Part 6

Tyler Swann

Agenda

- Iterators

- Data Structures

- Algorithms

- Ranges

- Views

- Discussion

Iterators

Iterators

- Iterators are abstractions that represent an element or item that belongs to a range or container.

- Iterators are traversal objects, as in they are used to traverse between data that has a common owner.

- Iterators are a lot like pointers in most cases as they are used to hold or refer to some element somewhere else and can read and write to the stored value.

Iterator Categories

Iterator Category	Valid Operations						
	write	read	increment	multiple passes	decrement	random access	contiguous storage
Output	☑		☑				
Input	❓ (might support writing)	☑	☑				
Forward (Satisfies Input)		☑	☑	☑			
Bidirectional (Satisfies Forward)		☑	☑	☑	☑		
Random Access (Satisfies Bidirectional)		☑	☑	☑	☑	☑	
Contiguous (Satisfies Random Access)		☑	☑	☑	☑	☑	☑

Obtaining Iterators

- Iterators are generally defined for a *container-like* object.

- Iterators to the beginning and end of the container can be obtained using the std::begin and std::end respectively.

- The "end" iterator usually holds the element that is one-past-the-end in memory.

- There are customizations of the iterator obtaining functions that can get constant (immutable underlying element), reverse and constant reverse iterators for a [some] container[s].

```cpp
#include <array>
#include <iostream>
#include <iterator>

auto main() -> int
{
    auto a = std::to_array<int>({1, 3, 4, 565, 868, 5, 46});

    std::cout << std::begin(a) << std::endl;
    std::cout << *std::begin(a) << std::endl;
    std::cout << std::end(a) << std::endl;
    std::cout << *(std::end(a) - 1) << std::endl;

    return 0;
}
```

Iterator Operators

- Most iterators; depending on there iterator category, support the same set of operators used by pointers to dereference, increment, decrement etc.

Operation			
dereference	`*i`	`*i = v`	`v = *i`
increment	`i++`	`++i`	
decrement	`i--`	`--i`	
difference	`i - j`		
advance	`i + n`	`i - n`	
index	`i[n]`		

```cpp
#include <array>
#include <iostream>
#include <iterator>

auto main() -> int
{
    auto a  = std::to_array<int>({1, 3, 4, 565, 868, 5, 46});
    auto it = std::begin(a);

    std::cout << *it << std::endl;
    std::cout << *(it++) << std::endl;
    std::cout << *(++it) << std::endl;
    std::cout << *(it--) << std::endl;
    std::cout << *(--it) << std::endl;
    std::cout << *(it + 4) << std::endl;
    std::cout << *(std::end(a) - 4) << std::endl;
    std::cout << it[6] << std::endl;

    auto v { *it };
    *it = 757657;
    std::cout << v << std::endl;
    std::cout << *it << std::endl;

    return 0;
}
```

Iterator Functions

- There are also standard interfaces that allow for the manipulation of iterators.

- These are able to find the correct set of operations for the general functionality (say moving to the *nth* next element) for a given iterator depending on its category.

```cpp
#include <array>
#include <iostream>
#include <iterator>

auto main() -> int
{
    auto a  = std::to_array<int>({1, 3, 4, 565, 868, 5, 46});
    auto it = std::begin(a);

    std::cout << *it << std::endl;
    std::cout << *std::next(it) << std::endl;
    std::cout << *std::prev(it) << std::endl;
    std::cout << *std::next(it, 4) << std::endl;

    auto end = std::end(a);

    std::cout << *std::next(end, -4) << std::endl;

    std::cout << std::distance(it, end - 3) << std::endl;

    std::advance(it, 3);
    std::cout << *it << std::endl;

    return 0;
}
```

Sentinels

- Iterators have no internal notion of the end of a sequence they traverse through, much like pointers.

- Sentinels are a marker that indicate the end of a sequence. A common sentinel that is used by any language that does string processing is the literal character `\0` which denotes the end of string.

- In C++, the 'end' iterator is used as a sentinel, indicating there are no more values that can be yielded by an iterator.

- Any iterator can be used as a sentinel for a sequence of values if it doesn't mark the true end of the sequence.

Data Structures

Data Structures

- Data structures are a fundamental concept in all of Computer Science.

- They are used to arrange and organize data into different shapes, memory layouts and access patterns.

- Different data structures offer different complexities for reading, writing, insertion and erasure.

- There are three main categories of data structures in C++; sequence, associative and unordered associative.

C++ Standard Containers

| Data Structure	Description	Search	Insertion	Erasure	Extraction	Random Access
std::vector<T>	Dynamic contiguous array with fast pushing and popping to the back of the array.	$O(n)$	$O(n)$	$O(n)$	N/A	Index - $O(1)$
std::deque<T>	Double-Ended queue with fast pushing and popping to the front and back of the container.	$O(n)$	$O(1)$	$O(1)$	N/A	Index - $O(1)$
std::forward_list<T>	Singly-Linked List with fast random insertion and erasure.	$O(n)$	Unknown – "Fast"	Unknown – "Fast"	N/A	N/A
std::list<T>	Doubly-Linked List with fast pushing and popping to the front and back of the container.	$O(n)$	$O(1)$	$O(1)$	N/A	N/A
std::set<K>	Unique keys sorted by key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	N/A
std::map<K, V>	Unique key-value pairs sorted by key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	Key - $O(\log n)$
std::multiset<K>	Keys sorted by keys	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	N/A
std::multimap<K, V>	Key-value pairs sorted by keys	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	N/A
std::unordered_set<K>	Unique keys hashed by keys	$O(1)$	$O(1)$	$O(1)$	$O(1)$	N/A
std::unorderd_map<K, V>	Unique keys-value pairs hashed by keys	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Key - $O(1)$
std::unordered_multiset<K>	Keys hashed by keys	$O(1)$	$O(1)$	$O(1)$	$O(1)$	N/A
std::unordered_multimap<K, V>	Keys-value pairs hashed by keys	$O(1)$	$O(1)$	$O(1)$	$O(1)$	N/A

Arrays vs Linked-Lists

- One big debate in Computer Science is; are Linked-Lists actually useful. The answer is, it depends. The biggest (and most naïve) argument made for linked-lists is that are great for inserting and erasing elements in the middle of the structure because you only need to allocate the nodes and correctly link the node for lists, but vectors need to move n elements back in the array possibly causing reallocation.

- The issue with this argument is that this doesn't really apply to modern hardware. Due to arrays and vectors being contiguous in memory it makes it very easy for the CPU to operate and manipulate vectors. Even if insertion or erasure causes reallocation or copying of elements from a vector to a new memory location, because the CPU is often able to vectorize the operation causing the entire copy to occur in a single SIMD instruction. The reason it can do this is CPU caching. CPU's will try to prefetch data it might need for future operations. Rather than getting a few elements it will page a whole chunk of memory into the cache. This means that vectors will often have a large amount or even all their data in the cache all at once meaning it save cycles on the most expensive operations, IO (fetching).

- This means that allocation is often trivial for modern CPU's to efficiently reallocate, insert or erase and copy elements from a vector. Linked lists can never benefit from this kind of optimization due entirely to the fundamental nature of a list. Because the list's elements are broken into nodes that are spread randomly throughout memory, the CPU has no way of finding most or all of the list in one fetch without reading through the list (which is $O(n)$) to get each the next element. Because each element in a list must be fetched this maximizes the amount of cache misses that will occur. This is because if the next node does happen to randomly be in the cache, it must free some space in the cache, go through some pointer indirections and copy the node into the cache. It must repeat this for every node essentially performing n fetch cycles.

- Linked this do have their uses though. You need to store very large elements an they need to be stored for a long time where the fetching of only a few nodes is needed at a time and you need to maintain a linked relationship between data, linked lists are a great choice. The Linux kernel uses a generic linked list structure to connect different data across the kernel. Linked lists are also used in memory paging applications at the OS level.

Vector vs. List — sequence test
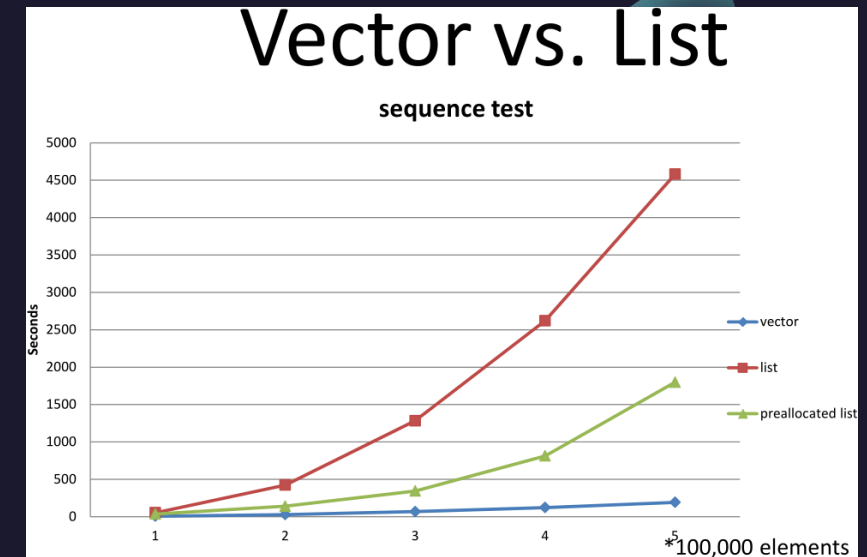
OO Arrays vs C++ Vector

- One reason people think linked lists perform better then dynamic arrays is that most languages with dynamic array like structures use an OO design. C++ does not.

- C++ focuses on zero-cost abstraction. That is, you don't pay for what you don't use and there is often lots you don't need.

- With all of the indirections used in indirections and sub-objects used OO style arrays, a linked list doesn't look so bad

`std::vector<int>`

padding	1	2	3	4	5	6	unused	unused

`std::list<int>`

OO Style
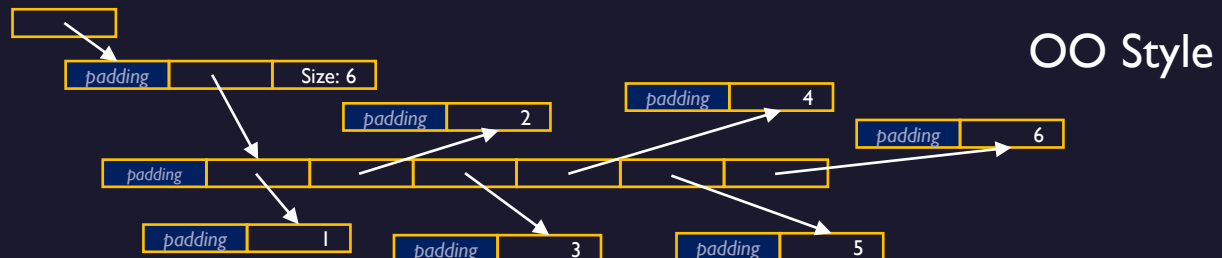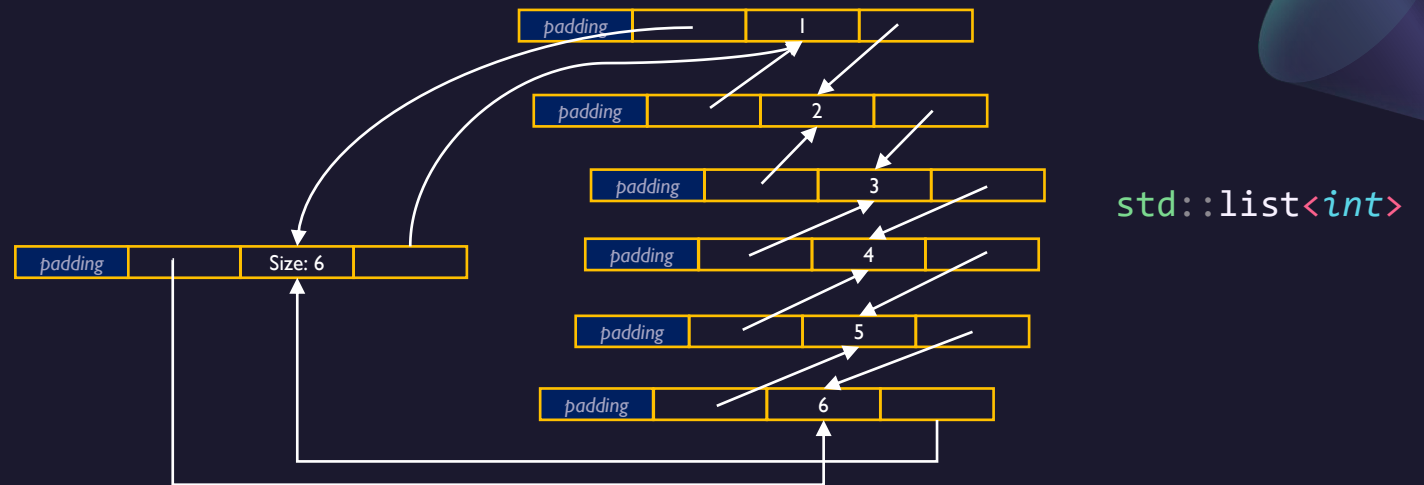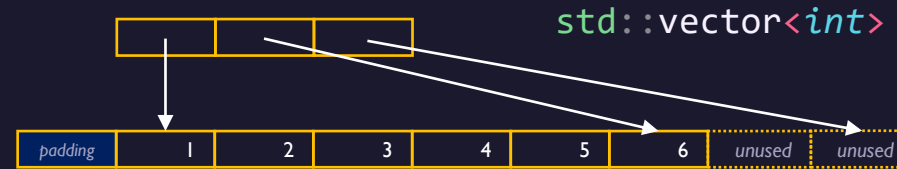
Bitset

- Static range of individually addressable bits.

- Can be used in *constexpr* contexts.

- Overloads for bitwise operations.

- Conversion functions to string representations of the bits.

- Bit testing methods.

```cpp
#include <bitset>
#include <iostream>
#include <string>


auto main() -> int
{
    auto b = std::bitset<6>(0b011010uLL);

    println(b);
    b[2] = true;
    println(b);
    b.set(4) = false;

    b.flip(0);
    println(b);
    b.flip();
    println(b);
    b.reset();
    println(b);

    std::cout << std::boolalpha;
    std::cout << (b.test(5) == false) << std::endl;
    std::cout << b.any() << std::endl;
    std::cout << b.all() << std::endl;
    std::cout << b.none() << std::endl;
    std::cout << std::noboolalpha;

    return 0;
}
```

Any

- Storage of dynamic type, copyable object.

- Access to value through casting with std::any_cast<T>.

- In-place construction and destruction of held object.

```cpp
#include <iostream>
#include <any>

auto main() -> int
{
    auto a = std::make_any<int>(6);

    println<int>(a);
    a.emplace<double>(6.797898);
    println<double>(a);

    std::cout << a.type().name() << std::endl;

    std::cout << std::boolalpha;
    std::cout << a.has_value() << std::endl;
    a.reset();
    std::cout << a.has_value() << std::endl;
    std::cout << std::noboolalpha;

    return 0;
}
```

Algorithms

Algorithms

- Algorithms are functions that perform a specific set of steps often to perform some computation.

- Algorithms are mostly used to manipulate data. When paired with data structures, algorithms can make development far more seamless, especially when written in a generic style.

- C++ Standard Algorithms were the brainchild of Alex Stepanov that used templates and generic techniques to create powerful interfaces and seamless interoperability between algorithms and data structures.

- The focus of this section is not on how to write algorithms but rather how to use them. How to compose them together using functional techniques create expressible and type safe code that is also fast.

- C++ algorithms take a begin and end iterator indicating the range of elements the algorithm applies to.

C++ Standard Algorithms

Category	C++ Algorithm	Common Name	Description
General	`std::for_each`	foreach	Applies a function (usually with side effects) to every element in a range.
Sorting	`std::sort`	sort	Sorts elements in-place according to a predicate (`<`).
	`std::partition`	partition	Partially sort range so that every element for which predicate is true proceeds every element for which a predicate is false.
	`std::nth_element`	nth	Partially sort range so that the nth element is in its sorted position.
Comparisons	`std::equal`	equal	Compares two ranges and returns false if the predicate (`==`) fails.
	`std::lexicographical_compare`	lexicographical compare	Compares two ranges lexicographically using a predicate (`<`).
	`std::all_of`	all	Checks if all elements in a range satisfy a predicate.
	`std::any_of`	any	Checks if any elements in a range satisfy a predicate.
	`std::none_of`	none	Checks if none elements in a range satisfy a predicate.
	`std::mismatch`	mismatch	Finds the first pair of elements between two ranges that fail a predicate (`==`).

C++ Standard Algorithms cont.

Category	C++ Algorithm	Common Name	Description
Searching	`std::find`	find	Finds the first element equal to desired value.
	`std::search`	search	Finds first occurrence of a subrange of desired values.
	`std::adjacent_find`	adjacent find	Finds first occurrence of adjacent elements satisfying a predicate (`==`).
	`std::binary_search`	binary search	Uses binary search on partially sorted range to check if a value exists.
	`std::equal_range`	equal range	Finds the subrange of elements in a partially sorted range that are equal to the desired value.
	`std::lower_bound`	lower bound	Finds the subrange of elements in a partially sorted range that are less than the desired value.
	`std::upper_bound`	upper bound	Finds the subrange of elements in a partially sorted range that are greater than the desired value.
Generators	`std::fill`	repeat	Fills a range with a particular value.
	`std::iota`	iota	Fills a range with incrementing values from a starting value.
	`std::generate`	generate	Fills range with the result of a function.

C++ Standard Algorithms cont.

Category	C++ Algorithm	Common Name	Description
Modifying	`std::copy`	copy	Copies elements from one range to another.
	`std::move`	move	Moves elements from one range to another.
	`std::swap_range`	swap	Swaps elements between two ranges.
	`std::remove`	filter	Logically removes elements (moves to the back of range) equal to the given value.
	`std::replace`	replace	Replaces elements equal to the given value with a new value.
	`std::reverse`	reverse	Reverses a range in-place.
	`std::transform`	map	Applies a function to every element in a range or to two ranges, writing to a new range.
	`std::rotate`	left rotate	Rearranges a range such that the desired element is the new beginning and the element previously just before is the new end of the range.
	`std::sample`	sample	Uses a pseudo-random generator to sample random elements from a range.
	`std::shuffle`	shuffle	Uses a pseudo-random generator to reorganise a ranges elements.

C++ Standard Algorithms cont.

Category	C++ Algorithm	Common Name	Description
Numeric	std::min_element	minimum	Returns the smallest element in a range.
	std::max_element	maximum	Returns the largest element in a range.
	std::count	count	Counts number if elements satisfying an unary predicate ($e == v$).
	std::clamp	clamp	Clamps a scalar between particular bounds
	std::accumulate	left fold	Performs a left-fold (reduction) on a range applying a binary function (+) between every element and an accumulator.
	std::inner_product	map-reduce	Performs a binary map (*) on two ranges and then performs a left fold with a different binary function (+) between every mapped element.
	std::partial_sum	left scan (left scan 1)	Applies a binary operator (+) between every element in a range and an accumulator, saving intermediate values in a new range.
	std::adjacent_difference	adjacent difference	Applies a binary function (-) between adjacent elements in a range.

Reductions

- Reductions are powerful *higher-order* functions that recursively combine elements in a range to build up a return value, usually called folds.

- There are two main fold algorithms that exist in computer science called a left-fold and a right-fold.

- Reductions also include operations such as scans which store the intermediate accumulation of a range.

- Reductions along with function application are some of the most fundamental operations in all of computing

$$foldl\ f\ z\ [\,] = z$$
$$foldl\ f\ z\ (x:xs) = foldl\ f\ (f\ z\ x)\ xs$$
$$f(z, f\left(x_0, f(x_1, \ldots f(x_{n-1}, x_n))\right))$$

$$foldr\ f\ z\ [\,] = z$$
$$foldr\ f\ z\ (x:xs) = f\ x\ (foldr\ z\ xs)$$
$$f\left(x_0, f(x_1, \ldots f(x_{n-1}, f(x_n, z)))\right)$$

$$scanl\ f\ z\ [\,] = [z]$$
$$scanl\ f\ z\ (x:xs) = z : scanl\ f\ (f\ z\ x)\ xs$$
$$[z, f(z, x_0), f(f(z, x_0), x_1), \ldots f(\ldots, x_{n-1}), f(\ldots, x_n)]$$

Ranges

Ranges

- C++ has used the notion of a range for over two decades now however, ranges were almost exclusively made up from iterator pairs representing the endpoints of a range.

- As you may suspect, using at least two iterators for every algorithm can feel tedious. There are also ranges that use different sentinels such as a count and unique to markers.

- In C++20, the Ranges Library was added which featured range forms of many of the standard library's algorithms. This formalized a range to be an *iterator-sentinel* pair. This formalization allowed for possible ranges to be more permissible while; with the addition of concepts, put stricter and safer constraints on C++ algorithms. The biggest nicety is that the input range can now be specified without needing to declare its iterators.

- Ranges also accept projections that can be used to extract specific data from

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

auto main() -> int
{
    auto v = std::vector<int>{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    println(v);

    std::ranges::transform(
        v, v.begin(),
        [](const auto& x){ return x * x; }
    );

    println(v);

    return 0;
}
```

Range Categories

Concept	std::forward_list	std::list	std::deque	std::array	std::vector
std::ranges::input_range	☑	☑	☑	☑	☑
std::ranges::forward_range	☑	☑	☑	☑	☑
std::ranges::bidirectional_range		☑	☑	☑	☑
std::ranges::random_access_range			☑	☑	☑
std::ranges::contiguous_range				☑	☑

Views

Views

WHAT IS A VIEW?

- A view is an abstraction or extension of a range. Views do not own any data but refer to it.

- Views formally model a *range-like* object and are cheap to copy, move, assign and destroy,

- Views often manipulate the elements they refer to in a lazy manner meaning only when element's values are need.

LAZY EVALUATION

- Lazy evaluation is the process of delaying computation until the value is needed.

- This allow for more efficient programming as well as working with infinite ranges.

- C++ views use lazy evaluation to delay the value of an element until it is needed.

COMPOSITION

- Composition is the processes of combining functions.

- This allows you to create a new function that is the applies two functions, one after the other to a single input.

- Views are composed to build up new expressions that manipulate ranges.

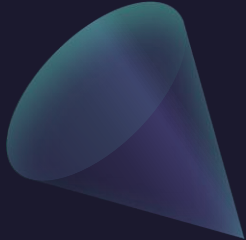- Composition uses left-to-right piping (|) syntax to compose views.

Factories

- A factory is a view constructor. It generates a view from a given argument.

- There are four standard factories in C++20 that will generate a view.

- `std::views::empty` creates an empty view.

- `std::views::single` creates a view of a single element.

- `std::views::iota` generates a view by incrementing an initial value. Can generate an infinite view if no upper bound is specified.

- `std::views::istream` will generate a view from an `std::istream<T>` or another input stream type.

```cpp
#include <algorithm>
#include <iostream>
#include <ranges>

auto main() -> int
{
    /// Prints "0 1 2 3 4 5 6 7 8 9 10"
    std::ranges::copy(
        std::views::iota(0, 11),
        std::ostream_iterator<int>(std::cout, " ")
    );

    return 0;
}
```

Range Adaptor Views

Range Adaptor	Common Name	Description
std::views::transform	map	Creates a view that maps an unary function onto a range
std::views::filter	filter	Creates a view that filters elements that fail an unary predicate from a range
std::views:reverse	reverse	Creates a view that reverses a range
std::views::take	take	Creates a view that only takes N elements from a range
std::views::take_while	take while	Creates a view that takes elements from a range until an unary predicate fails
std::views::drop	drop	Creates a view that skips N elements from a range
std::views::drop_while	drop while	Creates a view that skips elements from a range until an unary predicate fails
std::views::join	concat	Creates a view of joined subranges of a range
std::views::split	split	Creates a view that splits a range into subranges based on a delimitator
std::views::lazy_split	split	Creates a view that lazily splits a range into subranges based on a delimitator
std::views::common	N/A	Creates a view whose *iterator-sentinel* pair are of the same type
std::views::elements	N/A	Creates a view consisting of the Nth *Tuple-Element* from a range of *Tuple-Like* elements
std::views::keys	map-first	Creates a view consisting of the $0th$ *Tuple-Element* from a range of *Tuple-Like* elements
std::views::values	map-second	Creates a view consisting of the $1st$ *Tuple-Element* from a range of *Tuple-Like* elements

Discussion

- Any questions?

- Need help?

- Open discussion.

- Concerns?

Next Week

Parallel Algorithms

Atomics

Threads

Mutexes & Locks

Async/Await

Thank You

Tyler Swann

https://github.com/MonashDeepNeuron/HPP