2. STL Algorithms

Klaus Iglberger
March 21st, 2024

Content

2.1. Terminology

Terminology

Standard Library:

The *Standard Library* is the official collection of classes and functions described in and provided with the C++ standard. In parts, the STL is a subset of the Standard Library.

Alexander Stepanov

Andrew Koenig

Standard Template Library (STL):

The *STL* is a template-based C++ library developed in the 80s and 90s by Dave Musser, Alexander Stepanov and Meng Lee. Many concepts, ideas, classes, etc., were introduced into the C++ standard library.

2.2. Overview of the STL

The STL in a Nutshell

The STL consists of the following **six concepts**:

- **Containers**: Implementations of the common data collections
- **Algorithms**: work on the data contained in containers
- **Iterators**: The glue between containers and algorithms
- **Function Objects**: Provide flexibility and customizability
- **Adapters**: Adapting the basic containers to special purposes
- **Allocators**: Generalization and customization of memory allocation

The STL in a Nutshell

The Expert’s View on the STL

”Generic programming depends on the decomposition of programs in components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces.”

(Alexander Stepanov, Fundamentals of Generic Programming)

2.3. Motivation

The Expert’s View on the STL

”There was never any question that the [standard template] library represented a breakthrough in efficient and extensible design.”

(Scott Meyers, Effective STL)

The Expert’s Advice

”If you want to improve code quality in your organization, I would say, take all your coding guidelines and replace them with the one goal. That’s how important I think this one goal is: No Raw Loops. This will make the biggest change in code quality within your organization.”

(Sean Parent, C++ Seasoning, Going Native 2013)

2.4. STL Iterators

Iterators: Glue Between Containers and Algorithms

- The STL mechanism to decouple algorithms from containers
- Algorithms are parameterized by iterator types
- Pointers are iterators
- Containers provide iterators over their elements (begin and end)
- Iterator concepts form a hierarchy (no inheritance, but refinement)

Iterators: Glue Between Containers and Algorithms

All algorithms expect at least a pair of iterators specifying the range to work on:

```
[begin; end)
```

begin specifies the first element of the range.

end specifies the element after the last element of the range.

```
std::reverse( vec.begin(), vec.end() );
std::copy( vec.begin(), vec.end(), deque.begin() );
```

What are the advantages of this half-open interval concept? Discuss.

Iterator Guidelines

Guideline: Prefer using iterators with `[begin,end)` semantics.

Guideline: Remember that pointers, references, and iterators into a container with contiguous storage are invalidated when elements are added to this container.

Iterator Guidelines

Guideline: Prefer prefix increment and decrement to postfix increment and decrement for all iterator types.

```
std::vector<int> vec;
// ... Initialization
for(std::vector<int>::iterator it=vec.begin(); it!=vec.end(); it++)
{ /* ... */ }
                                                             ++it
```

Iterator Guidelines

Guideline: Prefer range-based `for` loops for the standard traversal of elements of a collection.

```
std::vector<int> vec;
// ... Initialization
for(auto& element : vec)
{ /* ... */ }
```

2.5. STL Algorithms

STL Algorithms

- Free functions, not member functions
- Operate on half open ranges
- Algorithms are decoupled from containers
- Provide an intuitive naming and parameter convention

```
namespace std {

template< class RandomIt >
void sort( RandomIt first, RandomIt last );

template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );

template< class InputIt, class UnaryPredicate >
InputIt find_if( InputIt first, InputIt last, UnaryPredicate p );

} // namespace std
```

STL Algorithms

cppreference.com Create account Search

Page Discussion View Edit History

C++ **Algorithm library** Constrained algorithms

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past* the last element to inspect or modify.

Non-modifying sequence operations

Defined in header `<algorithm>`

all_of (C++11) **any_of** (C++11) **none_of** (C++11)	checks if a predicate is `true` for all, any or none of the elements in a range (function template)
ranges::all_of (C++20) **ranges::any_of** (C++20) **ranges::none_of** (C++20)	checks if a predicate is `true` for all, any or none of the elements in a range (niebloid)
for_each	applies a function to a range of elements (function template)
ranges::for_each (C++20)	applies a function to a range of elements (niebloid)
for_each_n (C++17)	applies a function object to the first n elements of a sequence (function template)
ranges::for_each_n (C++20)	applies a function object to the first n elements of a sequence (niebloid)
count	returns the number of elements satisfying specific criteria

Examples

- Copy from a vector to a deque

```
std::copy( vec.begin(), vec.end(), deq.begin() );
```

- Sort the elements in a vector

```
std::sort( vec.begin(), vec.end() );
```

- Reverse the order of elements

```
std::reverse( vec.begin(), vec.end() );
```

- Find the value 5 in a list

```
std::find( lst.begin(), lst.end(), 5 );
```

Examples

- Copy from a vector of integers to std::cout

```
std::copy( vec.begin(), vec.end()
         , std::ostream_iterator<int>( std::cout, "\n" ) );
```

- Removing all duplicates from a range

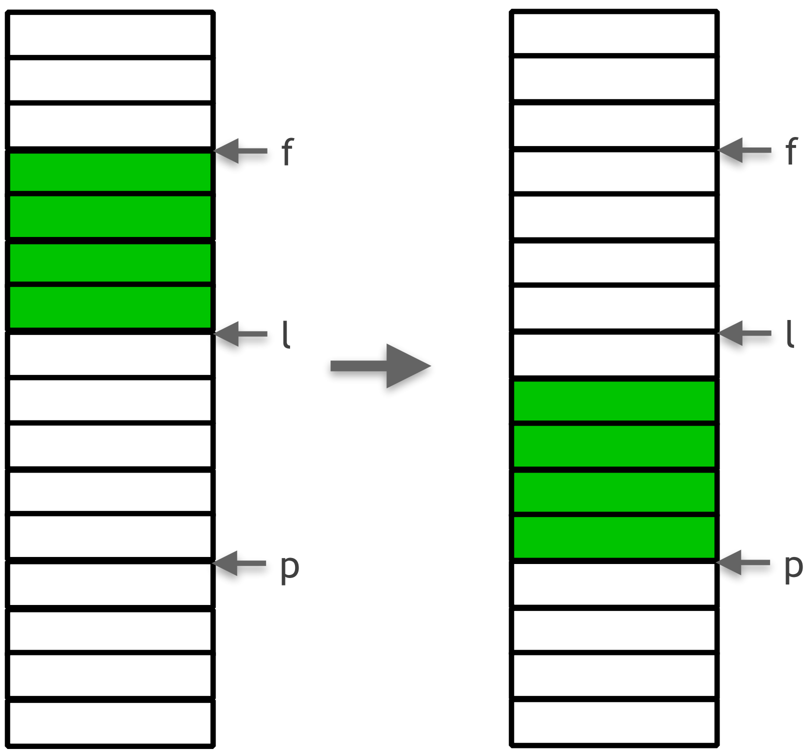
```
std::sort( vec.begin(), vec.end() );
vec.erase( std::unique( vec.begin(), vec.end() ), vec.end() );
```

- Find the first odd integer in a list

```
struct IsOdd {
   bool operator()( int i ) const { return i & 0x1; }
};

std::find_if( lst.begin(), lst.end(), IsOdd{} );
```

Examples

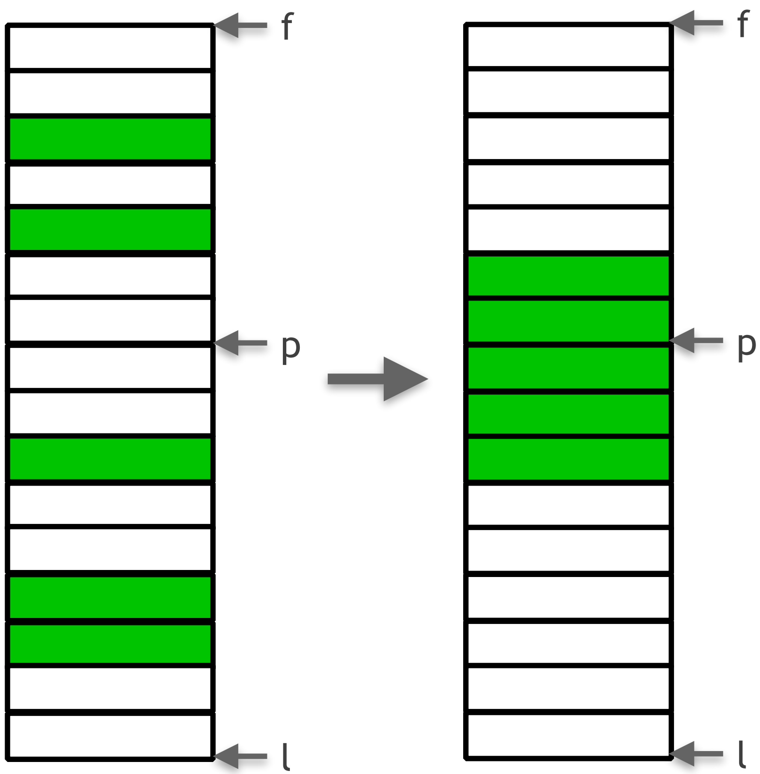
- Move a number of consecutive elements in a vector




```
std::rotate( f, l, p );
```

Examples

- Gather an arbitrary number of element at a specific position




```
using namespace std;

stable_partition(f, p, not1(s));
stable_partition(p, l, s);
```

Programming Task

Task (3_Concepts_and_STL/Algorithms/STLintro): Solve the following tasks on a vector of integers by means of STL algorithms:

- Print the contents of the vector to the screen
- Reverse the order of elements in the vector
- Find the first element with the value 5
- Count the elements with the value 5
- Replace all 5s by 2s
- Sort the vector
- Determine the range of 2s

Hint: Use either of the following two web pages as reference.

www.cppreference.com

www.cplusplus.com

Programming Task

Task (3_Concepts_and_STL/Algorithms/STLpro): Solve the following tasks on a vector of integers by means of STL algorithms:

- Compute the product of all elements in the vector
- Extract all numbers <= 5 from the vector
- Compute the (numerical) length of the vector
- Compute the ratios v[i+1]/v[i] for all elements v[i] in v
- Move the range [v[3], v[5]] to the beginning of the vector

Hint: Use either of the following two web pages as reference.

www.cppreference.com

www.cplusplus.com

Programming Task

Task (3_Concepts_and_STL/Algorithms/Simpson): Implement the empty functions to perform the following operations on the Simpson characters:

- Print all persons to the screen
- Randomize their order
- Find the youngest person
- Order them by first name
- Order them by last name without affecting the order of first names
- Order them by age without affecting the order of first and last names
- Count the number of children
- Put all Simpsons first without affecting the general order of persons
- Compute the total age of all persons
- Put the last person first, moving all others by one position
- Determine the third oldest person as quickly as possible

Programming Task

Task (3_Concepts_and_STL/Algorithms/SimpsonPro): Implement the empty functions to perform the following operations on the Simpson characters:

- Print all persons to the screen
- Randomize their order
- Find the youngest person
- Order them by last name without affecting the order of first names
- Highlight the last name of all persons with the given name
- Put all children first
- Compute the total length of all last names
- Check if two adjacent persons have the same age
- Compute the maximum age difference between two adjacent persons
- Determine the median age of all persons
- After ordering all persons by last name, find all the Simpsons
- Print a string containing the first names of all children

Programming Task

Task (3_Concepts_and_STL/Algorithms/Accumulate):

Step 1: Implement the `accumulate()` algorithm. The algorithm should take a pair of iterators, an initial value for the reduction operation, and a binary operation that performs the elementwise reduction.

Step 2: Implement an overload of the `accumulate()` algorithm that uses `std::plus` as the default binary operation.

Step 3: Implement an overload of the `accumulate()` algorithm that uses the default of the underlying data type as initial value and `std::plus` as the default binary operation.

Step 4: Test your implementation with a custom binary operation (e.g. `Times`).

Programming Task

Task (3_Concepts_and_STL/Algorithms/Partition): Implement the `partition()` algorithm that separates two groups of elements. The algorithm should take a pair of iterators and a predicate that identifies the elements of the first group.

Programming Task

Task (3_Concepts_and_STL/Algorithms/CartesianProduct): Write the cartesian_product() algorithm, which combines every element of the first range with every element of the second range (see https://en.wikipedia.org/wiki/Cartesian_product). By default, the two elements should be combined in a `std::tuple`, but it should be possible to configure the binary operation.

Programming Task

Task (3_Concepts_and_STL/Algorithms/SortSubrange): Implement the `sort_subrange()` algorithm in the following example. The algorithm should take four iterators, which specify the total range of elements and the subrange to be sorted.

Programming Task

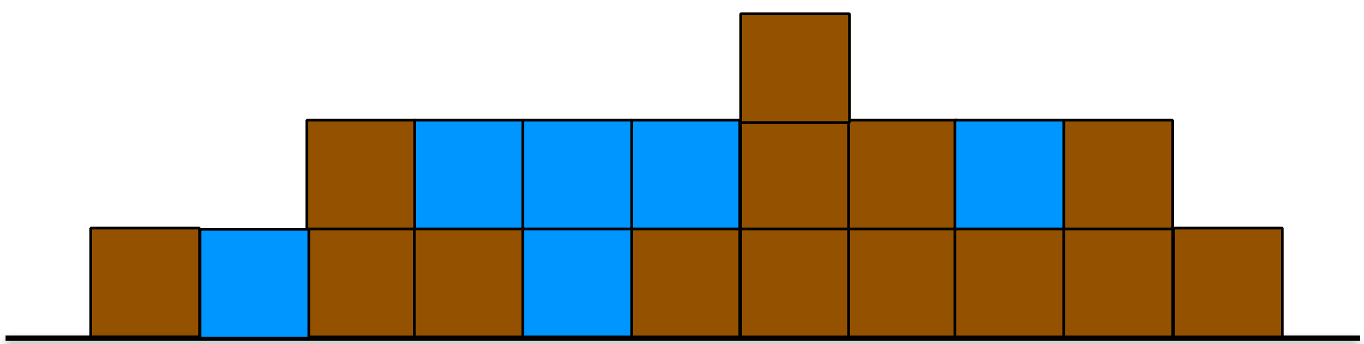
Task (3_Concepts_and_STL/Algorithms/ExtractStrings): Implement the `extract_strings()` algorithm. The algorithm should extract all strings from a long string of space-separated words.

Programming Task

Task (3_Concepts_and_STL/Algorithms/LongestStreak): Determine the longest streak of consecutive equal values in the given range of elements.

Programming Task

Task (3_Concepts_and_STL/Algorithms/Trap): Implement the following `trap()` algorithm for a given vector of non-negative integers. The given vector represents an elevation map, where the width of each bar is 1. The `trap()` algorithm should compute how much water can be trapped in between the peaks.



```
vector<int> v{ 1,  0,  2,  1,  0,  1,  3,  2,  1,  2,  1 };
```

Programming Task

Task (3_Concepts_and_STL/Algorithms/IsEmailAddress): Implement the `is_email_address()` algorithm, which determines whether the given email address is valid. An email address should be considered valid, if the following properties hold:

- the address must contain exactly one '@' symbol
- both the part before and the part after the '@' symbol …
  - … must contain only alphanumeric characters or dots
  - … must not start or end with a dot
  - … must not contain consecutive dots (e.g. "..")
- the part after the '@' symbol must contain at least one dot

Programming Task

Task (3_Concepts_and_STL/Algorithms/IsPalindrome):

Step 1: Implement the `is_palindrome()` algorithm in the following example. The algorithm should detect if the given range is the same when traversed forward and backward. The algorithm should return `true` only for true palindromes, and `false` for empty ranges and non-palindromes.

Step 2: Restrict the algorithm to bidirectional iterators by means of C++20 concepts.

Programming Task

Task (3_Concepts_and_STL/Algorithms/Median): Implement the `median()` algorithm that computes the median of a given range of arithmetic values (i.e. integral or floating point values). The algorithm should take a pair of random access iterators or a range and return an optional representing the result.

Programming Task

Task (3_Concepts_and_STL/Algorithms/MajorityVote): Implement the Boyer-Moore majority vote algorithm [1], which determines the majority of a sequence of elements (that is, an element that occurs repeatedly for more than half of the elements of the input). The algorithm should take two iterators or a range and return an optional representing the majority vote, if there is one.

[1] https://en.wikipedia.org/wiki/Boyer-Moore_majority_vote_algorithm

Programming Task

Task (3_Concepts_and_STL/Algorithms/AlgorithmPerformance1): Copy-and-paste the following code into godbolt.org. Compare the generated assembly code for the following three different solutions:

- an iterator-based manual for loop;
- the STL `accumulate()` algorithm;
- an index-based manual for loop.

Programming Task

Task (3_Concepts_and_STL/Algorithms/AlgorithmPerformance2): Copy-and-paste the following code into quick-bench.com. Benchmark the time to sort a `std::vector` of integers.

Programming Task

Task (3_Concepts_and_STL/Algorithms/RangesRefactoring_1):

Step 1: Understand the inner workings of the 'select_birthday_children()' function: what does the function return?

Step 2: Refactor the function from an imperative to a declarative style by means of C++20 ranges.

Step 3: Compare the runtime performance of both versions (imperative and declarative).

Programming Task

Task (3_Concepts_and_STL/Algorithms/RangesRefactoring_2):

Step 1: Understand the code of the `main()` function: what does the final output print?

Step 2: Refactor the `main()` function from an imperative to a declarative style by means of C++20 ranges.

Programming Task

Task (3_Concepts_and_STL/Algorithms/RangesRefactoring_3):

Step 1: Understand the code of the `main()` function: what does the final output print?

Step 2: Refactor the `main()` function from an imperative to a declarative style by means of C++20/23 ranges.

The Definition of Raw Loops

- A raw loop is any loop inside a function where the function serves purpose larger than the algorithm implemented by the loop.
- Range-based for loops for for-each and simple transforms
    - Use **auto const&** for for-each and **auto&** for transforms

```
for( auto const& elem : range ) f(elem);   // for-each
for( auto& elem : range ) e = f(elem);     // simple transform
```

- Keep the body short

```
for( auto const& elem : range ) f(g(elem));
for( auto const& elem : range ) { f(elem); g(elem); }
for( auto& elem : range ) e = f(e) + g(e);
```

The Expert's Interpretation of Raw Loops

"9 times out of 10, a for-loop should either be the only code in a function, or the only code in the loop should be a function (or both)."

(Tony Van Eerd, @tvaneerd via Twitter)

The Expert’s Definition of “Beauty”

”Beauty

The ease with which a language allows the expression of correct code”

(Sean Parent, The Tragedy of C++, Acts One & Two, CppNorth 2022)

The Expert’s Opinion On The Cost of Code

”Each line of code costs a little. The more code you write, the higher the cost. The longer a line of code lives, the higher its cost. Clearly, unnecessary code needs to meet a timely demise before it bankrupts us.”

(Pete Goodliffe, Becoming a Better Programmer)

The Expert’s Opinion On Complexity

”Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are by definition, not smart enough to debug it.”

(Brian Kernighan)

Algorithm Guidelines

Guideline: “No raw loops” (Sean Parent)

Guideline: Keep your code simple (KISS).

Guideline: Use algorithms to reduce duplication (DRY).

Guideline: Know the standard algorithms. They can handle all basic tasks elegantly and efficiently (zero cost abstraction).

Guideline: Use the right algorithm for the right task.

Algorithm Guidelines

Guideline: Consider the design of the STL: It follows SRP, OCP, DRY and builds on the Strategy and Command design patterns.

Core Guideline P.3: Express intent

Core Guideline T.40: Use function objects to pass operations to algorithms

Core Guideline T.141: Use an unnamed lambda if you need a simple function object in one place only

Limitations of STL Algorithms

Limitations of STL Algorithms - Example 1

Task (3_Concepts_and_STL/Algorithms/BadCopy): Explain the error in the following program.

```
std::vector<int> vec;
std::list<int> lst;

// ... Initialization of lst

std::copy( lst.begin(), lst.end(), vec.begin() );
```

Limitations of STL Algorithms - Example 1

Task (3_Concepts_and_STL/Algorithms/BadCopy): Explain the error in the following program.

```
std::vector<int> vec;
std::list<int> lst;

// ... Initialization of lst

std::copy( lst.begin(), lst.end(), vec.begin() );
```

- `copy()` assumes that the target holds enough elements for all elements to be copied
- Reasonable assumption since it is not possible to change the size of a container via the given iterators
- In case the target vector is empty, we enter the realm of undefined behavior

Limitations of STL Algorithms - Example 1

Either resize the `vector` accordingly ...

```
std::vector<int> vec;
std::list<int> lst;

// ... Initialization of lst

vec.resize( lst.size() );
std::copy( lst.begin(), lst.end(), vec.begin() );
```

Limitations of STL Algorithms - Example 1

... or use the following approach:

```
std::vector<int> vec;
std::list<int> lst;

// ... Initialization of lst

vec.reserve( lst.size() );  // Optional
std::copy( lst.begin(), lst.end(), std::back_inserter(vec) );
```

Limitations of STL Algorithms - Example 1

Guideline: Beware that algorithms cannot add new elements to a container.

Limitations of STL Algorithms - Example 2

Task (3_Concepts_and_STL/Algorithms/BadTransform): Explain the error in the following program.

```
int transmogrify( int x );

std::vector<int> values;
// ... Put data into the vector

std::vector<int> results;

// Apply ‘transmogrify’ to each object in values,
// appending the return values to results
std::transform( values.begin(), values.end(),
                results.end(), transmogrify );
```

Limitations of STL Algorithms - Example 2

Task (3_Concepts_and_STL/Algorithms/BadTransform): Explain the error in the following program.

```
int transmogrify( int x );

std::vector<int> values;
// ... Put data into the vector

std::vector<int> results;

// Apply 'transmogrify' to each object in values,
// appending the return values to results
std::transform( values.begin(), values.end(),
                results.end(), transmogrify );
```

Same problem as in the previous task: The target `vector` has not enough elements → undefined behavior.

Limitations of STL Algorithms - Example 2

Task (continued): Ok, now that we have repaired the access violation, there is an easy way to considerably improve performance. Show how this can be achieved.

```
int transmogrify( int x );

std::vector<int> values;
// ... Put data into the vector

std::vector<int> results;

// Apply ‘transmogrify()’ to each object in values,
// appending the return values to results
std::transform( values.begin(), values.end(),
                std::back_inserter(results), transmogrify );
```

Limitations of STL Algorithms - Example 2

If we turn the transmogrify function into a functor, the compiler can take advantage of the inline function definition and inline the function call. This is **not** possible in case of a function pointer.

```
struct Transmogrify {
   inline int operator()( int x ) const { return x * x; }
};

std::vector<int> values;
// ... Put data into the vector

std::vector<int> results;

// Apply 'Transmogrify' to each object in values,
// appending the return values to results
std::transform( values.begin(), values.end(),
                std::back_inserter(results), Transmogrify() );
```

Limitations of STL Algorithms - Example 2

Core Guideline T.40: Use function objects to pass operations to algorithms

Limitations of STL Algorithms - Example 3

Task (3_Concepts_and_STL/Algorithms/BadAccumulate): Explain the error in the following program:

```
std::vector<double> vec;

// ... Adding elements to vec

double const sum =
   std::accumulate( vec.begin(), vec.end(), 0 );
```

Limitations of STL Algorithms - Example 3

Task (3_Concepts_and_STL/Algorithms/BadAccumulate): Explain the error in the following program:

```
std::vector<double> vec;

// ... Adding elements to vec

double const sum =
   std::accumulate( vec.begin(), vec.end(), 0 );
```

- The type of the third parameter defines the type of the accumulator
- adding `double` values to an `int` strips away the floating point part
- the final result is wrong!

Limitations of STL Algorithms - Example 3

Make sure to use the right type for the `init` argument:

```
std::vector<double> vec;

// ... Adding elements to vec

double const sum =
   std::accumulate( vec.begin(), vec.end(), double{} );
```

Limitations of STL Algorithms - Example 3

Guideline: Beware the power of the third argument of `std::accumulate()`, `std::reduce()`, and similar algorithms.

Limitations of STL Algorithms - Example 4

Task (3_Concepts_and_STL/Algorithms/BadRemove): Explain the error in the following program:

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```

Limitations of STL Algorithms - Example 4

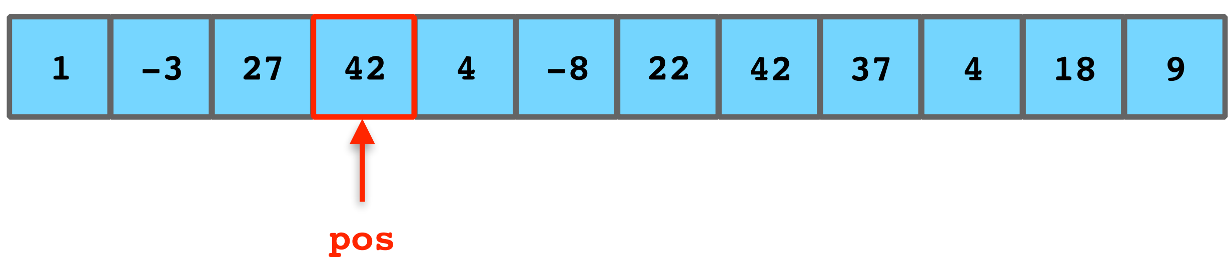
Task (3_Concepts_and_STL/Algorithms/BadRemove): Explain the error in the following program:

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```

- `std::remove()` takes its third argument by reference
- passing a reference to the value to be removed may result in aliasing effects
- In case of aliasing final result may be wrong!

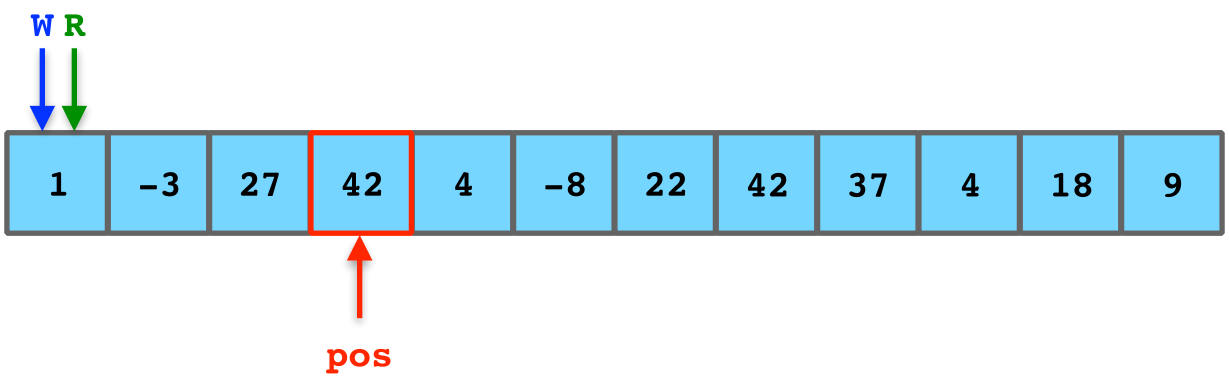
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



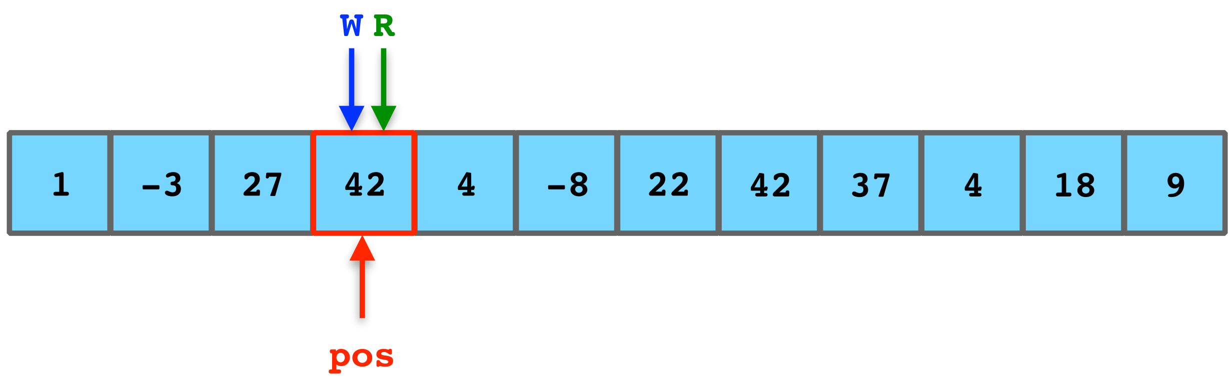
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



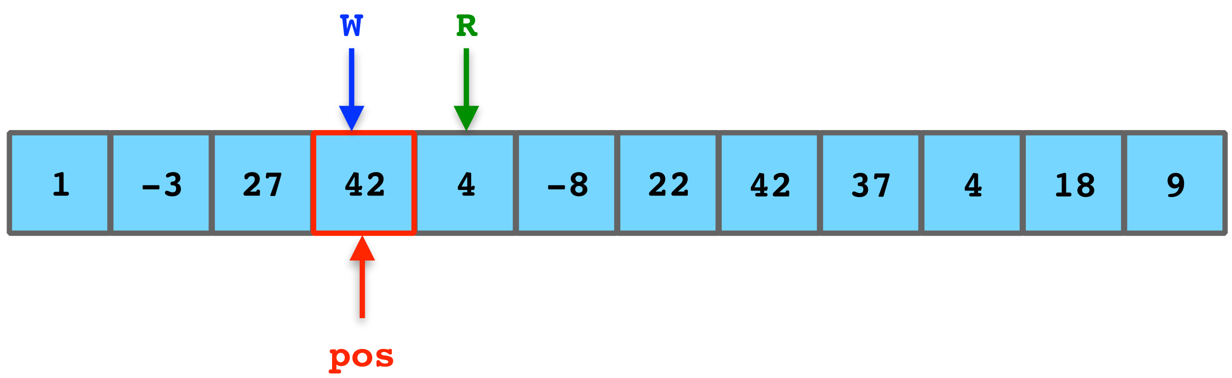
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



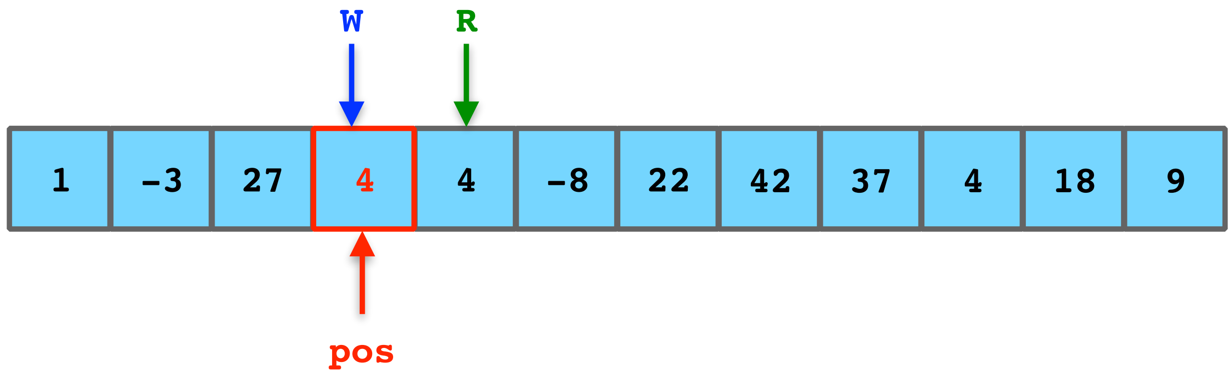
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



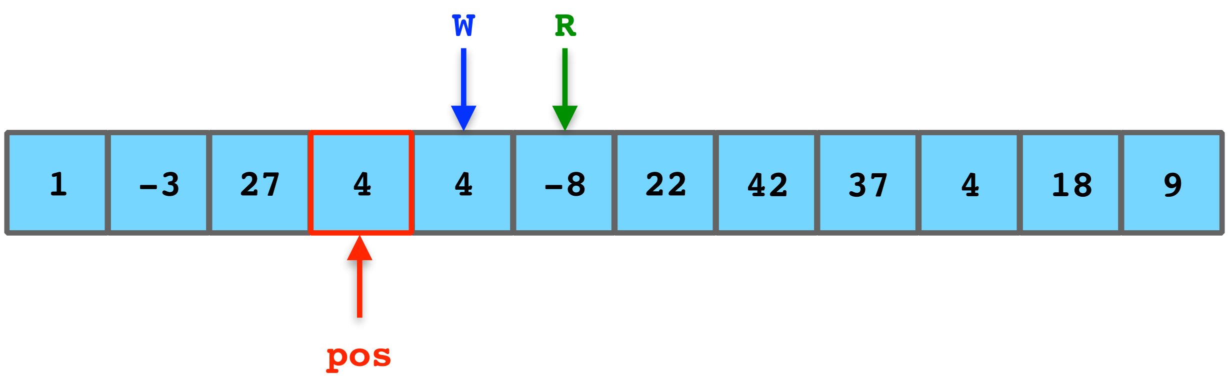
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



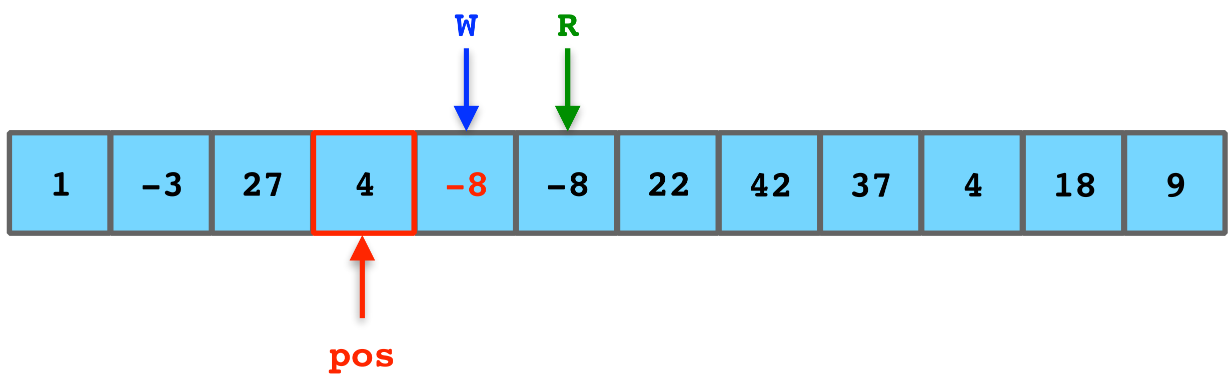
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



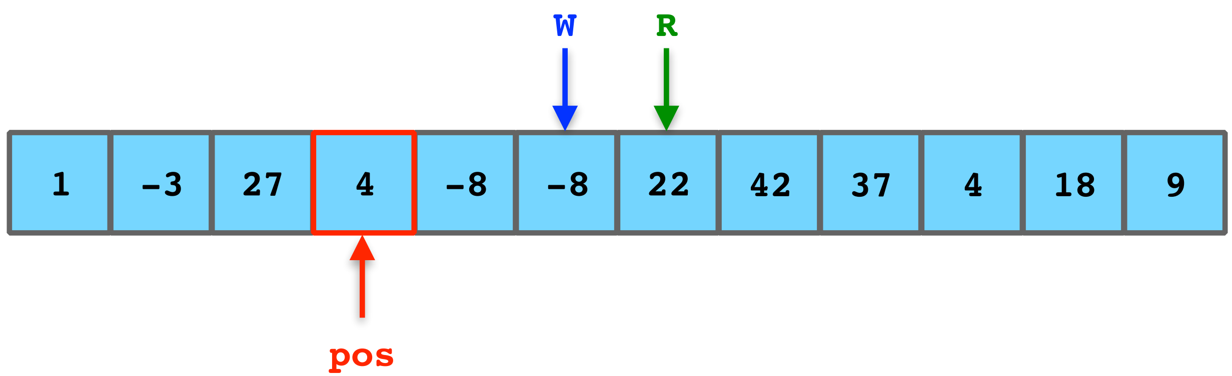
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



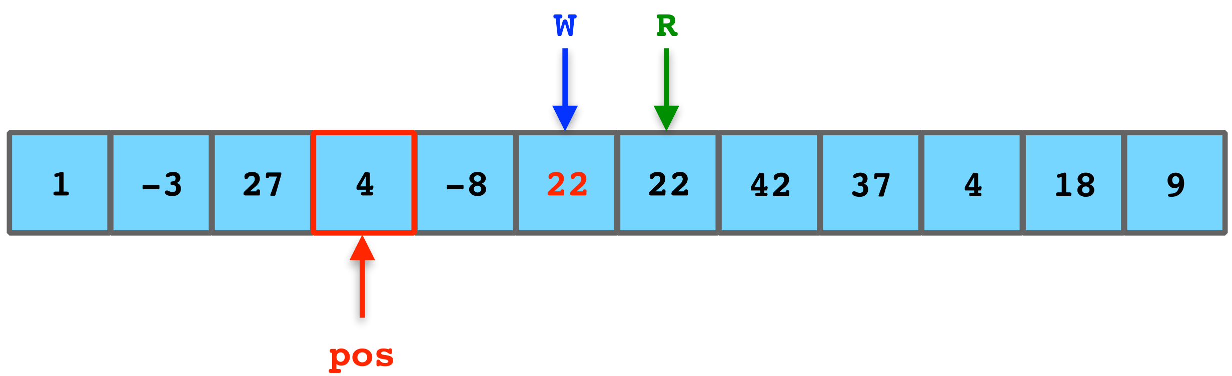
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



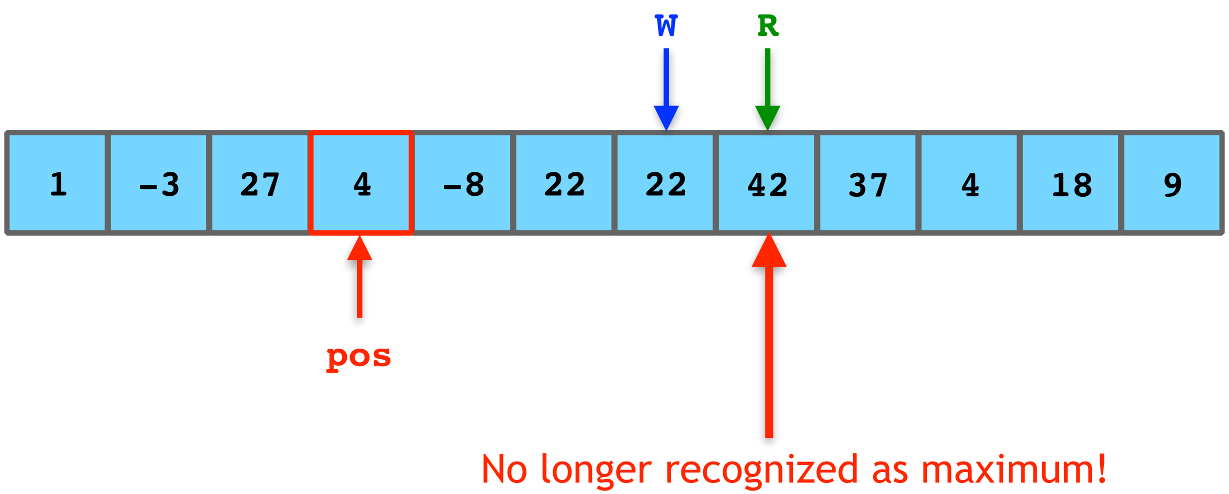
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



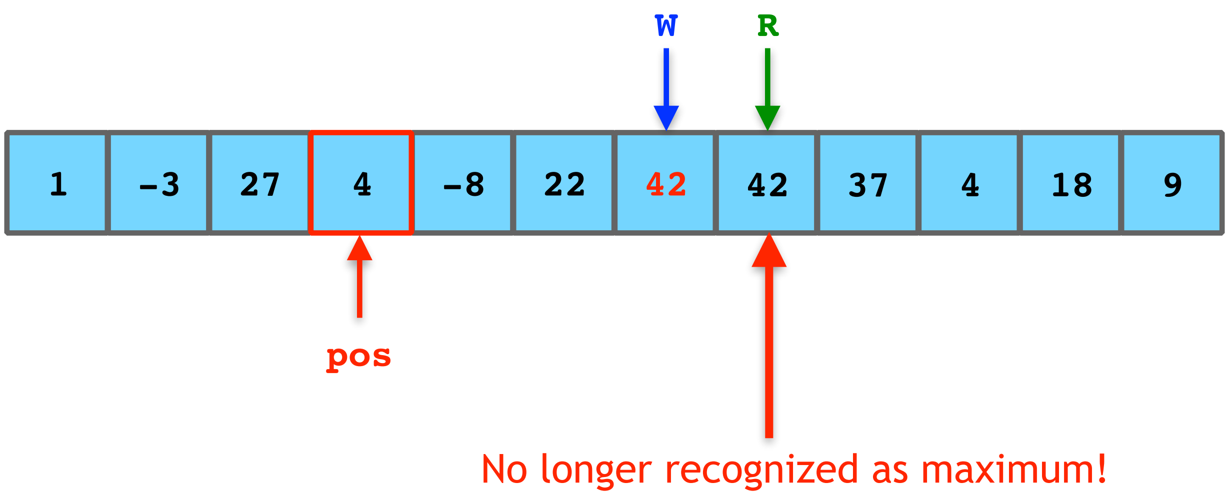
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



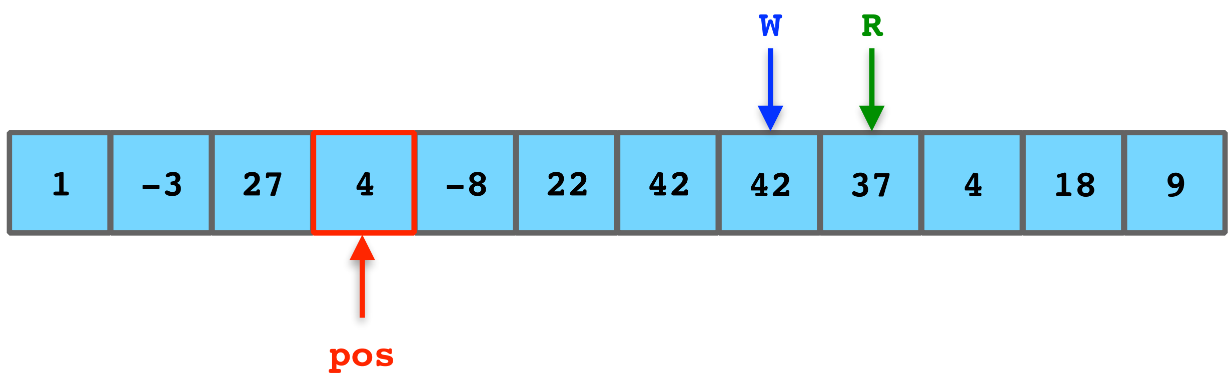
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



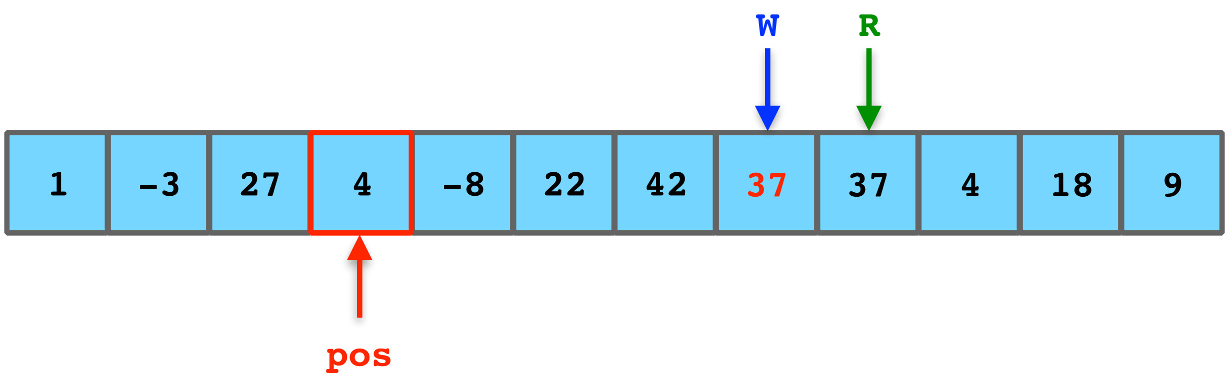
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



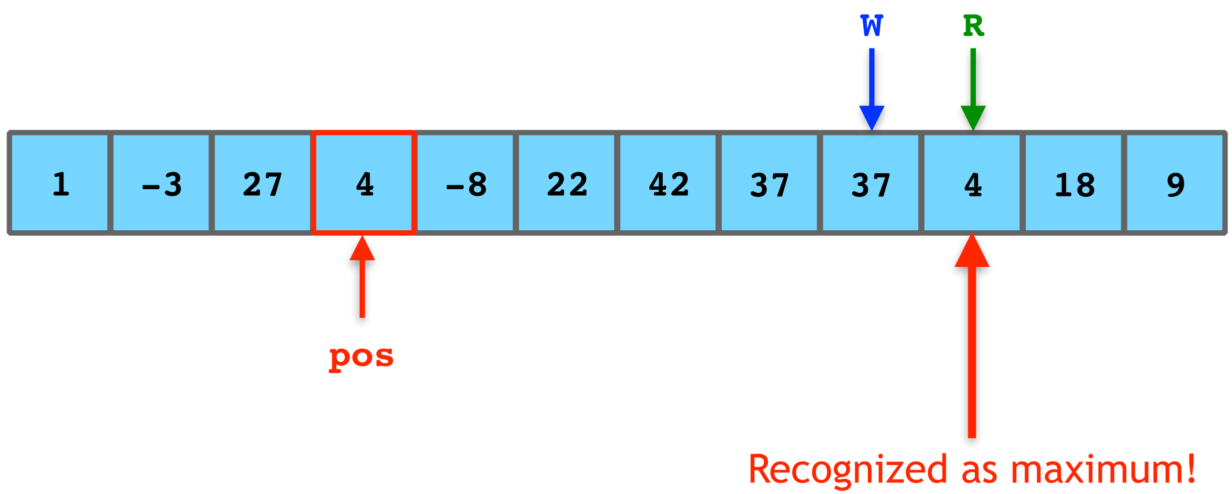
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



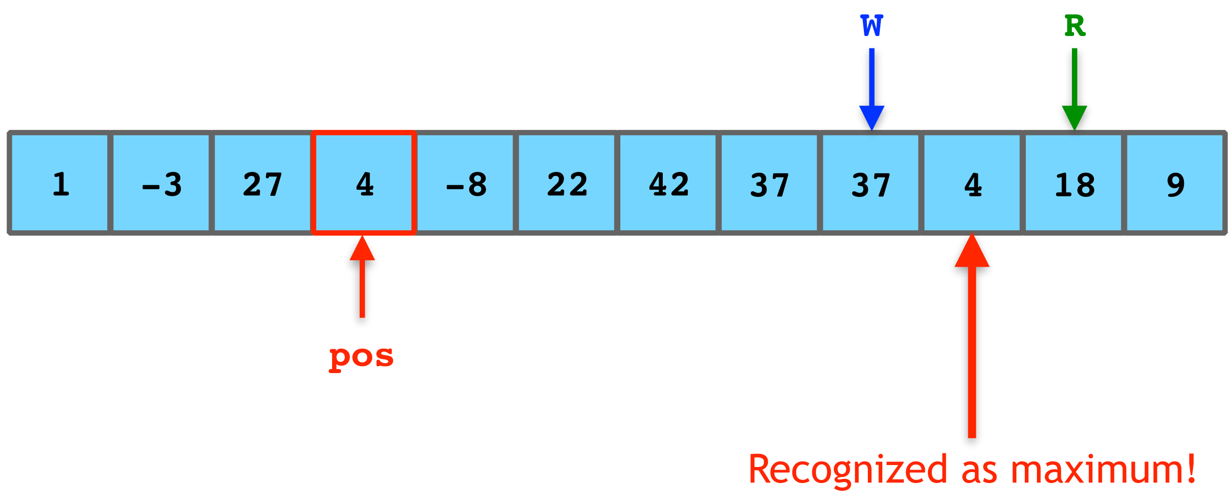
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



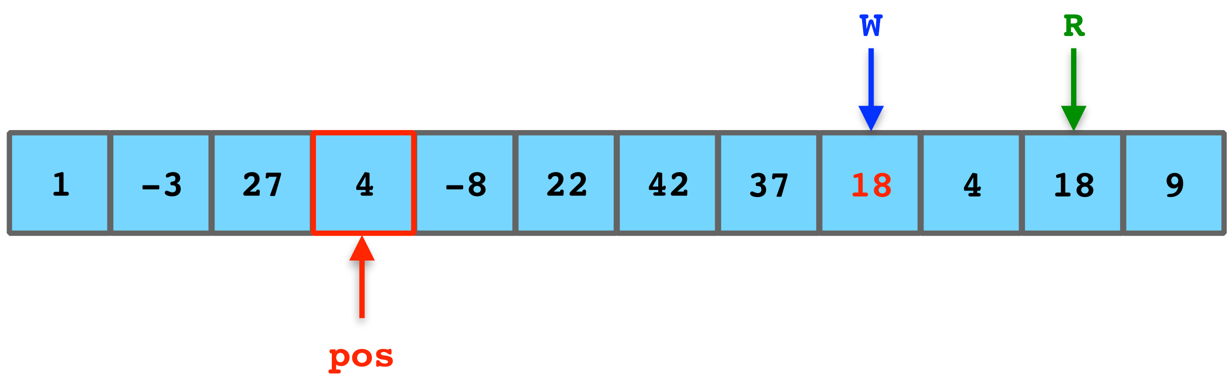
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



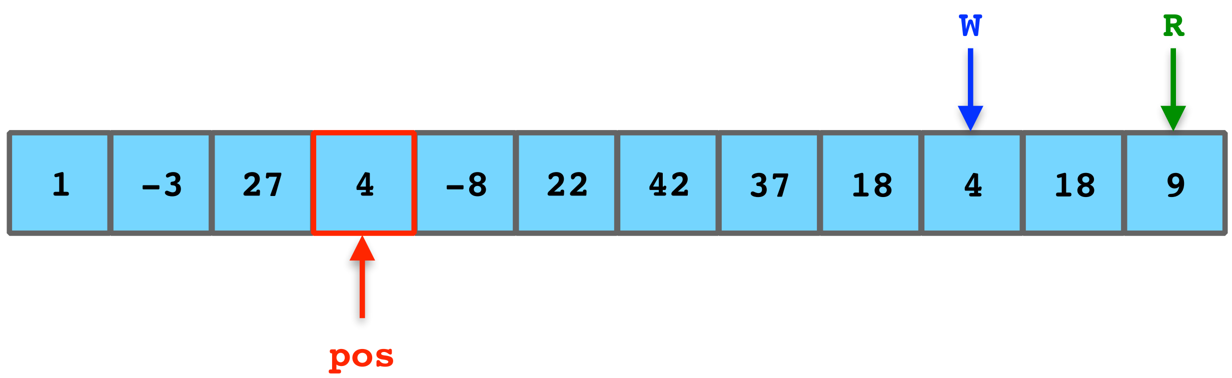
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



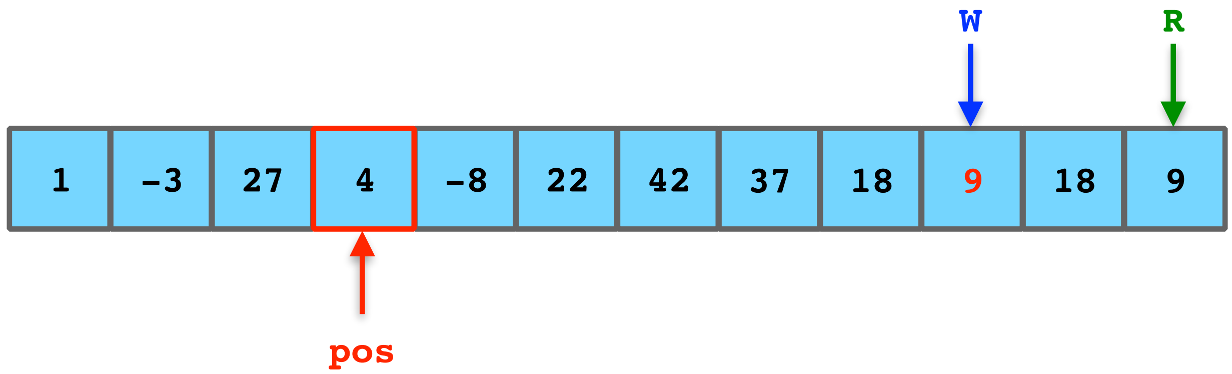
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



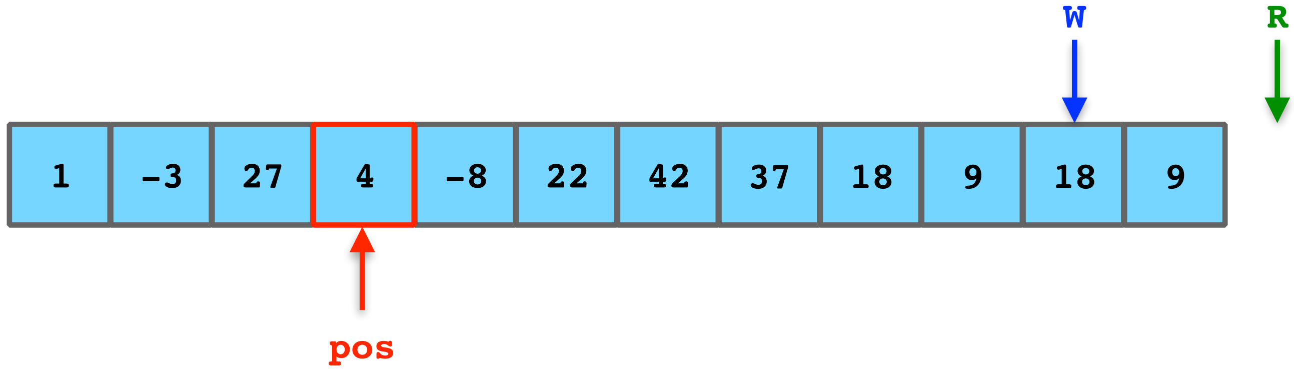
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };
auto const pos = std::max_element( begin(vec), end(vec) );
vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



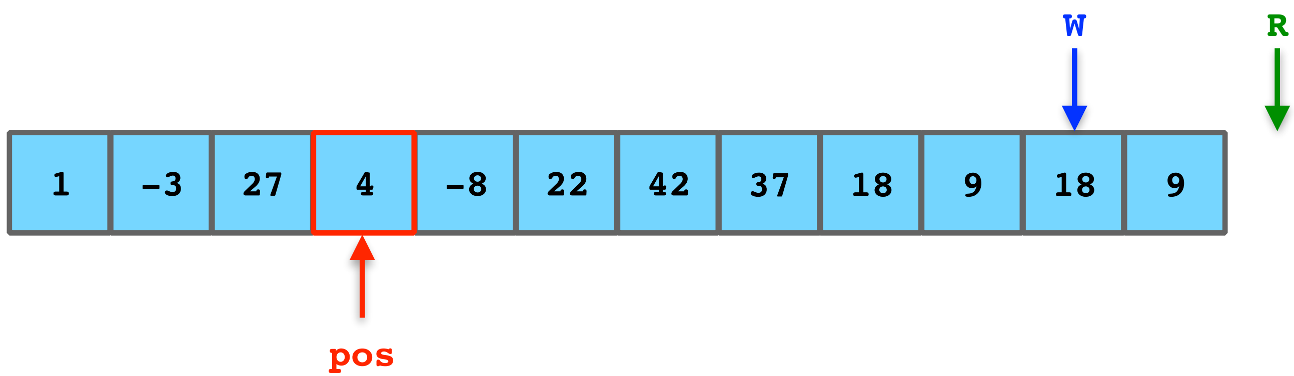
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



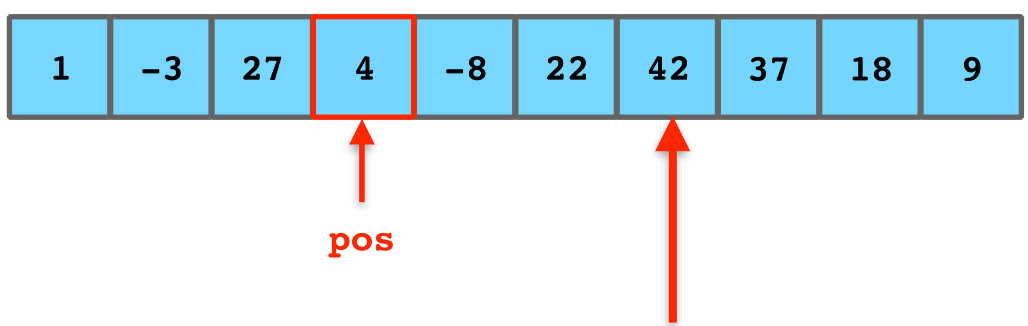
Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



Limitations of STL Algorithms - Example 4

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );
```



The old maximum is still present!

Limitations of STL Algorithms - Example 4

Make sure to evaluate the value in case there is aliasing:

```
std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

auto const pos = std::max_element( begin(vec), end(vec) );

vec.erase( std::remove( begin(vec), end(vec), int{*pos} ), end(vec) );
```

Limitations of STL Algorithms - Example 4

Guideline: Beware the few reference arguments in the STL.

Limitations of STL Algorithms - Example 5

Task (3_Concepts_and_STL/Algorithms/Simpson): Consider the following implementation for the Simpson `order_by_lastname()` function:

```
std::stable_sort( std::begin(table), std::end(table),
   std::not_fn( []( Person const& lhs, Person const& rhs ) {
      return lhs.lastname < rhs.lastname;
   } ) );
```

Explain the error in the implementation.

Limitations of STL Algorithms - Example 5

Task (3_Concepts_and_STL/Algorithms/Simpson): Consider the following implementation for the Simpson `order_by_lastname()` function:

```
std::stable_sort( std::begin(table), std::end(table),
   std::not_fn( []( Person const& lhs, Person const& rhs ) {
      return lhs.lastname < rhs.lastname;
   } ) );
```

Explain the error in the implementation.

- All sorting algorithms (including `std::nth_element`) are based on equivalence (`!(a<b) && !(b<a)`), not on equality (`a == b`)
- The negation of the lambda result in a >= comparison (including equality!)
- That comparison does not adhere to the sorting requirements: Undefined behavior!

Limitations of STL Algorithms - Example 5

Possible output:

```
Enter command: r
Bart       Simpson    10
Marge      Simpson    34
Hans       Moleman    33
Ralph      Wiggum      8
Montgomery Burns     104         // Random order of characters after
Homer      Simpson    38         // a call to std::shuffle
Lisa       Simpson     8
Maggie     Simpson     1
Jeff       Albertson  45

Enter command: l
Ralph      Wiggum      8
Maggie     Simpson     1
Lisa       Simpson     8
Homer      Simpson    38         // Order of characters after a call to
Marge      Simpson    34         // std::stable_sort. The order of equal
Bart       Simpson    10         // elements is NOT preserved!
Hans       Moleman    33
Montgomery Burns     104
Jeff       Albertson  45
```

Limitations of STL Algorithms - Example 6

Task (3_Concepts_and_STL/Algorithms/BadFind): Explain the problem in the following program.

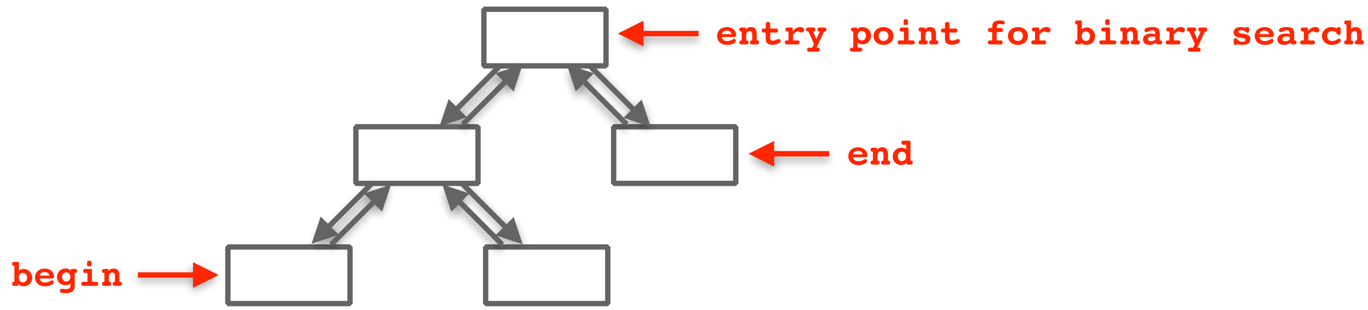
```
std::set<int> s{ /*...*/ };

auto const pos = std::find( std::begin(s), std::end(s), 42 );
```

Limitations of STL Algorithms - Example 6

Task (3_Concepts_and_STL/Algorithms/BadFind): Explain the problem in the following program.

```
std::set<int> s{ /*...*/ };

auto const pos = std::find( std::begin(s), std::end(s), 42 );
```

- All find() algorithm cannot exploit the tree structure of the `std::set` due to the `begin` and `end` iterators



- This results in a linear search instead of a binary search

Guidelines

Guideline: If available, prefer member functions to general algorithms (`find()`, `lower_bound()`, `upper_bound()`, …).

Wait a Second…

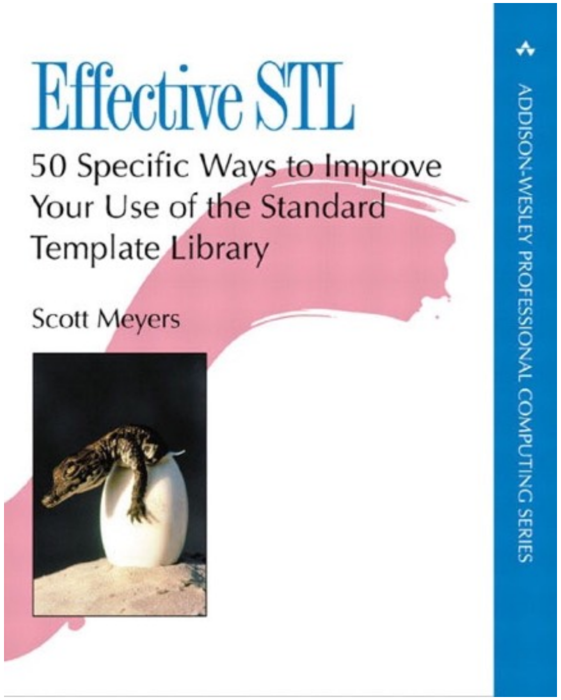
Can't I overload the free `find()` algorithm to call the member function?

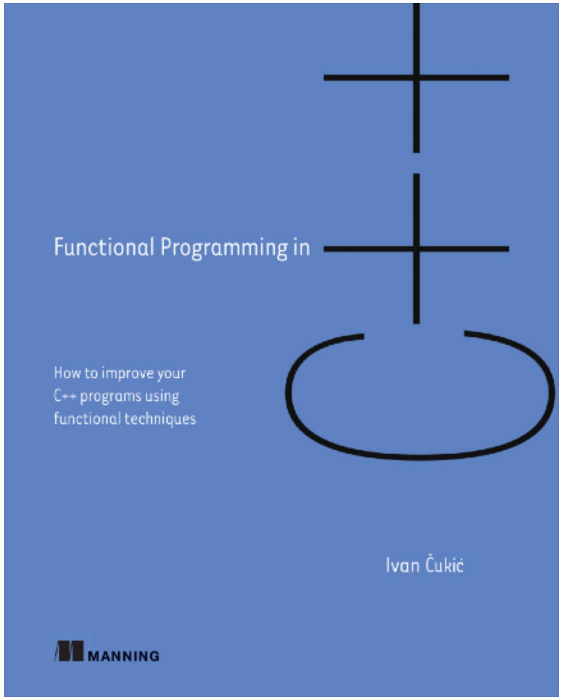
No, unfortunately not. You would need a reference to the container to call the member function, but the algorithm is only given iterators. However, it is possible in C++20 😃

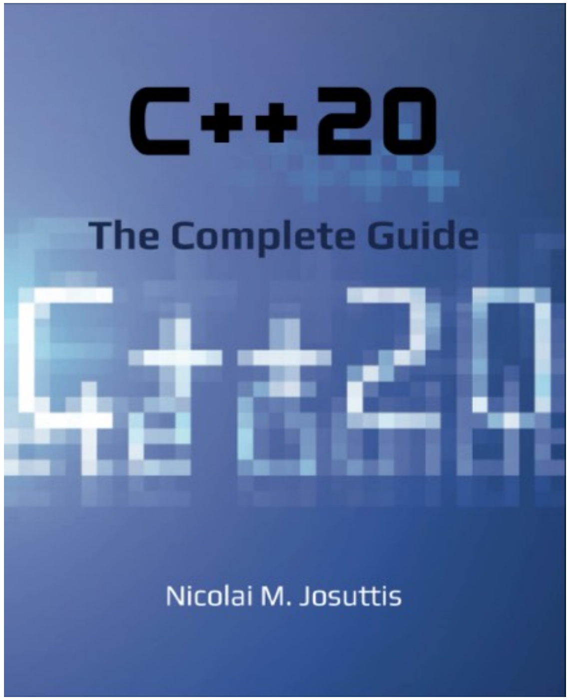
Things to Remember

- Familiarize yourself with the STL and STL-style code
- Understand the importance of concepts
- Prefer algorithms over handwritten loops
- Remember the conventions and possible pitfalls of algorithms

Literature


Effective STL
50 Specific Ways to Improve Your Use of the Standard Template Library
Scott Meyers
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Functional Programming in C++
How to improve your C++ programs using functional techniques
Ivan Čukić
MANNING


C++20
The Complete Guide
Nicolai M. Josuttis

References

- Alexander Stepanov, "STL and Its Design Principles". (https://www.youtube.com/watch?v=COuHLky7E2Q)
- Bjarne Stroustrup, "C++11 Style". GoingNative 2012 (http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style)
- Sean Parent, "C++ Seasoning", GoingNative 2013 (https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning)
- Jonathan Boccara, "105 STL Algorithms in Less Than An Hour". CppCon 2018 (https://www.youtube.com/watch?v=2olsGf6JIkU)
- Chandler Carruth, "Efficiency with Algorithms, Performance with Data Structures". cppcon 2014 (https://www.youtube.com/watch?v=fHNmRkzxHWs)
- Michael VanLoon, "STL Algorithms in Action". CppCon 2015 (https://www.youtube.com/watch?v=eidEEmGLQcU)
- Eric Niebler, "Ranges for the Standard Library". CppCon 2015 (https://www.youtube.com/watch?v=mFUXNMfaciE)
- Jeff Garland, "Effective Ranges: A Tutorial for Using C++2x Ranges". CppCon 2023 (TBA)

References

- Ben Deane, "std::accumulate: Exploring an Algorithmic Empire". CppCon 2016 (https://www.youtube.com/watch?v=B6twozNPUoA)
- Frederic Tingaud, "A Little Order: Delving into the STL sorting algorithms". CppCon 2018 (https://www.youtube.com/watch?v=-0tO3Eni2uo)
- Pete Isensee, "Evolution of a Median Algorithm". CppCon 2023 (TBA)
- Conor Hoekstra, "Algorithm Intuition (Part 1 of 2)". CppCon 2019 (https://www.youtube.com/watch?v=pUEnO6SvAMo)
- Barry Revzin, "Iterators and Ranges: Comparing C++ to D, Rust, and Others". CPPP 2021 (https://www.youtube.com/watch?v=95uT0RhMGwA)
- Bob Steagall, "Back to Basics: Classic STL". CppCon 2021 (https://www.youtube.com/watch?v=tXUXl_RzkAk)
- Arthur O'Dwyer, "Back to Basics: Lambdas from Scratch". CppCon 2019 (https://www.youtube.com/watch?v=3jCOwajNch0)
- Scott Wlaschin, "Function Programming Design Patterns". NDC London 2014 (https://www.youtube.com/watch?v=E8I19uA-wGY)

email: klaus.iglberger@gmx.de

LinkedIn: linkedin.com/in/klaus-iglberger-2133694/

Xing: xing.com/profile/Klaus_Iglberger/cv