Reintroduction to Generic Programming

Nick DeMarco
All code is a liability.
All code is a liability.
Let's write less.
Make it generic!

```javascript
/// Arranges the elements of `c` in non-decreasing order.
function sort(c: number[]) { /* ... */ }
```
Make it generic!

```javascript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }
```
The spectrum of generality

sort(c: number[])  
*One type is valid*

sort<C>(c: C)  
*Any type is valid*
The spectrum of generality

- `sort(c: number[])`<br>  One type is valid
- `sort<T>(c: T[])`<br>  Any built-in array is valid
- `sort<C>(c: C)`<br>  Any type is valid
The spectrum of generality

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }
```
The spectrum of generality

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }

let numArray = new Int32Array([3, 2, 1]);
```
The spectrum of generality

/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }

let numArray = new Int32Array([3, 2, 1]);
sort(numArray);
The spectrum of generality | That's too constrained!

/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }

let numArray = new Int32Array([3, 2, 1]);
sort(numArray);

Argument of type 'Int32Array' is not assignable to parameter of type 'T[]'.

Type 'Int32Array' is missing the following properties from type 'T[]': pop, push, concat, shift, and 3 more
The spectrum of generality

- `sort(c: number[])`: One type is valid
- `sort<T>(c: T[])`: Any built-in array is valid
- `sort<C>(c: C)`: Any type is valid
The spectrum of generality

/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
  /*...*/
}
The spectrum of generality

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
    // Bubble sort implementation to demonstrate.
    for (let i = 0; i < c.length; i++) {
        for (let j = 0; j < c.length - i - 1; j++) {
            if (!(c[j] < c[j + 1])) {
                [c[j], c[j + 1]] = [c[j + 1], c[j]];
            }
        }
    }
}
```
The spectrum of generality | That's too generic!

```javascript
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
    // Bubble sort implementation to demonstrate.
    for (let i = 0; i < c.length; i++) {
        for (let j = 0; j < c.length - i - 1; j++) {
            if (!c[j] < c[j + 1]) {
                [c[j], c[j + 1]] = [c[j + 1], c[j]];
            }
        }
    }
}
```

- Property 'length' does not exist on type 'C'.
- No index signature with a parameter of type 'number' was found on type 'C'.
The spectrum of generality

- `sort(c: number[])` (Constrained): One type is valid
- `sort<T>(c: T[])` (Generic): Any built-in array is valid
- `sort<C>(c: C)` (Generic): Any type is valid

Any type with the required functionality is valid
The spectrum of generality

- `sort(c: number[])`: One type is valid
- `sort<T>(c: T[])`: Any built-in array is valid
- `sort<C>(c: C)`: Any type is valid
- `sort<???>((c: ???)`: Any type with the required functionality is valid
The spectrum of generality | Precise constraints

/// Arranges the elements of `c` in non-decreasing order.
function sort<???><??>(c: ???) { ... }

The spectrum of generality | Precise constraints

/// Arranges the elements of `c` in non-decreasing order.

function sort<C>(c: C) { ... }
The spectrum of generality | Precise constraints

/// Arranges the elements of `c` in non-decreasing order.
function sort<C extends {}>(c: C) { ... }
The spectrum of generality | Precise constraints

```javascript
/// Arranges the elements of `c` in non-decreasing order.
function sort<C extends {length: number}>(c: C) { ... }
```
The spectrum of generality | Precise constraints

/// Arranges the elements of `c` in non-decreasing order.

function sort<C extends {length: number, [n: number]: ???}>(c: C) { ... }
The spectrum of generality | Precise constraints

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {length: number, [n: number]: T}>(c: C) { ... }
```
The spectrum of generality | Precise constraints

```javascript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {length: number, [n: number]: T}>(c: C) {
    for (let i = 0; i < c.length; i++) {
        for (let j = 0; j < c.length - i - 1; j++) {
            if (!(c[j] < c[j + 1])) {
                [c[j], c[j + 1]] = [c[j + 1], c[j]];
            }
        }
    }
}
```
The spectrum of generality | Precise constraints

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) {
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```
The spectrum of generality | Precise constraints

```javascript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) {
    for (let i = 0; i < c.length; i++) {
        for (let j = 0; j < c.length - i - 1; j++) {
            if (!(c[j] < c[j + 1])) {
                [c[j], c[j + 1]] = [c[j + 1], c[j]];
            }
        }
    }
}
```
The spectrum of generality | That's just right!

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) { ... }

let numArray = new Int32Array([3, 2, 1]);
sort(numArray); // [1, 2, 3] ✔

let stringArray = ["c", "d", "a"];
sort(stringArray); // ["a", "c", "d"] ✔

let unsortable: Record<string, string> = { first: "Jane", last: "Doe" };
sort(unsortable); // Error: Property 'length' is missing... ✔
The spectrum of generality

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}> (c: C) { ... }

Any type with the required functionality is valid
Key takeaway 1/8:
Maximize reusability with minimally* complete constraints
/// Arranges the elements of `c` in non-decreasing order.

function sort<T, C extends {readonly length: number, [n: number]: T}> ...
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}> ... 

/// Arranges the elements in `c` from `first` to `last` in non-decreasing order.
function sortSubrange<T, C extends {readonly length: number, [n: number]: T}> ...
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}> ...

/// Arranges the elements in `c` from `first` to `last` in non-decreasing order.
function sortSubrange<T, C extends {readonly length: number, [n: number]: T}> ...

/// Partitions `c` around `nth`, leaving the value of `nth` as if `c` were entirely sorted.
function nthElement<T, C extends {readonly length: number, [n: number]: T}> ...
Naming our constraints

```typescript
interface ???<T> = {
    readonly length: number,
    [n: number]: T
};
```

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends ???> ...
```
Naming our constraints

```typescript
interface ??<T> = {
  readonly length: number,
  [n: number]: T
};
```

MDN web docs

`Array.prototype.sort()`

The `sort()` method is generic. It only expects the this value to have a length property and integer-keyed properties.

Although strings are also array-like, this method is not suitable to be applied on them, as strings are immutable.
Naming our constraints

interface ???<T> = {
    readonly length: number,
    [n: number]: T
};

Array.prototype.sort()

The sort() method is generic. It only expects the this value to have a length property and integer-keyed properties.

Although strings are also array-like, this method is not suitable to be applied on them, as strings are immutable.
interface ???<T> = {
  readonly length: number,
  [n: number]: T
};

Array.prototype.sort()

The sort() method is generic. It only expects the this value to have a length property and integer-keyed properties.

Although strings are also array-like, this method is not suitable to be applied on them, as strings are immutable.
Naming our constraints

interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
interface MutableArrayLike<T> = {
    readonly length: number,
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
interface MutableArrayLike<T> = {
    readonly length: number,
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...

About 119,000 results (0.26 seconds)

Stack Overflow

Unable to understand ArrayLike interface in Typescript
Jan 18, 2019 — I am struggling to understand what [n:number]:T means. Is this declaring an array of type T and the size of the array is n?
1 answer · Top answer: It's an index signature. An object can be indexed using a string or a nu...

How to implement a "ArrayLike" class in TypeScript?  Feb 17, 2022
Declare TypedArray with ArrayLike?  TypeScript  Stack Overflow  Oct 25, 2022

Typescript array-like type with same generic type between ...  Oct 23, 2019

TypeScript: is there an interface for array-like object?  May 24, 2019

More results from stackoverflow.com

Microsoft Open Source
https://microsoft.github.io/interfaces/

ArrayLike  TypeScript  v3.7.7
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...

interface ArrayLike<T> {
  length: number;
  [n: number]: T;
}

I recently found about `ArrayLike` interface

I am struggling to understand what `[n:number]: T` means. Is this declaring an array of type `T` and the size of the array is `n`?
interface MutableArrayLike<T> = {
    readonly length: number,
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...

Compare this to this interface:

interface ArrayLike<T> {
    length: number;
    [n: number]: T;
}

I am struggling to understand what `[n:number]:T` means. Is this declaring an array of type `T` and the size of the array is `n`?
interface MutableArrayLike<T> = {
    readonly length: number,
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
Documenting semantics

/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
    readonly length: number,
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
Documenting semantics

/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
    /// The number of elements
    readonly length: number,
    /// Gets the `n`th element
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
Documenting semantics

```typescript
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
    /// The number of elements
    readonly length: number,
    /// Gets the `n`th element
    [n: number]: T
};
```

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```
Key takeaway 2/8:

Extract common constraints and document their semantics
What is this thing?

/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
    /// The number of elements
    readonly length: number,
    /// Gets the `n`th element
    [n: number]: T
};
We call the set of axioms satisfied by a data type and a set of operations on it a concept.

The critical insight... is that highly reusable components must be programmed assuming a minimal collection of such concepts, [which] must match as wide a variety of concrete program structures as possible.

Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components.
We call the set of axioms satisfied by a data type and a set of operations on it a concept.

The critical insight... is that highly reusable components must be programmed assuming a minimal collection of such concepts, [which] must match as wide a variety of concrete program structures as possible.

Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components.
We call the set of axioms satisfied by a data type and a set of operations on it a concept.

The critical insight... is that highly reusable components must be programmed assuming a minimal collection of such concepts, [which] must match as wide a variety of concrete program structures as possible.

Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components.
Concepts

/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
    /// The number of elements
    readonly length: number,
    /// Gets the `n`th element
    [n: number]: T
};

/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...

/// Arranges the elements in `c` from `first` to `last` in non-decreasing order.
function sortSubrange<T, C extends MutableArrayLike<T>> ...

/// Partitions `c` around `nth`, leaving the value of `nth` as if `c` were entirely sorted.
function nthElement<T, C extends MutableArrayLike<T>> ...
Concepts | Finding common behavior

/// A sequence of `T` stored non-contiguously

template <class T>
struct linked_list { ... };
Concepts | Finding common behavior

/// A sequence of `T` stored non-contiguously

template <class T>
struct linked_list {
    struct node {
        std::unique_ptr<node> _next = nullptr;
        T _value;
    };
    std::unique_ptr<node> _head = nullptr;
};
Concepts | Finding common behavior

/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list { ... };

/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t { ... }
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list { ... };

/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t { ... }

(T a, T b) {
    a == b; // is legal & meaningful
    a != b; // is legal & meaningful
}
Concepts | Finding common behavior

/// A sequence of `T` stored non-contiguously

```cpp
template <class T>
struct linked_list {
...
};
```

/// Returns the frequency of `t` in the linked list.

```cpp
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t {
    std::size_t count{0};
    for (auto i = list._head.get(); i != nullptr; i = i->_next.get())
        if (i->_value == t) count++;
    return count;
}
```
Concepts | Finding common behavior

```cpp
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input { ... };
```
Concepts | Finding common behavior

/// A stream of `T` from stdin.

```cpp
template <input_streamable T>
class user_input {
    ... 
};
```

```cpp
(std::istream& stream, T t) {
    stream >> t; // is legal & meaningful
}
```
Concepts | Finding common behavior

/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
    /// Returns one `T` read from std::cin, or `std::nullopt` on failure.
    auto read() -> std::optional<T> { ... } 
};
Concepts | Finding common behavior

/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
    /// Returns one `T` read from std::cin, or `std::nullopt` on failure.
    auto read() -> std::optional<T> { ... }
};

/// Returns the frequency of `t` in the user input stream.
template <std::equality_comparable T>
auto count(T t, user_input<T>& in) -> std::size_t { ... }
Concepts | Finding common behavior

/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
    /// Returns one `T` read from std::cin, or `std::nullopt` on failure.
    auto read() -> std::optional<T> { ... }
};

/// Returns the frequency of `t` in the user input stream.
template <std::equality_comparable T>
auto count(T t, user_input<T>& in) -> std::size_t {
    std::size_t count{0};
    for (auto i = in.read(); i != std::nullopt; i = in.read())
        if (*i == t) count++;
    return count;
}
Concepts | Finding common behavior

/// Returns the frequency of `t` in the user input stream.
template <std::equality_comparable T>
auto count(T t, user_input<T>& in) -> std::size_t {
    std::size_t count{0};
    for (auto i = in.read(); i != std::nullopt; i = in.read())
        if (*i == t) count++;
    return count;
}

/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t {
    std::size_t count{0};
    for (auto i = list._head.get(); i != nullptr; i = i->_next.get())
        if (i->_value == t) count++;
    return count;
}
Lifting Concepts

// linked_list<T>
for (auto i = list._head.get(); i != nullptr; i = i->_next.get())
  if (i->_value == t) count++;

// user_input<T>
for (auto i = in.read(); i != std::nullopt; i = in.read())
  if (*i == t) count++;

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}

Lifting Concepts

template <class I>
concept ??? = requires(I i) { ... };

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
Lifting Concepts

template <class I>
concept ??? = requires(I i) { ... };

template <class I>
constexpr bool ??? =
   /* some predicates over I */;

// Generally,
for (auto i = the_beginning(), i := the_end(); get_next(i)) {
   if (get_value(i) == t) count++;
}
Lifting Concepts

template <class I>
concept ??? = requires(I i) { ... };

(I i) {
   /* ... these operations on i are legal & meaningful */
}

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
   if (get_value(i) == t) count++;
}
Lifting Concepts

template <class I>
concept ??? = requires(I i) { ... };

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
Lifting Concepts

template <class I>
concept ??? = requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
};

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
template <class I>
concept ??? = requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> ???;
};

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
Lifting Concepts

template <class I, class T>
concept ??? = requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> T&;
};

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
template <class I, class T> concept ??? =

std::equality_comparable &&
requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> T&;
};

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
Lifting Concepts

template <class I, class T>
concept ??? =
    std::equality_comparable &&
    requires(I i) {
        /// Returns the next valid position or an invalid sentinel for the end.
        { ++i } -> I&;
        /// Returns the value at this position; invalid iff this position is invalid.
        { *i } -> T&;
    };

    // Generally,
    for (auto i = the_beginning(); i != the_end(); get_next(i)) {
        if (get_value(i) == t) count++;
    }
Lifting Concepts

/// A position in a sequence of `T`.

```
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        /// Returns the next valid position or an invalid sentinel for the end.
        { ++i } -> I&;
        /// Returns the value at this position; invalid iff this position is invalid.
        { *i } -> T&;
    };
```

// Generally,
```
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
```
Lifting Concepts

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        /// Returns the next valid position or an invalid sentinel for the end.
        { ++i } -> std::same_as<I&>;
        /// Returns the value at this position; invalid iff this position is invalid.
        { *i } -> std::same_as<T&>;
    };

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
Modeling concepts

// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
    struct node {
        unique_ptr<node> _next; T _value;
    };
    unique_ptr<node> _head = nullptr;

    struct list_iterator { ... };
};

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}
Modeling concepts

/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
    struct node { unique_ptr<node> _next; T _value; };
    unique_ptr<node> _head = nullptr;

    struct list_iterator { ... };

    auto begin() { return list_iterator(_head.get()); }  \(\textcolor{green}{\text{green}}\)
    auto end()   { return list_iterator(nullptr); }  \(\textcolor{green}{\text{green}}\)  // "invalid sentinel for the end."
};

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
}

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };

Modeling concepts

/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node { unique_ptr<node> _next; T _value; };
  unique_ptr<node> _head = nullptr;

  struct list_iterator {
    node* _node;
    explicit list_iterator(node* n) : _node(n) {}
  };

  auto begin() { return list_iterator(_head.get()); }
  auto end()   { return list_iterator(nullptr); } // "invalid sentinel for the end."
};

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
  };

Modeling concepts

/// A sequence of `T` stored non-contiguously

```cpp
template <class T>
struct linked_list {
    struct node { unique_ptr<node> _next; T _value; };
    unique_ptr<node> _head = nullptr;

    struct list_iterator {
        node* _node;
        explicit list_iterator(node* n) : _node(n) {}  
        auto& operator++() { }  
        auto& operator*() { }  
        bool operator==(const list_iterator& rhs) const = default;
    };
    auto begin() { return list_iterator(_head.get()); }  
    auto end()   { return list_iterator(nullptr); }  // "invalid sentinel for the end."
};
```

/// A position in a sequence of `T`.

```cpp
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;  
        { *i } -> std::same_as<T&>;
    };
```
Modeling concepts

/// A sequence of `T` stored non-contiguously

template <class T>
struct linked_list {
    struct node { unique_ptr<node> _next; T _value; }
    unique_ptr<node> _head = nullptr;

    struct list_iterator {
        node* _node;
        explicit list_iterator(node* n) : _node(n) {} 
        auto& operator++() { _node = _node->next.get(); return *this; }
        auto& operator*() { return _node->value; }
        bool operator==(const list_iterator& rhs) const = default;
    };

    auto begin() { return list_iterator(_head.get()); }
    auto end()   { return list_iterator(nullptr); } // "invalid sentinel for the end."
};
Modeling concepts

/// A sequence of `T` stored non-contiguously

template <class T>
struct linked_list {
  auto begin() { return list_iterator(_head.get()); }
  auto end() { return list_iterator(nullptr); }
};

/// A stream of `T` from stdin.

template <input_streamable T>
class user_input {
  auto begin() { return user_input_iterator(std::addressof(std::cin)); }
  auto end() { return user_input_iterator(nullptr); }
};
Modeling concepts

/// A sequence of `T` stored non-contiguously

```cpp
template <class T>
struct linked_list {
    iterator<T> auto begin() { return list_iterator(_head.get()); }
    iterator<T> auto end() { return list_iterator(nullptr); }
};
```

/// A stream of `T` from stdin.

```cpp
template <input_streamable T>
class user_input {
    iterator<T> auto begin() { return user_input_iterator(std::addressof(std::cin)); }
    iterator<T> auto end() { return user_input_iterator(nullptr); }
};
```

See: https://tinyurl.com/GCCIstreamIterator
Modeling concepts

/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
    iterator<T> auto begin() { ... }
    iterator<T> auto end() { ... }
};

/// Returns the frequency of `t` in the sequence.
template <std::equality_comparable T, iterator<T> Iter>
auto count(T t, Iter begin, Iter end) -> std::size_t { ... }

/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
    iterator<T> auto begin() { ... }
    iterator<T> auto end() { ... }
};
Lifting concepts (again)

/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
    iterator<T> auto begin() { ... }
    iterator<T> auto end() { ... }
};

/// Returns the frequency of `t` in the sequence.
template <std::equality_comparable T, iterator<T> Iter>
auto count(T t, Iter begin, Iter end) -> std::size_t { ... }

/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
    iterator<T> auto begin() { ... }
    iterator<T> auto end() { ... }
};
Lifting concepts (again)

/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
    /// Returns an iterator to the first element.
    iterator<T> auto begin() { ... }
    /// Returns a sentinel representing the end.
    iterator<T> auto end() { ... }
};

template <class R, class T>
concept ??? = requires(R r) { ... };
### Lifting concepts (again)

```cpp
template <class R, class T>
concept ??? = requires(R r) {
    /// Returns an iterator to the first element.
    iterator<T> auto begin() { ... } 
    /// Returns a sentinel representing the end.
    iterator<T> auto end() { ... } 
};

template <class T>
struct linked_list {
    /// Returns an iterator to the first element.
    iterator<T> auto begin() { ... }
    /// Returns a sentinel representing the end.
    iterator<T> auto end() { ... }
};

template <class R, class T>
concept ??? = requires(R r) {
    /// Returns an iterator to the first element.
    { r.begin() } -> iterator<T>;
    /// Returns a sentinel representing the end.
    { r.end() } -> iterator<T>;
};

template <input_streamable T>
class user_input {
    /// Returns an iterator to the first element.
    iterator<T> auto begin() { ... }
    /// Returns a sentinel representing the end.
    iterator<T> auto end() { ... }
};
```

/// A sequence of `T` stored non-contiguously
```cpp
template <class T>
struct linked_list {
    // Returns an iterator to the first element.
    iterator<T> auto begin() { ... }
    // Returns a sentinel representing the end.
    iterator<T> auto end() { ... }
};
```

/// A stream of `T` from stdin.
```cpp
template <input_streamable T>
class user_input {
    // Returns an iterator to the first element.
    iterator<T> auto begin() { ... }
    // Returns a sentinel representing the end.
    iterator<T> auto end() { ... }
};
```
Lifting concepts (again)

/// A sequence of `T` stored non-contiguously
-template <class T>
-struct linked_list {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};

/// A stream of `T` from stdin.
-template <input_streamable T>
-class user_input {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
template <class R, class T>
concept range = requires(R r) {
    { r.begin() } -> iterator<T>;
    { r.end() } -> iterator<T>;
};

static_assert(range<int[42], int>);
Lifting concepts (again)

/// A sequence of `T`

```cpp
template <class R, class T>
concept range = requires(R r) {
    /// Returns an iterator to the first element.
    { r.begin() } -> iterator<T>;
    /// Returns a sentinel representing the end.
    { r.end() } -> iterator<T>;
};
```

```cpp
static_assert(range<int[], int>);
```

```cpp
static_assert(range<int[42], int>);
```

```
error: static assertion failed
static_assert(range<int[], int>);
  ^~~~~~~~~~~~~~~~~~~
... because 'range<int[42], int>' evaluated to false
... because 'r.begin()' would be invalid: 'int *' is not a structure or union
{ r.begin() } -> iterator<T>;
```

```cpp
static_assert(range<int[42], int>); // ✗ Sized array-of-int is not a range.
```
Lifting concepts (again)

/// A sequence of `T`

```cpp
template <class R, class T>
concept range = requires(R r) {
    /// Returns an iterator to the first element.
    { begin(r) } -> iterator<T>;
    /// Returns a sentinel representing the end.
    { end(r) } -> iterator<T>;
};
```
Lifting concepts (again)

/// A sequence of `T`  

```cpp
template <class R, class T>
concept range = requires(R r) {
    /// Returns an iterator to the first element.
    { begin(r) } -> iterator<T>;
    /// Returns a sentinel representing the end.
    { end(r) } -> iterator<T>;
};
```

```cpp
template <class R>
auto begin(R& r) -> decltype(r.begin()) { ... }
```

```cpp
template <class T, std::size_t N>
auto begin(T(&array)[N]) -> T* { ... }
```
Lifting concepts (again)

```cpp
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
    /// Returns an iterator to the first element.
    { begin(r) } -> iterator<T>;
    /// Returns a sentinel representing the end.
    { end(r) } -> iterator<T>;
};

template <class R>
auto begin(R& r) -> decltype(r.begin) { ... }

template <class T, std::size_t N>
auto begin( T(&array)[N] ) -> T* { ... }

static_assert(range<int[42], int>); // sized array-of-int is a range.
```
Generic algorithms in terms of concepts

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };

/// Returns the frequency of `t` in the range.
template <std::equality_comparable T, range<T> Range>
auto count(T t, Range r) -> std::size_t { ... }

/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
    { begin(r) } -> iterator<T>;
    { end(r) } -> iterator<T>;
};
Generic algorithms in terms of concepts

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };

/// A sequence of `T`.
template <class R, class T>
concept range = requires(R r) {
    { begin(r) } -> iterator<T>;
    { end(r) } -> iterator<T>;
};

/// Returns the frequency of `t` in the range.
template <std::equality_comparable T, range<T> Range>
auto count(T t, Range r) -> std::size_t {
    std::size_t count{0};
    for (iterator<T> auto i = begin(r); i != end(r); ++i)
        if (*i == t) count++;
    return count;
}
Generic algorithms in terms of concepts

```cpp
/// A position in a sequence of `T`.
template <class I, class T>
class concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };

/// A sequence of `T`.
template <class R, class T>
class concept range = requires(R r) {
    { begin(r) } -> iterator<T>;
    { end(r) } -> iterator<T>;
};

/// Returns the frequency of `t` in the range.
template <std::equality_comparable T, range<T> Range>
auto count(T t, Range r) -> std::size_t {
    std::size_t count{0};
    for (iterator<T> auto i = begin(r); i != end(r); ++i)
        if (*i == t) count++;
    return count;
}
```
Key takeaway 3/8:

Discover *Concepts* by analyzing common semantics in algorithms
Fundamentals of Generic Programming

James C. Delahunt and Alexander Stepanov
Silicon Graphics, Inc.
delahunt@sun.sgi.com
stepanov@wabash.edu

Keywords: Generic programming, generic semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasive and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g., copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.
If we hope to reuse code [which uses] the standard C++ operators, and apply it to both built-in and user-defined types, we must extend the semantics as well as the syntax of the standard operators to user-defined types.

... concepts which match the semantics of built-in types and operators provide an excellent foundation for generic programming.

... a regular type matches the built-in type semantics, thereby making our user-defined types behave like built-in types as well.
Regular types have value semantics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default constructor</strong></td>
<td><code>T a;</code></td>
</tr>
<tr>
<td><strong>Copy constructor</strong></td>
<td><code>T a = b;</code></td>
</tr>
<tr>
<td><strong>Destructor</strong></td>
<td><code>~T(a);</code></td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
<td><code>a = b;</code></td>
</tr>
<tr>
<td><strong>Equality</strong></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td><strong>Inequality</strong></td>
<td><code>a != b</code></td>
</tr>
<tr>
<td><strong>Ordering, e.g.</strong></td>
<td><code>a &lt; b</code></td>
</tr>
</tbody>
</table>

**Fig. 1. Fundamental Operations on Type T**

Fundamentals of Generic Programming
Regular types have value semantics

| Default constructor | T a;
|---------------------|------------------|
| Copy constructor    | T a = b;
| Destructor          | ~T(a);           |
| Assignment          | a = b;
| Equality            | a == b           |
| Inequality          | a != b           |
| Ordering, e.g.      | a < b            |

Fig. 1. Fundamental Operations on Type T

Fundamentals of Generic Programming
Regular types have value semantics

### Fig. 1. Fundamental Operations on Type T

<table>
<thead>
<tr>
<th>Default constructor</th>
<th>T a;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td>T a = b;</td>
</tr>
<tr>
<td>Destructor</td>
<td>~T(a);</td>
</tr>
<tr>
<td>Assignment</td>
<td>a = b;</td>
</tr>
<tr>
<td>Equality</td>
<td>a == b</td>
</tr>
<tr>
<td>Inequality</td>
<td>a != b</td>
</tr>
<tr>
<td>Ordering, e.g.</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

```cpp
template <class T>
concept regular =
    std::semiregular<T> &&
    std::equality_comparable<T>;

template <class T>
concept semiregular =
    std::copyable<T> &&
    std::default_initializable<T>;

// ... taxonomy 4-5 layers deep
```

Fundamentals of Generic Programming
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  /// The number of elements
  readonly length: number,
  /// Gets the `n`th element
  [n: number]: T
};
/// A position in a sequence of `T`.

template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        /// Returns the next valid position or an invalid sentinel for the end.
        { ++i } -> I&;
        /// Returns the value at this position; invalid iff this position is invalid.
        { *i } -> T&;
    };

Intertwined semantics
Intertwined semantics

```cpp
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
    /// Returns an iterator to the first element.
    { begin(r) } -> iterator<T>;
    /// Returns a sentinel representing the end.
    { end(r) } -> iterator<T>;
};
```
Intertwined semantics

**Fig. 1. Fundamental Operations on Type T**

<table>
<thead>
<tr>
<th>Default constructor</th>
<th>T a;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td>T a = b;</td>
</tr>
<tr>
<td>Destructor</td>
<td>∼T (a);</td>
</tr>
<tr>
<td>Assignment</td>
<td>a = b;</td>
</tr>
<tr>
<td>Equality</td>
<td>a == b</td>
</tr>
<tr>
<td>Inequality</td>
<td>a != b</td>
</tr>
<tr>
<td>Ordering, e.g.</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

Fundamentals of Generic Programming
Intertwined semantics

<table>
<thead>
<tr>
<th>Default constructor</th>
<th>( T \ a; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td>( T \ a = b; )</td>
</tr>
<tr>
<td>Destructor</td>
<td>( \sim T(a); )</td>
</tr>
<tr>
<td>Assignment</td>
<td>( a = b; )</td>
</tr>
<tr>
<td>Equality</td>
<td>( a == b )</td>
</tr>
<tr>
<td>Inequality</td>
<td>( a != b )</td>
</tr>
</tbody>
</table>

Fig. 1. Fundamental Operations on Type \( T \)

Fundamentals of Generic Programming

\[ T \ a = b; \]
\[ \text{assert}(a == b); \]
## Intertwined semantics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default constructor</td>
<td><code>T a;</code></td>
</tr>
<tr>
<td>Copy constructor</td>
<td><code>T a = b;</code></td>
</tr>
<tr>
<td>Destructor</td>
<td><code>~T(a);</code></td>
</tr>
<tr>
<td>Assignment</td>
<td><code>a = b;</code></td>
</tr>
<tr>
<td>Equality</td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Inequality</td>
<td><code>a != b</code></td>
</tr>
<tr>
<td>Ordering (e.g.)</td>
<td><code>a &lt; b</code></td>
</tr>
</tbody>
</table>

### Fig. 1. Fundamental Operations on Type T

- `T a = b;`  
  - `assert(a == b);`  
  - `assert(f(a) == f(b))`

**Fundamentals of Generic Programming**
**Intertwined semantics**

<table>
<thead>
<tr>
<th>Default constructor</th>
<th>T a;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td>T a = b;</td>
</tr>
<tr>
<td>Destructor</td>
<td>~T(a);</td>
</tr>
<tr>
<td>Assignment</td>
<td>a = b;</td>
</tr>
<tr>
<td>Equality</td>
<td>a == b</td>
</tr>
<tr>
<td>Inequality</td>
<td>a != b</td>
</tr>
<tr>
<td>Ordering, c.g.</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

```cpp
T a = c;
T b = c;
always_mutates(a);
assert(b==c && a != b);
```
**Intertwined semantics**

<table>
<thead>
<tr>
<th>Default constructor</th>
<th>( T \ a; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td>( T \ a = b; )</td>
</tr>
<tr>
<td>Destructor</td>
<td>(~T(a);)</td>
</tr>
<tr>
<td>Assignment</td>
<td>( a = b; )</td>
</tr>
<tr>
<td>Equality</td>
<td>( a == b )</td>
</tr>
<tr>
<td>Inequality</td>
<td>( a != b )</td>
</tr>
<tr>
<td>Ordering, e.g.</td>
<td>( a &lt; b )</td>
</tr>
</tbody>
</table>

**Fig. 1. Fundamental Operations on Type T**

_Fundamentals of Generic Programming_
### Intertwined semantics

#### Fig. 1. Fundamental Operations on Type T

<table>
<thead>
<tr>
<th>Default constructor</th>
<th>T a;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td>T a = b;</td>
</tr>
<tr>
<td>Destructor</td>
<td>~T(a);</td>
</tr>
<tr>
<td>Assignment</td>
<td>a = b;</td>
</tr>
<tr>
<td>Equality</td>
<td>a == b;</td>
</tr>
<tr>
<td>Inequality</td>
<td>a != b;</td>
</tr>
<tr>
<td>Ordering, e.g.</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

```
// ... taxonomy 4-5 layers deep
```

```cpp
template <class T>
concept regular =
    std::semiregular<T> &&
    std::equality_comparable<T>;

template <class T>
concept semiregular =
    std::copyable<T> &&
    std::default_initializable<T>;
```
std::copy_constructible

Defined in header `<concepts>`

```cpp
template< class T >
concept copy_constructible =
    std::move_constructible<T> &&
    std::constructible_from<T, T&> && std::convertible_to<T&, T> &&
    std::constructible_from<T, const T&> && std::convertible_to<const T&, T> &&
    std::constructible_from<T, const T> && std::convertible_to<const T, T>;
```

(since C++20)

The concept `copy_constructible` is satisfied if `T` is an lvalue reference type, or if it is a `move_constructible` object type where an object of that type can be constructed from a (possibly const) lvalue or const rvalue of that type in both direct- and copy-initialization contexts with the usual semantics (a copy is constructed with the source unchanged).

**Semantic requirements**

If `T` is an object type, then `copy_constructible<T>` is modeled only if given

- `v`, an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`,

the following are true:

- After the definition `T u = v;`, `u` is equal to `v` and `v` is not modified;
- `T(v)` is equal to `v` and does not modify `v`.
Key takeaway 4/8:
Define concepts with intertwined semantics.
Using our iterator

/// A monotonically increasing integer count.
struct counter { };

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };

Using our iterator

/// A monotonically increasing integer count.
struct counter {
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };
Using our `iterator`

```cpp
/// A monotonically increasing integer count.
struct counter {
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.
}
```

```cpp
template <class I, class T>
concept iterator =
    std::equality_comparable && requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };
```
Using our iterator

/// A monotonically increasing integer count.
struct counter {
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

    auto c = counter{40};
    std::ranges::advance(c, 2);
}
Using our iterator

/// A monotonically increasing integer count.
struct counter {
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        { ++i } -> std::same_as<I&>;
        { *i } -> std::same_as<T&>;
    };

int main() {
    static_assert(iterator<counter, std::size_t>);
    // ✅ counter is an iterator over std::size_t.

    auto c = counter{40};
    std::ranges::advance(c, 2);
}
Using our iterator

/// A monotonically increasing integer count.
struct counter {
    using difference_type = std::ptrdiff_t;
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

    auto c = counter{40};
    std::ranges::advance(c, 2);
}
Using our `iterator`

```cpp
/// A monotonically increasing integer count.
struct counter {
    using difference_type = std::ptrdiff_t;
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

    auto c = counter{40}; // ✅
    std::ranges::advance(c, 2);
}
```

// A position in a sequence of `T`.
```cpp
template <class I, class T>
concept iterator =
    std::equality_comparable &&
    requires(I i) {
        ++i           // std::same_as<I&>
    } &&
    requires(I i) {
        *i            // std::same_as<T&>
    };
```
The spectrum of generality

Any type with the required functionality is valid
std::input_or_output_iterator

Defined in header <iterator>

```cpp
template< class I >
concept input_or_output_iterator =
  requires(I i) {
    { *i } -> /*can-reference*/;
  } &&
  std::weakly_incrementable<I>;
```

The `input_or_output_iterator` concept forms the basis of the iterator concept taxonomy; every iterator type satisfies the `input_or_output_iterator` requirements.

---

std::weakly_incrementable

Defined in header <iterator>

```cpp
template< class I >
concept weakly_incrementable =
  std::movable<I> &&
  requires(I i) {
    typename std::iter_difference_t<I>;
    requires /*is-signed-integer-like*/<std::iter_difference_t<I>>;
    { ++i } -> std::same_as<I&>; // not required to be equality-preserving
    i++;
    // not required to be equality-preserving
  };
```
Concepts are abstract entities

template <class T>
concept Addable = requires(T t) {
    { t + t } -> std::same_as<T>;
};;

template <class T>
concept Subtractable = requires(T t) {
    { t - t } -> std::same_as<T>;
};;

template <class T>
concept Multipliable = requires(T t) {
    { t * t } -> std::same_as<T>;
};;

...etc
Concepts are abstract entities

```cpp
template <class T>
concept Addable = requires(T t) {
  { t + t } -> std::same_as<T>;
};

template <class T>
concept Subtractable = requires(T t) {
  { t - t } -> std::same_as<T>;
};

template <class T>
concept Addable = requires(T t) {
  { t / t } -> std::same_as<T>;
};

...etc
```

```cpp
template <class T>
concept Numeric = requires(T t) {
  { t + t } -> std::same_as<T>;
  { t - t } -> std::same_as<T>;
  { t * t } -> std::same_as<T>;
  { t / t } -> std::same_as<T>;
  ...etc
};
```
Using our `iterator`

/// A monotonically increasing integer count.
struct counter {
    using difference_type = std::ptrdiff_t;
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.
    auto c = counter{40};
    std::ranges::advance(c, 2); // '__i++' invalid: cannot increment value of type 'counter'
}
Using our iterator

/// A monotonically increasing integer count.
struct counter {
    using difference_type = std::ptrdiff_t;
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter>);
    auto c = counter{40};
    std::ranges::advance(c, 2); // '__i++' invalid: cannot increment value of type 'counter'
}
The spectrum of generality

Any type \textit{with the required functionality} is valid
The spectrum of generality

Any type **satisfying the required concepts** is valid

Any type with the required functionality is valid
Key takeaway 5/8:

[Over-]constrain algorithms with Concepts that model abstract entities.
/// A monotonically increasing integer count.
struct counter {
    using difference_type = std::ptrdiff_t;
    friend bool operator==(counter, counter) = default;
    auto& operator++() { ++count; return *this; }
    auto& operator*() { return count; }
    std::size_t count = 0;
};

int main() {
    static_assert(iterator<counter, std::size_t>); // counter is an iterator over std::size_t.
    auto c = counter{40};
    std::ranges::advance(c, 2); // '__i++' invalid: cannot increment value of type 'counter'
}
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}
/// Returns an iterator to the first element equal to `x`.

```cpp
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}
```

```cpp
int main() {
    std::vector<int> v{1, 2, 3};
    find(begin(v), end(v), 2); // ✅ finds the 2
}
```
WTF Concepts

/// Returns an iterator to the first element equal to `x`.

template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}

int main() {
    std::vector<int> v{1, 2, 3};
    find(begin(v), end(v), 2); // ✅ finds the 2

    std::istream_iterator first(std::cin), last;
    find(first, last, 42);
}
/// Returns an iterator to the first element equal to `x`.

template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}

int main() {
    std::vector<int> v{1, 2, 3};
    find(begin(v), end(v), 2); // ✅ finds the 2

    std::istream_iterator first(std::cin), last;
    find(first, last, 42);
}

In 'auto find(I, I, T) [with I = std::istream_iterator<char>; T = int]'
no match for 'operator<' (operand types are 'std::istream_iterator<char>'...)

    while (first < last && !(*first == value))
        ~~~~~~~~~~\~~~~~~~~~
The spectrum of generality | That's too generic!

/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
    // Bubble sort implementation to demonstrate.
    for (let i = 0; i < c.length; i++) {
        for (let j = 0; j < c.length - i - 1; j++) {
            if (!(c[j] < c[j + 1])) {
                [c[j], c[j + 1]] = [c[j + 1], c[j]];
            }
        }
    }
}

Property 'length' does not exist on type 'C'.
No index signature with a parameter of type 'number' was found on type 'C'.
/// Returns an iterator to the first element equal to `x`.

```cpp
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}
```

```cpp
int main() {
    std::vector<int> v{1, 2, 3};
    find(begin(v), end(v), 2); // ✅ finds the 2
    std::istream_iterator first(std::cin), last;
    find(first, last, 42);
}
```
/// Returns an iterator to the first element equal to `x`.

```cpp
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}
```

```cpp
int main() {
    std::vector<int> v{1, 2, 3};
    find(begin(v), end(v), 2); // ✅ finds the 2

    std::istream_iterator first(std::cin), last;
    find(first, last, 42); // No operator `<`
}
```
Archetypes

/// Returns an iterator to the first element equal to `x`.

template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(first == value))
        ++first;
    return first;
}

inline void test_find(input_iterator_archetype f,
        input_iterator_archetype l,
        equality_comparable_archetype x) {
    find(f, l, x); // No operator `<`... but caught by the author of `find`.
}
template <class T>
class input_iterator_archetype
{
private:
    typedef input_iterator_archetype self;
public:
    typedef std::input_iterator_tag iterator_category;
    typedef T value_type;
    struct reference {
        operator const value_type&() const { return static_object<T>::get(); }
    };
    typedef const T* pointer;
    typedef std::ptrdiff_t difference_type;
    self& operator=(const self&) { return *this; }
    bool operator==(const self&) const { return true; }
    bool operator!=(const self&) const { return true; }
    reference operator*() const { return reference(); }
    self& operator++() { return *this; }
    self operator++(int) { return *this; }
};
Archetypes

• Boost Concept Check Library
  • https://www.boost.org/doc/libs/1_84_0/libs/concept_check/concept_check.htm

• Andrzej Krzemieński Blog Post: Concept Archetypes
  • https://akrzemi1.wordpress.com/2020/09/02/concept-archetypes/

• "I'm skeptical that we should expect early-checked generics to land in a C++ compiler, given their inflexibility." -- Sean Baxter
  • https://github.com/seanbaxter/circle/blob/master/new-circle/README.md#generic
Stealthy Requirements

/// Returns an iterator to the first element equal to `x`.

```cpp
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
    while (first < last && !(*first == value))
        ++first;
    return first;
}
```

```cpp
int main() {
    std::vector<int> v{1, 2, 3};
    find(begin(v), end(v), 2); // ✅ finds the 2

    std::istream_iterator first(std::cin), last;
    find(first, last, 42); // No operator `<`
}
```
Key takeaway 6/8:

⚠ Beware of *stealthy requirements*
“Making maximally reusable functions isn't what's needed to ship product.”
Industrial Revolution in Software

- Large, systematic catalogs
- Validated, efficient, generic components
- Component engineers (few)
- System engineers (many)

Alexander Stepanov: STL and its Design Principles (Jan 2002)
https://youtu.be/COuHLky7E2Q
Product Code

UI Layouts

std::vector

Library Code
Product Code

UIS Layouts

Composite Graph

File Parsers

std::vector

Library Code

User preferences

Event Handling

Mouse tracker

File Parsers
All code is a liability.
Key takeaway 7/8:

Generalize your stable code before it rots.
Lehman's laws of software evolution

1 Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

ABSTRACT—By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the important factors in software evolution and shows why this results in a general trend toward software complexity. The paper then introduces laws of program evolution that have been formulated following systematic study of the evolution of a number of different systems. Finally, as an example of the application of these laws, a computer technique is described to assist in program design planning.

I. BACKGROUND

A. The Nature of the Problem

The total U.S. expenditure on programming in 1977 is estimated to have exceeded $50 billion, and may have been as high as $100 billion. This figure, which represents more than 3 percent of the U.S. GNP for that year, is already an unwelcome figure. It has increased ever since, in real terms, and will continue to do so as the microprocessor finds ever wider applications. Programming effectiveness is clearly a significant component of national economic health. Even small percentage improvements in productivity can make significant financial impact. The potential for saving is large.

Economic considerations, however, are not necessarily the main cause of widespread concern. As computers play an ever larger role in society and the life of the individual, it becomes more and more critical to be able to create and maintain effective, cost-effective, and timely software. For more than two decades, however, the programming fraternity, and through them the computer-user community, has faced serious problems in achieving this [1]. As the application of microprocessors extends ever deeper into the fabric of society the problems will be compounded unless very basic solutions are found and developed.

B. Programming

The early 1950's has been a pioneering time. The new images of computers, the dreams of automatic computation, completely hid the fact that the real aspect of programming, the lack of efficient, a priori and systematic, languages, which an acceptable program was, present uncertainty about the accuracy of final results. More recently, the gradual move into the academic, industrial, and government environments, the availability of tools, and the development of practical languages and associated computer software, have been most significant contributions. The logical step for successful computing is to use an adequate programming tool.

Languages have become a major tool in the hands of the programmer. Like all tools, they sought to reduce the manual effort of the user and at the same time improve the quality of his work. They permitted and encouraged concentration on the intellectual tasks which are the real provision of the human programmer. The search for better languages has, however, consumed more effort than has been put into the actual design of the tools. In fact, the tools, their design, and programming-related tasks, have been neglected in favor of the languages themselves, which have been shown to be a major tool in the construction of programs. However, the tools have not been extensively usable for automatic solving of problems. The larger problem of the software community is to develop tools for solving the larger problem of software engineering.
Lehman's laws of software evolution

[The classifications of software in this paper are] based on a recognition of the fact that, at the very least, any program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse.
Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is embedded in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.

- **P-Programs**
  - Outcome specified, but implementation is a model of the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.

- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.
Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is embedded in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.

- **P-Programs**
  - Outcome specified, but implementation models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.

- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.

The only symmetrical solution to the eight queens puzzle (up to rotation and reflection)
Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is **embedded** in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.

- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.

- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.
Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is embedded in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.

- **P-Programs**
  - Outcome specified, but implementation is a model of the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.

- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.
Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is *embedded* in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.

- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.

- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.
Lehman's laws of software evolution

- **Continuing Change**
  - E-Programs must be continually adapted or they become progressively less satisfactory.

- **Continuing Growth**
  - The functional content of an E-Program must be continually increased to maintain user satisfaction over its lifetime.

- **Increasing Complexity**
  - As E-Programs evolve, their complexity increases unless work is done to maintain or reduce it.

- **Declining Quality**
  - The quality of an E-Program will appear to be declining unless it is rigorously maintained.
Lehman's laws of software evolution

- Continuing Change
  - E-Programs must be continually adapted or they become progressively less satisfactory.

- Continuing Growth
  - the functional content of an E-Program must be continually increased to maintain user satisfaction over its lifetime.

- Increasing Complexity
  - as E-Programs evolve, their complexity increases unless work is done to maintain or reduce it.

- Declining Quality
  - the quality of an E-Program will appear to be declining unless it is rigorously maintained
Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is *embedded* in the problem space (delivery truck scheduling)
  - Change frequently in response to changing environment.

- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.

- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.
Key takeaway 8/8:

Identify stable specifications and generalize their implementations.