board at a total cost of US $370 million. The failure was due to the error handling of a caught integer overflow. [LIONS-1996] In 2015 and 2020, two separate bugs related to integer overflow of time values in the Boeing 787 aircraft caused the flight systems to either crash or report misleading information on cockpit instruments. [AVAREZ-2015] [CORFIELD-2015]

Integer overflows and the associated consequences are a significant source of functional, security, and safety bugs. Even when an integer overflow is detected at run-time, incorrect error handling can still result in failure.

There are multiple programming rules, guidelines, and standards that, in part, attempt to tackle the problem of integer overflow, underflow, divide-by-zero, and out of bounds array access:

— C++ Core Guidelines
— MISRA C and MISRA C++ Guidelines
— CERT C and CERT C++ Secure Coding Standards
— ISO/IEC TS 17961:2013 (C Secure Coding Rules)
— IEC 61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems)

Despite these industry accepted and applied standards, integer overflow and related issues continue to be an expensive and dangerous problem in common software libraries, user applications, firmware, and safety critical systems.

Existing standards, guidelines, and libraries for correctly operating on numbers are not sufficient. A system built into the C++ standard library that allows correct by construction numeric operations, is ergonomic to use, minimizes run-time impact, and can be adopted piece-meal will eliminate several classes of coding errors where it is used.

3 Scope

This paper proposes a new system of constrained numbers for the C++ standard library. This system employs compile-time checking throughout its design and falls back to early run-time checking where necessary.

The current scope is limited to integrals.

The scope includes:

1. A DSL for precise specification of constraints.
2. A constrained number value type and associated concepts with compile-time constraints on the run-time value.
3. Both compile-time and run-time checked mechanisms for creating constrained numbers without violating constraints.
4. Arithmetic operators and functions for constrained numbers that correctly propagate new constraints to the result type based on the input constraints to the function.
5. A mechanism to extract raw values from constrained numbers using `static_cast`.
6. A method to cast a raw value into a `constrained_number`, bypassing all compile-time and run-time checks.
7. Additions to a limited set of algorithms to enable use of `constrained_number`s for those algorithms.
8. Additions to a limited set of containers to enable use of `constrained_number`s for random access.

4 History and Related Work

Constrained Numbers is not the only proposal or library that attempts to solve the problem of unsafe integer or numeric operations in programs. There are a number of other libraries with different strategies and tradeoffs that are worthwhile to look at.

— Boost's safe_numerics. A library by Robert Ramey included in Boost. It works on the {cpp}14 standard. It offers customization points on how exceptional cases are handled: compile-time, runtime exceptions, or a custom error handler. It also allows customization for how integer promotion is handled. It only supports a single interval to specify value requirements, while this proposal specifies a DSL to allow more tightly-constrained requirements.
— bounded::integer. Defines a `bounded::integer<Min, Max>` template type that tracks the set of potential values of operations using interval math. Implemented in {cpp}20.
— SafeInt. Provides both a {cpp}11 and C library implementation. Appears to only support runtime checking.
— PSsimplesafeint. "A {cpp}20 implementation of safe (wrap around) integers following MISRA {cpp} rules."
— Clang's -fbounds-safety "-fbounds-safety is a C extension to enforce bounds safety to prevent out-of-bounds (OOB) memory accesses, which remain a major source of security vulnerabilities in C. -fbounds-safety aims to eliminate this class of bugs by turning OOB accesses into deterministic traps."
— Safe Arithmetic The Constrained Numbers proposal is based on concepts introduced in this library. The library itself will be updated by the author to reflect this proposal.

Related proposals?

Related ISO standards?

5 Comparison Tables

Table 1: C++ Core Guidelines, ES.103 [COREGUIDE-2023]

Before	After

`std::array<int, 10> a{};`
`a.at(10) = 7; // runtime exception`
`a[10] = 7; // out of bounds memory access!`
 	
`for (int n = 0; n < 11; ++n) {`
`  a.at(n) = 9; // runtime exception`
`  a[n] = 9; // out of bounds memory access!`
`}`
	`std::array<int, 10> a{};`
	`a.at(10_cn) = 7; // fails compilation!`
	`a[10_cn] = 7; // fails compilation!`
	
	`std::for_each(0_cn, 11_cn, [](auto n){`
	`  a.at(n) = 9; // fails compilation!`
	`  a[n] = 9; // fails compilation!`
	`});`

Table 2: C++ Core Guidelines, ES.103 [COREGUIDE-2023]

Before	After

`int n =`
`  numeric_limits<int>::max();`
 	
`int m =`
`  n + 1; // bad, numeric overflow`
	`constrained_integral<int> n =`
	`  numeric_limits<int>::max();`
	
	`constrained_number<int> m =`
	`  n + 1_cn; // Fails compilation!`

Table 3: C++ Core Guidelines, ES.103 [COREGUIDE-2023]

Before	After

`int area(int h, int w) {`
`  return h * w;`
`}`
 	
`// bad, numeric overflow`
`auto a = area(10'000'000, 100'000'000);`
	`any_constrained auto area(`
	`  any_constrained auto h,`
	`  any_constrained auto w`
	`) {`
	`  return h * w;`
	`}`
	
	`// Fails compilation!`
	`auto a = area(10'000'000_cn, 100'000'000_cn);`

6 Design Overview

The constrained number system takes the following philosophies of the C++ Core Guidelines to heart and assists users of this design in doing the same: [COREGUIDE-2023]

- Philosophies
  - P.4: Ideally, a program should be statically type safe
  - P.5: Prefer compile-time checking to run-time checking
  - P.6: What cannot be checked at compile time should be checkable at run time
  - P.7: Catch run-time errors early
- Interfaces
  - I.4: Make interfaces precisely and strongly typed
  - I.5: State preconditions (if any)

    - I.7: State postconditions
- Expressions
    - ES.100: Don't mix signed and unsigned arithmetic
    - ES.101: Use unsigned types for bit manipulation
    - ES.102: Use signed types for arithmetic
    - ES.103: Don't overflow
    - ES.104: Don't underflow
    - ES.105: Don't divide by integer zero
    - ES.106: Don't try to avoid negative values by using `unsigned`

The design of the system's API attempts to create a "pit of success" where using it correctly is the easiest and simplest thing to do. [MARIANI-2018] This is achieved by providing clear entry points for safely ferrying raw numbers into constrained numbers, functions that operate on constrained numbers, and clear exit points for extracting raw numbers back out.

Operations performed on one or more `constrained_number`s calculate new constraints for the return type. The returned value is guaranteed to satisfy the new constraints.

This design strives to move as much checking as possible to compile-time. Not only does this catch potential issues sooner when they are cheaper to fix, it also reduces the number of locations in a program in which run-time error checking and handling needs to be considered. This improves performance, but more importantly it makes the user's design simpler to define and easier to implement.

6.1 Creating Constrained Numbers

There are a limited number of ways in which a constrained number can be constructed. All but one of these methods produces a constrained number whose run-time value is guaranteed to satisfy the compile-time constraints.

6.1.1 Bare Assignment/Initialization

A bare number may be assigned to a constrained number if the representable range of the number's type satisfies the constraint. Checked at compile-time, no additional run-time overhead.

```
// Good! Any 32- or 64-bit signed integer provably satisfies the constraints of
// 'constrained_integral<int64_t>' at compile time.
constrained_integral<int64_t> foo = 42;
```

If the bare number's type does not satisfy the constraints then a `static_assert` will be triggered.

```
// Compilation error! A signed number violates the constraint.
constrained_number<constrain_interval<0, 10>> bar = 4;
```

6.1.2 Constant Assignment/Initialization

An integral constant may be assigned to a constrained number if the value of the constant satisfies the constraint. Checked at compile-time, no additional run-time overhead.

```
// Good! The constant value of '4' satisfies the constraint.
constrained_number<constrain_interval<0, 10>> bar = 4_cn;

// Compilation error! '42' does not satisfy the constraint.
bar = 42_cn;
```

6.1.3 Assignment/Initialization From Another Constrained Number

A constrained number may be assigned a value from another constrained number as long as the left-hand-side constraint represents a set of numbers that is a superset of the numbers represented by the right-hand-side

constraint. Checked at compile-time, no additional run-time overhead.

```
constrained_number<constrain_interval<0, 100>>  foo = get_some_foo();

// Compilation error! 'foo' may contain a run-time value that does not satisfy
// 'bar's compile-time constraints.
constrained_number<constrain_interval<0, 10>>  bar = foo;


bar = get_some_bar();

// Good! 'bar' is guaranteed to contain a run-time value that satisfies 'foo's
// constraint.
foo = bar;
```

6.1.4 Run-time Checked Assignment/Initialization

Validating input numbers satisfy constraints requires run-time checking. `make_constrained` will first attempt to use compile-time checks to ensure a given value satisfies the constraint. If the given value cannot be checked at compile-time, then it will perform a run-time check. If the value fails the run-time check it will throw a `constraint_violation` exception.

```
constexpr any_constraint auto multiple_of_five_c =
    constraint_of<int64_t> &&
    constrain_multiple<5>;

using mult_of_five_t =
    constrained_number<multiple_of_five_c>;

// Checked at compile-time, no runtime check
auto v1 = make_constrained<multiple_of_five_c>(1005_cn);

// Checked at runtime, exception thrown if constraint not satisfied
mult_of_five_t v2 = make_constrained<multiple_of_five_c>(get_some_raw_int());

// Fails at compile-time
auto v3 = make_constrained<multiple_of_five_c>(12_cn);

// Throws exception at runtime.
mult_of_five_t v4 = make_constrained<multiple_of_five_c>(12);
```

A proposed `match_constraint(...)` function may be used to perform the check as well as control flow.

```
constexpr auto foo_func = match_constraint(
  [](constrained_number<constrain_interval<0, 10>> foo){
    std::print("Value is a number in [0, 10], {}", foo);
  },
  [](){
    std::print("Value is not a number in [0, 10]");
  }
);

// Good! The raw integer will be checked at run-time before being assigned to a
// constrained number.
foo_func(get_some_raw_int());
```

6.1.5 Clamped Assignment/Initialization

Often a simple clamp operation is enough to validate input. This of course requires additional runtime overhead to perform the clamp operation.

```
// Good! The clamped value is guaranteed to fit in `foo`.
constrained_number<constrain_interval<0, 10>> foo = clamp(get_some_raw_int(), 0_cn, 10_cn);
```

6.1.6 Forced Assignment/Initialization

There are situations in which the programmer can prove a value satisfies the necessary constraint, but the constrained number system is not able to do so. When they also require absolute performance a `constraint_cast<T>()` may be used to force a value into a constrained number. This provides no run-time or compile-time checks. The unique cast name is used so it can be caught with linting tools and code reviews.

```
// DANGEROUS! Should almost never be used. 'foo' may contain a run-time value
// that does not satisfy its constraint.
auto foo = constraint_cast<constrain_interval<0, 10>>(get_some_raw_int());
```

6.2 Retrieving Raw Numbers

Constrained numbers do not provide an implicit conversions to their underlying number type. A `static_cast` must be used to explicitly retrieve the value of the underlying number type. This performs no run-time checks. `T` must be the numeric type of the constrained number.

```
constrained_number<constrain_interval<0, 10>> foo = get_some_foo();

// Good!
auto raw_foo = static_cast<uint32_t>(foo);
```

6.3 Constraint DSL

The constraint DSL is used to define the set of valid values for a constrained number templated type. safe_numerics and bounded::integer both use interval arithmetic at compile time to track the set of valid values. This proposal works with intervals, sets, tristate bitmasks, and set operators like union, intersection, and difference to define arbitrary requirements on values. Just like safe_numerics and bounded::integer, it will calculate the new set of possible values for any arithmetic, bitwise, or shift operation.

Since interval requirements are commonly used, there are convenience types for creating them:

```
constrained_number<constrain_interval<-100, 100>> small_number{};
```

Which is equivalent to the following:

```
constrained_number<constrain_interval<0, 10>> small_number = 0_cn;
```

If we want to exclude '0' from the range, the DSL allows us to do that:

```
constrained_number<constrain_interval<-100, -1> || constrain_interval<1, 100>> small_nonzero_number = 1_c
```

This enables the library to protect against divide-by-zero at compile-time. The division operator function arguments require the divisor to be non-zero.

```
// COMPILE ERROR: small_number _might_ be zero
auto result_1 = 10_cn / small_number;

// SAFE: small_nonzero_number is guaranteed to be non-zero.
auto result_2 = 10_cn / small_nonzero_number;
```

The DSL can be used by itself, outside of `constrained_number`. This can be helpful to illustrate the rules and capabilities of the DSL itself.

The assignment operator and constructors for `constrained_number<C, T>` that accept another `constrained_number<RhsC, Rh` use set inequality operators to determine whether it is safe or not. The right-hand-side argument's requirements must be a subset of the left-hand-side target.

```
constexpr auto non_zero_req = constrain_interval<-100, -1> || constrain_interval<1, 100>;
constexpr auto small_num_req = constrain_interval<-100, 100>;

// The constraint `<=` operator is used for 'is subset of'
static_assert(non_zero_req <= small_num_req);

constrained_number<non_zero_req> non_zero = 1_cn;

// The constraint `<=` operator ensures this assignment is safe at compile-time
constrained_number<small_num_req> small_num = non_zero;
```

When any operation is performed on a constrained number instance, the mirror operation is performed on the requirements.

```
constexpr auto one_to_ten_req = constrain_interval<1, 10>;
constexpr auto non_zero_req = constrain_interval<-100, -1> || constrain_interval<1, 100>;

constrained_number<non_zero_req> a = 42_cn;
constrained_number<one_to_ten_req> b = 3_cn;

auto c = a * b;

// runtime value is updated as expected
assert(c == 126);

// static constraints are also updated as expected
static_assert(c.constraint == constrain_interval<-1000, -1> || constrain_interval<1, 1000>);
```

6.3.1 Operands

Table 4: Constraint DSL Operands

Name	Definition	C++	Description
Interval	$[a, b]$	`constrain_interval<a, b>`	A set of values from a to b, inclusive.
Set	$\{a, b, ...\}$	`constrain_set<a, b, ...>`	A set of explicitly defined values.
Mask	$\{x \in \mathbb{Z} \mid 0 \leq x < 2^n \wedge (x \,\&\, \sim V) = C\}$	`constrain_mask<V, C>`	V is the variable bits mask. C is the constant bits mask. Mask produces a set of integers where the binary digits match C if the corresponding digits of V are unset. The binary digit places that are set in V are unconstrained in the elements of the produced set.

Name	Definition	C++	Description
Multiple	$\{x \in \mathbb{Z} \mid mod(x, a) = 0\}$	`constrain_multiple<a>`	A set of values that are multiples of `a`.
Constraint of	$[min(T), max(T)]$	`constraint_of<T>`	Constraint interval of an integral representation.
Empty	$\emptyset$	`constraint_empty`	The empty set, no values.
Invalid	$\perp$	`constraint_invalid`	An invalid set. A set that would have otherwise contained values that cannot be represented.

6.3.2 Operations

Name	Definition	C++	Description
Subset	$A \subseteq B$	`A <= B`	Test if A is a subset of B.
Superset	$A \supseteq B$	`A >= B`	Test if A is a superset of B.
Set Equality	$A = B$	`A == B`	Test if A and B contain identical elements.
Set Inequality	$A \neq B$	`A != B`	Test if A and B do not contain identical elements.
Set Union	$A \cup B$	`A || B`	Set of all elements in A and B.
Set Intersection	$A \cap B$	`A && B`	Set of common elements in A and B.
Set Difference	$A - B$	`A && !B`	Set of elements in A and not B.
Addition	$\{a + b \mid a \in A, b \in B\}$	`A + B`	Set of product pairs of A and B added.
Subtraction	$\{a - b \mid a \in A, b \in B\}$	`A - B`	Set of product pairs of A and B subtracted.
Multiplication	$\{a * b \mid a \in A, b \in B\}$	`A * B`	Set of product pairs of A and B multiplied.
Division	$\{a \ / \ b \mid a \in A, b \in B\}$	`A / B`	Set of product pairs of A and B divided.
Remainder	$\{a \ \% \ b \mid a \in A, b \in B\}$	`A % B`	Set of product pairs of A and B remainder.
Absolute Value	$\{\|a\| \mid a \in A\}$	`abs(A)`	Set of the absolute value of all elements in A.
Minimum Value	$\{min(a, b) \mid a \in A, b \in B\}$	`min(A, B)`	Set of the minimum of each product pair of A and B.
Maximum Value	$\{max(a, b) \mid a \in A, b \in B\}$	`max(A, B)`	Set of the maximum of each product pair of A and B.
Bitwise AND	$\{a \ \& \ b \mid a \in A, b \in B\}$	`A & B`	Set of product pairs of A and B bitwise ANDed.
Bitwise OR	$\{a \mid b \mid a \in A, b \in B\}$	`A | B`	Set of product pairs of A and B bitwise ORed.
Bitwise XOR	$\{a \ \oplus \ b \mid a \in A, b \in B\}$	`A ^ B`	Set of product pairs of A and B bitwise XORed.
Bitwise NOT	$\{\neg a \mid a \in A\}$	`~A`	Bitwise NOT of all elements in A.
Bitwise Shift Left	$\{a \ << \ b \mid a \in A, b \in B\}$	`A << B`	Set of product pairs of A and B bitwise shifted left.
Bitwise Shift Right	$\{a \ >> \ b \mid a \in A, b \in B\}$	`A >> B`	Set of product pairs of A and B bitwise shifted right.

6.4 Algorithms

6.5 Random Access Container Integration

7 Design Specification

7.1 Constraint DSL

7.1.1 Operands

7.1.1.1 Constrain Interval

```
template<integral auto MinValue, integral auto MaxValue>
struct constrain_interval_t {};

template<integral auto MinValue, integral auto MaxValue>
constexpr auto constrain_interval = constrain_interval_t<MinValue, MaxValue>{};
```

7.1.1.2 Constrain Set

```
template<integral auto... Vs>
struct constrain_set_t {};

template<integral auto... Vs>
constexpr auto constrain_set = constrain_set_t<Vs...>{};
```

7.1.1.3 Constrain Mask

```
template <integral auto V, integral auto C>
struct constrain_mask_t {};

template <integral auto V, integral auto C>
constexpr auto constrain_mask = constrain_mask_t<V, C>{};
```

7.1.1.4 Constrain Multiple

```
template <integral auto M>
struct constrain_multiple_t {};

template <integral auto M>
constexpr auto constrain_multiple = constrain_multiple_t<M>{};
```

7.1.1.5 Constraint of

```
template <integral T>
constexpr auto constraint_of =
    constrain_interval<T>(
        numeric_limits<T>::lowest(),
        numeric_limits<T>::max());
```

As the standard integrals have finite bounds, there is no concept of a universal set of all integer values. `constraint_of` is provided so that bounds of standard integral types can be used instead of the universal set for set difference.

7.1.1.6 Empty

```
struct constraint_empty_t {};
constexpr auto constraint_empty = constraint_empty_t{};
```

An empty constraint contains no possible values.

Given valid constraint A, the following equivalences hold true:

$$A \cup \emptyset \equiv A$$

$$A \cap \emptyset \equiv \emptyset$$

Given valid constraint A, any binary constraint operation op except for union and intersection, the following equivalences hold true:

$$op(A, \emptyset) \equiv \emptyset$$

$$op(\emptyset, A) \equiv \emptyset$$

7.1.1.7 Invalid

```
struct constraint_invalid_t {};
constexpr auto constraint_invalid = constraint_invalid_t{};
```

An invalid constraint is used to represent the result of any constraint operation in which a subset of the result is not representable using a C++ integral. The result of any operation in which at least one operand is `constraint_invalid` is in turn `constraint_invalid`.

Given any unary constraint operation op, the following equivalence holds true:

$$op(\bot) \equiv \bot$$

Given any constraint A, any binary constraint operation op, the following equivalences hold true:

$$op(A, \bot) \equiv \bot$$

$$op(\bot, A) \equiv \bot$$

7.1.2 Operations

```
// union
constexpr auto operator||(any_constraint auto, any_constraint auto) -> any_constraint auto;

// intersection
constexpr auto operator&&(any_constraint auto, any_constraint auto) -> any_constraint auto;

// inverse
constexpr auto operator!(any_constraint auto) -> any_constraint auto;

// usual arithmetic meaning
constexpr auto operator+(any_constraint auto, any_constraint auto) -> any_constraint auto;
```

```
constexpr auto operator-(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator*(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator/(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator%(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator&(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator|(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator^(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator~(any_constraint auto) -> any_constraint auto;
constexpr auto operator<<(any_constraint auto, any_constraint auto) -> any_constraint auto;
constexpr auto operator>>(any_constraint auto, any_constraint auto) -> any_constraint auto;
```

```
// equality (==, !=)
constexpr auto operator==(any_constraint auto, any_constraint auto) -> bool;

// superset >=
constexpr auto operator>=(any_constraint auto, any_constraint auto) -> bool;

// subset <=
constexpr auto operator<=(any_constraint auto, any_constraint auto) -> bool;
```

7.2 Constrained Number

```
template<
    any_constraint auto C,
    integral T = integral_type_for_t<C>>
struct constrained_number {
    constexpr static auto constraint = C;

    constexpr constrained_number()
    requires (constraint >= constrain_set<T{}>);

    template <typename U>
    constexpr constrained_number(__constraint_cast_ferry<U> ferry);

    template <integral U>
    constexpr constrained_number(U rhs)
    requires (constraint >= constraint_of<U>);

    template <integral U, U rhs>
    constexpr constrained_number(integral_constant<U, rhs>)
    requires (constraint >= constrain_set<rhs>);

    template <any_constrained U>
    constexpr constrained_number(U const & rhs)
    requires (constraint >= U::constraint);

    template <any_constrained U>
    constexpr auto operator=(U const & rhs) -> constrained_number &
    requires (constraint >= U::constraint);

    template <integral U>
    constexpr operator U()
    requires (constraint <= constraint_of<U>);
```

```
};

constexpr auto operator+(any_constrained auto a, any_constrained auto b) -> any_constrained auto;
constexpr auto operator-(any_constrained auto, any_constrained auto) -> any_constrained auto;
constexpr auto operator*(any_constrained auto, any_constrained auto) -> any_constrained auto;
constexpr auto operator<<(any_constrained auto, any_constrained auto) -> any_constrained auto;
constexpr auto operator>>(any_constrained auto, any_constrained auto) -> any_constrained auto;

template<any_constrained L, any_constrained R>
constexpr auto operator/(L, R)
requires !(constrain_set<0> <= R::constraint) -> any_constrained auto;

template<any_constrained L, any_constrained R>
constexpr auto operator%(L, R)
requires !(constrain_set<0> <= R::constraint) -> any_constrained auto;

constexpr auto operator<=>(any_constrained auto, any_constrained auto) -> strong_ordering;
```

7.2.1 Make Constrained

```
template<any_constraint auto C, std::integral T = integral_type_for_t<C>>
constexpr auto make_constrained(auto value);
```

7.2.2 Constraint Cast

```
template<any_constrained To, typename From>
requires (std::integral<From> or any_constrained<From>)
constexpr To constraint_cast(From from);
```

7.3 Random Access Container/View Integration

```
template<
    class T,
    size_t N
> struct array {

    ...

    constexpr reference at( constrained_number<constrain_interval<0, N - 1>> pos ) noexcept;
    constexpr const_reference at( constrained_number<constrain_interval<0, N - 1>> pos ) noexcept const;

    // OR should it be operator[] instead??
    // OR should it be both??

    constexpr reference operator[]( constrained_number<constrain_interval<0, N - 1>> pos ) noexcept;
    constexpr const_reference operator[]( constrained_number<constrain_interval<0, N - 1>> pos ) noexcept

    ...

};

template<
    class T,
```

```
    size_t Extent = dynamic_extent
> class span {

    ...

    constexpr reference at( constrained_number<constrain_interval<0, Extent - 1>> pos ) noexcept const
    requires (Extent != dynamic_extent);

    // OR should it be operator[] instead??
    // OR should it be both??

    constexpr reference operator[]( constrained_number<constrain_interval<0, Extent - 1>> pos ) noexcept
    requires (Extent != dynamic_extent);

    ...

};
```

7.4 Algorithms

```
template< class UnaryFunc >
UnaryFunc for_each( any_constrained auto first, any_constrained auto last, UnaryFunc f );
```

8 Design Decisions

Alternative strategies that could have been used, but were decided against.

9 Run-time Performance

Analysis of run-time performance along with code gen.

10 Examples

Full examples of using constrained numbers to solve problems.

11 Theory

11.1 Set Constraints

11.1.1 Intervals

11.1.2 Bit Masks

11.1.3 Common Multiples

11.2 Subset/Superset Evaluation

11.2.1 Subset is Left Distributive Over Union of Disjoint Sets

For sets A, B, and C, where B and C are disjoint, the following holds true:

$A \subseteq (B \cup C) \equiv (A \subseteq B) \vee (A \subseteq C)$

Step	Operation	Justification
1	Given $A \subseteq (B \cup C)$	
2	$x \in A \implies x \in (B \cup C)$	Definition of subset
3	$x \in A \implies (x \in B \vee x \in C)$	Definition of union
4	$(x \in A \implies x \in B) \vee (x \in A \implies x \in C)$	Implication is left distributive over disjunction
5	$(A \subseteq B) \vee (A \subseteq C)$	Definition of subset
6	$\therefore A \subseteq (B \cup C) \equiv (A \subseteq B) \vee (A \subseteq C)$	

11.2.2 Subset is Right Anti-Distributive over Union of Disjoint Sets

For sets A, B, and C, the following holds true:

$(A \cup B) \subseteq C \equiv (A \subseteq C) \wedge (B \subseteq C)$

Step	Operation	Justification
1	Given $(A \cup B) \subseteq C$	
2	$x \in (A \cup B) \implies x \in C$	Definition of subset
3	$(x \in A \vee x \in B) \implies x \in C$	Definition of union
4	$(x \in A \implies x \in C) \wedge (x \in B \implies x \in C)$	Implication is right anti-distributive over disjunction
5	$(A \subseteq C) \wedge (B \subseteq C)$	Definition of subset
6	$\therefore (A \cup B) \subseteq C \equiv (A \subseteq C) \wedge (B \subseteq C)$	

11.2.3 Subsets of Unions of Disjoint Sets

For disjoint intervals A and B, and disjoint intervals C and D, the following holds true:

$(A \cup B) \subseteq (C \cup D) \equiv (A \subseteq C) \vee (A \subseteq D) \wedge (B \subseteq C) \vee (B \subseteq D)$

Step	Operation	Justification
1	Given $(A \cup B) \subseteq (C \cup D)$	
2	$A \subseteq (C \cup D) \wedge B \subseteq (C \cup D)$	Subset is Right Anti-Distributive over Union of Disjoint Intervals
3	$(A \subseteq C) \vee (A \subseteq D) \wedge (B \subseteq C) \vee (B \subseteq D)$	Subset is Left Distributive Over Union of Disjoint Intervals
4	$\therefore (A \cup B) \subseteq (C \cup D) \equiv (A \subseteq C) \vee (A \subseteq D) \wedge (B \subseteq C) \vee (B \subseteq D)$	

11.2.4 Cartesian Product of Unions

$(A \cup B) \times (C \cup D) \equiv (A \times C) \cup (B \times D) \cup (A \times D) \cup (B \times C)$

Cartesian Product of Unions

11.2.5 Cartesian Product is Distributive Over Union

$A \times (B \cup C) \equiv (A \times B) \cup (A \times C)$

$(B \cup C) \times A \equiv (B \times A) \cup (C \times A)$

Cartesian Product Distributes over Union

12 Conclusion

13 References

[AVAREZ-2015] Edgar Avarez. 2015. To keep a Boeing Dreamliner flying, reboot once every 248 days. https://www.engadget.com/2015-05-01-boeing-787-dreamliner-software-bug.html

[COREGUIDE-2023] Bjarne Stroustrup and Herb Sutter. 2015. C++ Core Guidelines. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-philosophy

[CORFIELD-2015] Gareth Corfield. 2020. Boeing 787s must be turned off and on every 51 days to prevent "misleading data" being shown to pilots. https://www.theregister.com/2020/04/02/boeing_787_power_cycle_51_days_stale_data/

[CVE-2023] 2023. CVE List V5. https://github.com/CVEProject/cvelistV5

[LIONS-1996] J. L. Lions. 1996. ARIANE 5 Failure - Full Report. https://web.archive.org/web/20140426233419/http://www.ima.umn.edu/~arnold/disasters/LIONS-1996rep.html

[MARIANI-2018] Rico Mariani. 2018. The Pit of Success. https://ricomariani.medium.com/the-pit-of-success-cfefc6cb64c8