From Zero to Iterators

Building and Extending the Iterator Hierarchy in a Modern, Multicore World

Patrick M. Niedzielski
pniedzielski.net

I’ve heard of iterators before.

I’ve used iterators before.

I like iterators.

I love iterators.

I *really* love iterators.

I think iterators are fundamental to computer science.

The goal

- ✓ I’ve heard of iterators before.
- ✓ I’ve used iterators before.
- (✓) I love iterators.
- × I think iterators are fundamental to computer science.

The goal

- ✓ I’ve heard of iterators before.
- ✓ I’ve used iterators before.
- (✓) I love iterators.
- × I think iterators are fundamental to computer science. *(Ask me offline!)*

How we'll get there

Iterators 101: Introduction to Iterators

Iterators 301: Advanced Iterators

How we'll get there

Iterators 101: Introduction to Iterators

Iterators 301: Advanced Iterators

Iterators 101: Introduction to Iterators

Iterators 301: Advanced Iterators

Generic Programming

What this talk is not

- An introduction to the STL.
- A comprehensive guide to Concepts Lite.

A note about questions

Iterators 101

A simple starting point

```
template <typename T>
T* copy(T const*  in, T const*  in_end,
        T*       out, T*       out_end)
{
    while (in != in_end && out != out_end)  *out++ = *in++;
    return out;
}
```

A simple starting point

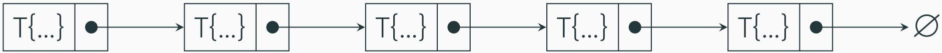
```
auto source      = array<T, 5>{ /* ... */ };
auto destination = array<T, 10>{ /* ... */ };

copy(
    &source[0],      &source[0]      + size(source),
    &destination[0], &destination[0] + size(destination)
);
```

But we don’t just deal with arrays...

```
template <typename T>
struct node
{
    T     value;
    node* next;
};
```

But we don’t just deal with arrays...

```
template <typename T>
struct node
{
    T     value;
    node* next;
};
```



But we don’t just deal with arrays...

```
template <typename T>
node<T>* copy(node<T> const*  in, node<T> const*  in_end,
              node<T>*       out, node<T>*       out_end)
{
    while (in != in_end && out != out_end) {
        out->value = in->value;
        in  = in->next;
        out = out->next;
    }
    return out;
}
```

But we don't just deal with arrays...

```
template <typename T>
T* copy(node<T> const*  in, node<T> const*  in_end,
        T*             out, T*             out_end)
{
    while (in != in_end && out != out_end) {
        *out++ = in->value;
        in = in->next;
    }
    return out;
}
```

But we don’t just deal with arrays...

```
template <typename T>
ostream& copy(T const* in, T const* in_end, ostream& out)
{
    while (in != in_end && out)  out << *in++;
    return out;
}
```

So what?

So what?

- Array $\rightarrow$ Array
- Singly-Linked List $\rightarrow$ Singly-Linked List
- Singly-Linked List $\rightarrow$ Array
- Array $\rightarrow$ Stream

So what?

- Array $\to$ Array
- Singly-Linked List $\to$ Singly-Linked List
- Singly-Linked List $\to$ Array
- Array $\to$ Stream

But also,

- Array $\to$ Singly-Linked List
- Singly-Linked List $\to$ Stream
- Stream $\to$ Array
- Stream $\to$ Singly-Linked List
- Stream $\to$ Stream

$3 \times 3 = 9$ functions

So what?

$$4 \times 4 = 16 \text{ functions}$$

So what?

$$5 \times 5 = 25 \text{ functions}$$

So what?

$$6 \times 6 = 36 \text{ functions}$$

So what?

$$7 \times 7 = 49 \text{ functions}$$

So what?

$7 \times 7 \times 2 = 98$ functions!

So what?

$$7 \times 7 \times 3 = 147 \text{ functions!}$$

So what?

$$7 \times 7 \times 4 = 196 \text{ functions!}$$

m data structures, n algorithms,

$$n \cdot m^2 \text{ functions!}$$

We must be missing something.

Let’s back up.

Let's back up.

```
template <typename T>
output copy(input sequence, output sequence)
{
    while (input not empty && output not empty) {
        write output = read input;
        increment output;
        increment input;
    }
    return output;
}
```

Generalization

- `T const*`
- `node<T> const*`
- `istream&`
- …

Generalization

- `T const*`
- `node<T> const*`
- `istream&`
- …

- `T*`
- `node<T>*`
- `ostream&`
- …

Generalization

- `T const*`
- `node<T> const*`
- `istream&`
- …

- `T*`
- `node<T>*`
- `ostream&`
- …

Iterators

Generalization

```
template <typename T>
output copy(input sequence, output sequence)
{
    while (input not empty && output not empty) {
        write output = read input;
        increment output;
        increment input;
    }
    return output;
}
```

Generalization

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        write output = read input;
        increment output;
        increment input;
    }
    return out;
}
```

Generalization

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        write output = source(in);  // e.g., *in
        increment output;
        increment input;
    }
    return out;
}
```

Generalization

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);     // e.g., *out
        increment output;
        increment input;
    }
    return out;
}
```

Generalization

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

```
inline auto const& source(auto const*       p) { return *p;       }
inline auto const& source(node<auto> const* p) { return p->value; }

inline auto& sink(auto*       p)               { return *p;       }
inline auto& sink(node<auto>* p)               { return p->value; }

inline auto successor(auto const*       p)     { return p + 1;    }
inline auto successor(node<auto> const* p)     { return p->next;  }
```

Let’s get formal.

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Equality Comparison

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Assignment

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Construction

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Destruction

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

successor()

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

source()

What do we need?

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

sink()

What do we need?

`InputIterator`

- Constructible
- Destructible
- Assignable
- Equality Comparable

- `successor()`

- `source()`

`OutputIterator`

- Constructible
- Destructible
- Assignable
- Equality Comparable

- `successor()`

- `sink()`

What do we need?

`InputIterator`

`Regular`

- Constructible
- Destructible
- Assignable
- Equality Comparable

- `successor()`

- `source()`

`OutputIterator`

`Regular`

- Constructible
- Destructible
- Assignable
- Equality Comparable

- `successor()`

- `sink()`

What do we need?

`InputIterator`

`Regular`

- Constructible
- Destructible
- Assignable
- Equality Comparable

`Iterator`

- `successor()`

- `source()`

`OutputIterator`

`Regular`

- Constructible
- Destructible
- Assignable
- Equality Comparable

`Iterator`

- `successor()`

- `sink()`

What do we need?

`InputIterator`

`Regular`

- Constructible
- Destructible
- Assignable
- Equality Comparable

`Iterator`

- `successor()`

`Readable`

- `source()`

`OutputIterator`

`Regular`

- Constructible
- Destructible
- Assignable
- Equality Comparable

`Iterator`

- `successor()`

`Writable`

- `sink()`

Concepts!

Concepts!

```
template <typename T>
concept bool Concept = /* constexpr boolean expression */;
```

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && is_equality_comparable_v<T>;
```

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && is_equality_comparable_v<T>;
```

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && is_equality_comparable_v<T>;
```

Doesn’t exist.

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && ∀x, y ∈ T:
          1. x == y can be used in boolean contexts
          2. == induces an equivalence relation on T
          3. Iff x == y, x and y represent the same value
    ;
```

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && requires(T x, T y) {
          1. x == y can be used in boolean contexts
          2. == induces an equivalence relation on T
          3. Iff x == y, x and y represent the same value
       };
```

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && requires(T x, T y) {
          { x == y } -> bool;
          2. == induces an equivalence relation on T
          3. Iff x == y, x and y represent the same value
       };
```

Concepts!

```
template <typename T>
concept bool Regular =
       is_default_constructible_v<T>
    && is_copy_constructible_v<T>
    && is_destructible_v<T>
    && is_copy_assignable_v<T>
    && requires(T x, T y) {
          { x == y } -> bool;
          // == induces an equivalence relation on T
          // Iff x == y, x and y represent the same value
       };
```

Concepts!

```
template <typename T>
concept bool Readable =
    requires (T x) {
        typename value_type<T>;
        { source(x) } -> value_type<T> const&;  // 𝒪(1)
    };
```

Concepts!

```
template <typename T>
concept bool Writable =
    requires (T x) {
        typename value_type<T>;
        { sink(x) } -> value_type<T>&;  // 𝒪(1)
    };
```

Concepts!

```
template <typename I>
concept bool Iterator =
    Regular<I> &&
    requires (I i) {
        { successor(i) } -> I;  // 𝒪(1)
          // successor() may mutate other iterators.
    };
```

Concepts!

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Concepts!

```
template <typename InputIterator, typename OutputIterator>
    requires Iterator<InputIterator>
          && Readable<InputIterator>
          && Iterator<OutputIterator>
          && Writable<OutputIterator>
OutputIterator copy(InputIterator   in, InputIterator   in_end
                    OutputIterator out, OutputIterator out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Concepts!

```
template <typename In, typename Out>
    requires Iterator<In>
          && Readable<In>
          && Iterator<Out>
          && Writable<Out>
Out copy(In in, In in_end, Out out, Out out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

Concepts!

```
template <Iterator In, Iterator Out>
    requires Readable<In> && Writable<Out>
Out copy(In in, In in_end, Out out, Out out_end)
{
    while (in != in_end && out != out_end) {
        sink(out) = source(in);
        out = successor(out);
        in  = successor(in);
    }
    return out;
}
```

More algorithms

```
template <Iterator It>
    requires Writable<It>
void fill(It it, It it_end, value_type<It> const& x)
{
    while (in != in_end) {
        sink(out) = x;
        it = successor(it);
    }
}
```

More algorithms

```
template <Iterator It, Function<value_type<It>, value_type<It>> Op>
    requires Readable<It>
auto fold(It it, It it_end, value_type<It> acc, Op op)
{
    while (in != in_end) {
        acc = op(acc, source(it));
        it = successor(it);
    }
    return acc;
}
```

More algorithms

```
template <Iterator It>
    requires Readable<It>
It find_first(It it, It it_end, value_type<It> const& value)
{
    while (it != it_end && source(it) != value)
        it = successor(it);
    return it;
}
```

Moving Forward

Moving Forward

```
template <Iterator It>
    requires Readable<It>
It max_element(It it, It it_end)
{
    auto max_it = it;
    while (it != it_end) {
        if (source(it) > source(max_it))  max_it = it;
        it = successor(it);
    }
    return max_it;
}
```

```
template <Iterator It>
    requires Readable<It>
It max_element(It it, It it_end)
{
    auto max_it = it;
    while (it != it_end) {
        if (source(it) > source(max_it))  max_it = it;
        it = successor(it);
    }
    return max_it;
}
```

No!

Moving Forward

```
template <typename I>
concept bool Iterator =
    Regular<I> &&
    requires (I i) {
        { successor(i) } -> I;  // O(1)
          // successor may mutate other iterators
    };
```

Moving Forward

```
template <typename I>
concept bool Iterator =
    Regular<I> &&
    requires (I i) {
        { successor(i) } -> I;  // 𝒪(1)
          // successor may mutate other iterators
    };
```

```
template <typename I>
concept bool ForwardIterator =
    Regular<I> &&
    requires (I i) {
        { successor(i) } -> I;  // O(1)

    };
```

Moving Forward

```
template <typename I>
concept bool ForwardIterator =
    Regular<I> &&
    requires (I i) {
        { successor(i) } -> I;  // O(1)

    };
```

Multi-pass Guarantee

Moving Forward

```
template <ForwardIterator It>
    requires Readable<It>
It max_element(It it, It it_end)
{
    auto max_it = it;
    while (it != it_end) {
        if (source(it) > source(max_it))  max_it = it;
        it = successor(it);
    }
    return max_it;
}
```

Proposition

If a type `T` *models* `ForwardIterator`*, it also models* `Iterator`*.*

Proposition

If a type `T` *models* `ForwardIterator`, *it also models* `Iterator`.

A `ForwardIterator` is just an `Iterator` that doesn't mutate other iterators in `successor()`!

Backing Up

Backing Up

```
template <??? I>
    requires Readable<I> && Writable<I>
I reverse(I it_begin, I it_end)
{
    while (it_begin != it_end) {
        it_end = predecessor(it_end);
        if (it_begin == it_end)  break;

        auto temp       = source(it_end);
        sink(it_end)    = source(it_begin);
        sink(it_begin) = temp;

        it_begin = successor(it_begin);
    }
}
```

Backing Up

```
template <??? I>
    requires Readable<I> && Writable<I>
I reverse(I it_begin, I it_end)
{
    while (it_begin != it_end) {
        it_end = predecessor(it_end);
        if (it_begin == it_end)  break;

        auto temp      = source(it_end);
        sink(it_end)   = source(it_begin);
        sink(it_begin) = temp;

        it_begin = successor(it_begin);
    }
}
```

Backing Up

```
template <typename I>
concept bool BidirectionalIterator =
    Regular<I> &&
    requires (I i) {
        { successor(i)   } -> I;  // 𝒪(1)
        { predecessor(i) } -> I;  // 𝒪(1)
          // i == predecessor(successor(i));
    };
```

Backing Up

```
template <BidirectionalIterator I>
    requires Readable<I> && Writable<I>
I reverse(I it_begin, I it_end)
{
    while (it_begin != it_end) {
        it_end = predecessor(it_end);
        if (it_begin == it_end)  break;

        auto temp       = source(it_end);
        sink(it_end)    = source(it_begin);
        sink(it_begin) = temp;

        it_begin = successor(it_begin);
    }
}
```

Backing Up

```
template <BidirectionalIterator I>
    requires Readable<I> && Writable<I>
I reverse(I it_begin, I it_end)
{
    while (it_begin != it_end) {
        it_end = predecessor(it_end);
        if (it_begin == it_end)  break;

        auto temp       = source(it_end);
        sink(it_end)    = source(it_begin);
        sink(it_begin) = temp;

        it_begin = successor(it_begin);
    }
}
```

Proposition

If a type `T` *models* `BidirectionalIterator`*, it also models* `ForwardIterator`*.*

Proposition

If a type `T` *models* `BidirectionalIterator`, *it also models* `ForwardIterator`.

A `BidirectionalIterator` has a `successor()` function that does not mutate other iterators.

Backing Up

Proposition

If a type `T` *models* `BidirectionalIterator`*, its dual also models* `BidirectionalIterator`.

Backing Up

Proposition

If a type `T` *models* `BidirectionalIterator`*, its dual also models* `BidirectionalIterator`.

Let's define a type whose `successor()` is our old `predecessor()` and whose `predecessor()` is our old `successor()`. This new type also models `BidirectionalIterator`.

Proposition

If a type `T` *models* `BidirectionalIterator`*, its dual also models* `BidirectionalIterator`.

Let's define a type whose `successor()` is our old `predecessor()` and whose `predecessor()` is our old `successor()`. This new type also models `BidirectionalIterator`.

We call the dual of a `BidirectionalIterator` a *reverse iterator*.

Jumping Around

Jumping Around

```
template <ForwardIterator It>
void increment(It& i, size_t n)
{
    // Precondition: n >= 0
    for (auto i = 0; i < n; ++i)  i = successor(i);
}
```

Jumping Around

```
template <ForwardIterator It>
void increment(It& i, size_t n)
{
    // Precondition: n >= 0
    for (auto i = 0; i < n; ++i)  i = successor(i);
}

void increment(auto*& i, size_t n)
{
    // Precondition: n >= 0
    i += n;
}
```

Jumping Around

```
template <BidirectionalIterator It>
void decrement(It& i, size_t n)
{
    // Precondition: n >= 0
    for (auto i = 0; i < n; ++i)  i = predecessor(i);
}

void decrement(auto*& i, size_t n)
{
    // Precondition: n >= 0
    i -= n;
}
```

Jumping Around

```
template <typename I>
concept bool RandomAccessIterator =
       Regular<I>
    && WeaklyOrdered<I>
    && requires (I i, I j, size_t n) {
          { i + n } -> I;       // 𝒪(1)
            // i + 0 == i                           if n == 0
            // i + n == successor(i) + n - 1        if n >  0
            // i + n == predecessor(i) + n + 1      if n <  0
          { i - n } -> I;       // 𝒪(1)
            // i - 0 == i                           if n == 0
            // i - n == predecessor(i) - (n - 1) if n >  0
            // i - n == successor(i) - (n + 1)      if n <  0
          { i - j } -> size_t;  // 𝒪(1)
            // i + (i - j) = i
       };
```

Jumping Around

```
template <RandomAccessIterator It>
    requires Readable<It> && WeaklyOrdered<value_type<It>>
I upper_bound(It it, It it_end, value_type<It> x)
{
    // Base case.
    if (it == it_end)  return it_end;

    // mid_dist is always less than or equal to end - begin,
    // because of integer division
    auto mid_dist = (it_end - it) / 2;
    auto mid = it + mid_dist;

    // Reduce problem size.
    if (source(mid) <= x)  return upper_bound(mid + 1,  it_end, x);
    else                   return upper_bound(     it, mid + 1, x);
}
```

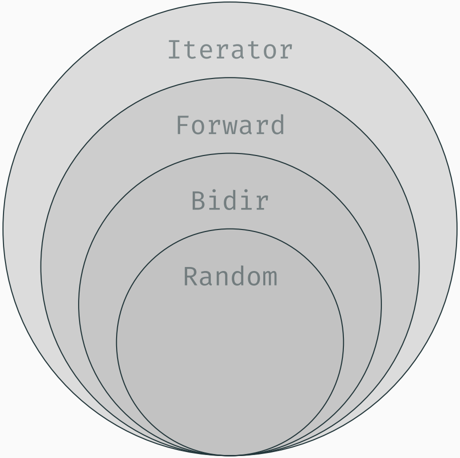
Jumping Around

Proposition
If a type `T` *models* `RandomAccessIterator`*, it also models* `BidirectionalIterator`*.*

Proposition
If a type `T` *models* `RandomAccessIterator`*, it also models* `BidirectionalIterator`*.*

Let's define `successor()` on an object `x` of type `T` to return `x + 1`. Similarly, let's define `predecessor()` to return `x - 1`.

The story so far



Extending the Iterator Hierarchy

Pop Quiz!

Can I `memmove` a `RandomAccess` sequence of `bytes`?

Let’s look at the concept

```
template <typename I>
concept bool RandomAccessIterator =
       Regular<I>
    && WeaklyOrdered<I>
    && requires (I i, I j, size_t n) {
           { i + n } -> I;       // 𝒪(1)
             // i + 0 == i                             if n == 0
             // i + n == successor(i) + n - 1          if n >  0
             // i + n == predecessor(i) + n + 1        if n <  0
           { i - n } -> I;       // 𝒪(1)
             // i - 0 == i                             if n == 0
             // i - n == predecessor(i) - (n - 1) if n >  0
             // i - n == successor(i) - (n + 1)        if n <  0
           { i - j } -> size_t;  // 𝒪(1)
             // i + (i - j) = i
       };
```

No!

A counterexample

```
template <Regular T>
struct segmented_array {
    vector< vector<T> > data;
    // where each inner vector except the last has size segsize
};
```

A counterexample

```
template <Regular T>
struct segmented_array_iterator {
    vector<T>* spine_iter;
    size_t     inner_index;
};

template <Regular T>
segmented_array_iterator<T> operator+(
    segmented_array_iterator<T> it, size_t n)
{
    return segmented_array_iterator<T>{
        it.spine_iter + (it.inner_index + n) / segsize,
        (it.inner_index + n) % segsize
    };
}
```

What does `memmove` need?

What does `memmove` need?

- Trivially copyable data, that is

What does `memmove` need?

- Trivially copyable data, that is
- contiguous in memory.

What does contiguous mean?

What does contiguous mean?

```
auto i = /* ... */;
auto j = /* ... */;
pointer_from(i + n) == pointer_from(i) + n;
pointer_from(i - n) == pointer_from(i) - n;
pointer_from(i - j) == pointer_from(i) - pointer_from(j);
```

What does contiguous mean?

```
auto i = /* ... */;
auto j = /* ... */;
pointer_from(i + n) == pointer_from(i) + n;
pointer_from(i - n) == pointer_from(i) - n;
pointer_from(i - j) == pointer_from(i) - pointer_from(j);
```

There must be an homomorphism `pointer_from` that preserves the range structure.

The simplest homomorphism

```
auto* pointer_from(auto* i) { return i; }
```

The slightly less simple homomorphism

```
template <typename T>
using base_offset_iterator = pair<T*, size_t>;

auto* pointer_from(base_offset_iterator<auto> i)
{
    return i.first + i.second;
}
```

Looks like we have a new concept!

```
template <typename T>
concept bool ContiguousIterator =
       RandomAccessIterator<T>
    && requires (T i) {
           typename value_type<T>;
           { pointer_from(i) } -> value_type<T> const*;
             // pointer_from homomorphism preserves range
             // structure
       };
```

Looks like we have a new concept!

```
template <ContiguousIterator In, ContiguousIterator Out>
    requires Readable<In> && Writable<Out>
          && is_same_v< value_type<In>, value_type<Out> >
          && is_trivially_copyable_v< value_type<In> >
Out copy(In in, In in_end, Out out, Out out_end)
{
    auto count = min( in_end - in, out_end - out );
    memmove(pointer_from(out), pointer_from(in), count);
    return out_end;
}
```

Segmentation Problems

Segmentation Problems

```
template <Regular T>
struct segmented_array {
    vector< vector<T> > data;
    // where each inner vector except the last has size segsize
};
```

Segmentation Problems

```
template <SegmentedIterator In, Iterator Out>
    requires Readable<In> && Writable<Out>
Out copy(In in, In in_end, Out out, Out out_end)
{
    auto seg     = segment(in);
    auto seg_end = segment(in_end);

    if (seg == seg_end)  copy(local(in), local(in_end), out, out_end);
    else {
        out = copy(local(in), end(seg), out, out_end);

        seg = successor(seg);
        while (seg != seg_end) {
            out = copy(begin(seg), end(seg), out, out_end);
            seg = successor(seg);
        }
        return copy(begin(seg), local(in_end), out, out_end);
    }
}
```

Segmentation Problems

```
template <typename T>
concept bool SegmentedIterator =
      Iterator<T>
    && requires (T i) {
          typename local_iterator<T>;
          typename segment_iterator<T>;

          requires Iterator<local_iterator>;
          requires Iterator<segment_iterator>;
          requires Range<segment_iterator>;  // begin(), end()

          { local(i)   } -> local_iterator<T>;
          { segment(i) } -> segment_iterator<T>;
      };
```

```cpp
// Associated types
template <typename T>
using local_iterator< segmented_array_iterator<T> > = T*;
template <typename T>
using segment_iterator< segmented_array_iterator<T> > = vector<T>*;

// Inner iterator range (dirty, to fit on slides!)
auto begin(vector<auto>* vec) { return vec->begin(); }
auto end(vector<auto>* vec)   { return vec->end();   }

// Access functions
auto local(segmented_array_iterator<auto> it) {
    return &it->spine_iter[it->inner_index];
}
auto segment(segmented_array_iterator<auto> it) {
    return it->spine_iter;
}
```

`SegmentedIterators` are great for parallelization.

`SegmentedIterators` are great for parallelization.

```
template <RandomAccessIterator In, SegmentedIterator Out>
    requires Readable<In> && Writable<Out>
          && RandomAccessIterator<Out>
Out copy(In in, In in_end, Out out, Out out_end)
{
    auto& task_pool = get_global_task_pool();

    auto seg     = segment(out);
    auto seg_end = segment(out_end);

    if (seg == seg_end) {
        // ...
    } else {
        // ...
    }
}
```

`SegmentedIterators` are great for parallelization.

```
if (seg == seg_end) {
    return copy(in, in_end, local(out), local(out_end));
} else {
    // ...
}
```

SegmentedIterators are great for parallelization.

```
    } else {
        task_pool.add_task(copy, in, in_end, local(out), end(seg_end));

        seg = successor(seg);
        in = in + min(in_end - in, end(seg_end) - local(out));
        while (seg != seg_end) {
            task_pool.add_task(copy, in, in_end, begin(seg), end(seg));

            seg = successor(seg);
            in = in + min(in_end - in, end(seg) - begin(seg));
        }

        task_pool.add_task(copy, in, in_end, begin(seg), local(out_end));
        return out + min(in_end - in, local(out_end) - begin(out));
    }
```

But wait!

`SegmentedIterators` are great for parallelization.

`ContiguousIterators` are great for bit blasting.

What if we combine them?

Cache-awareness

```
template <typename T>
concept bool CacheAwareIterator =
       SegmentedIterator<T>
    && ContiguousIterator<T>;
```

Cache-awareness

Since,

- each thread is fed its own set of cache lines,

Cache-awareness

Since,

- each thread is fed its own set of cache lines,
- no two threads will be writing to the same cache lines,

Cache-awareness

Since,

- each thread is fed its own set of cache lines,
- no two threads will be writing to the same cache lines,
- no *false-sharing*.

References

- Austern, Matthew H (1998). *Segmented Iterators and Hierarchical Algorithms*. In Proceedings of *Generic Programming: International Seminar, Dagstuhl Castle, Germany*.
- Liber, Nevin (2014). *N3884 — Contiguous Iterators: A Refinement of Random Access Iterators*.
- Stepanov, Alexander and Paul McJones (2009). *Elements of Programming*.

Patrick M. Niedzielski

pniedzielski.net