C++ Essentials: STL Algorithms

2. STL Algorithms

Klaus Iglberger
May 9th, 2025

Content

1. Terminology
2. Overview of the STL
3. Motivation
4. STL Iterators
5. STL Algorithms

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

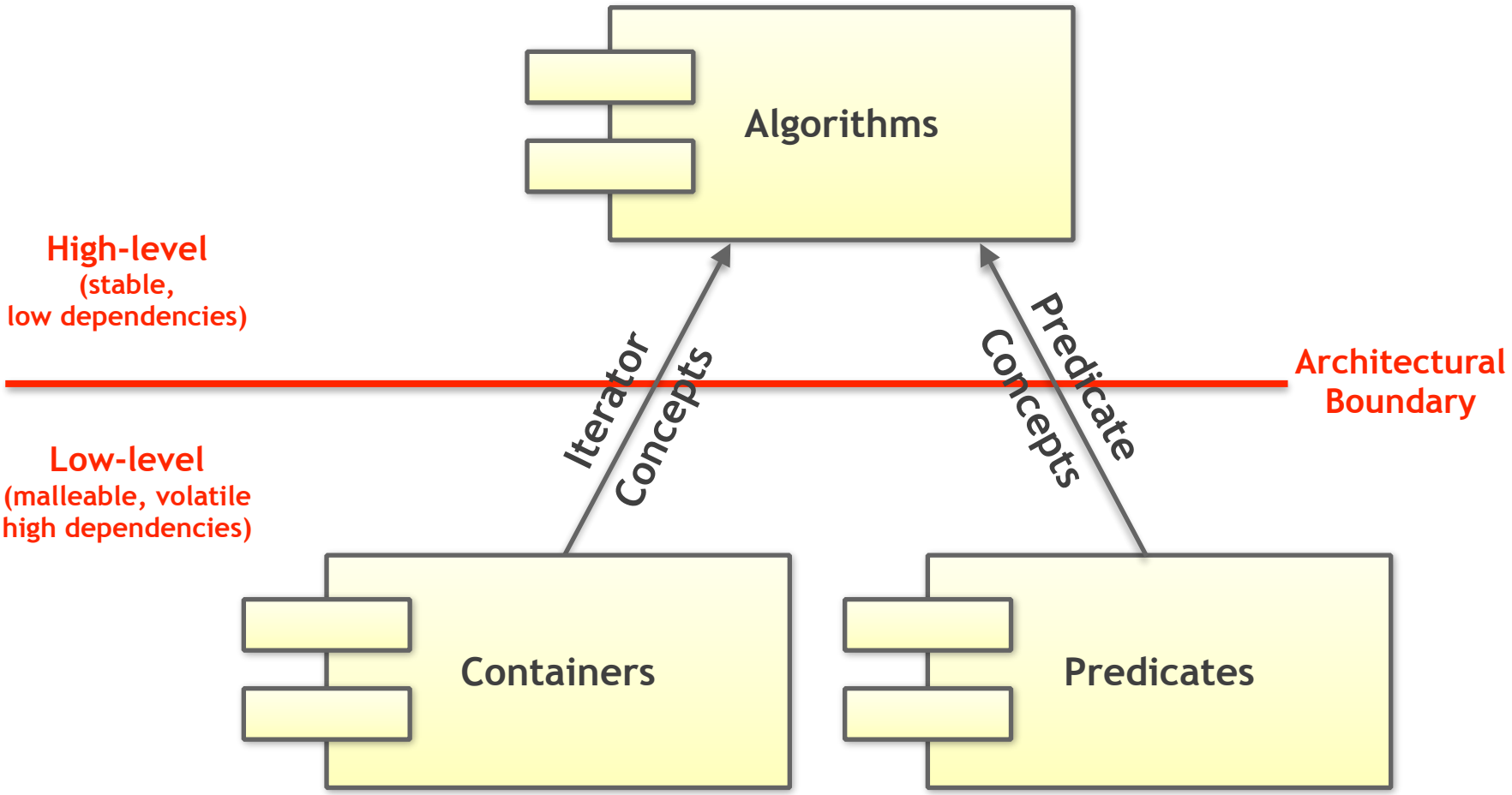
The STL in a Nutshell

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)

The STL in a Nutshell



2.3. Motivation

Example: std::copy

The copy algorithm …

```
template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt begin, InputIt end, OutputIt dst );
```

- can be used with all data types that adhere to the required concepts
- is not depending on any data structure (no inheritance!)
- adheres to the Single-Responsibility Principle (SRP)
- adheres to the Open-Closed Principle (OCP)
- adheres to the Interface Segregation Principle (ISP)
- adheres to the Dependency Inversion Principle (DIP)

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)

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++~~

++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

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

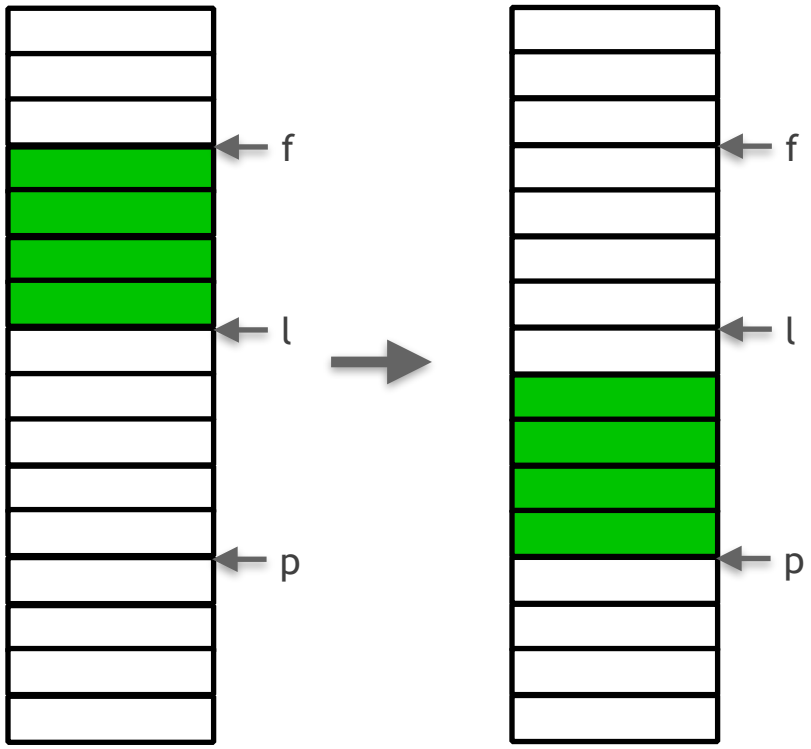
```
std::sort( vec.begin(), vec.end() );
vec.erase( std::unique( vec.begin(), vec.end() ), vec.end() );
```

- 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

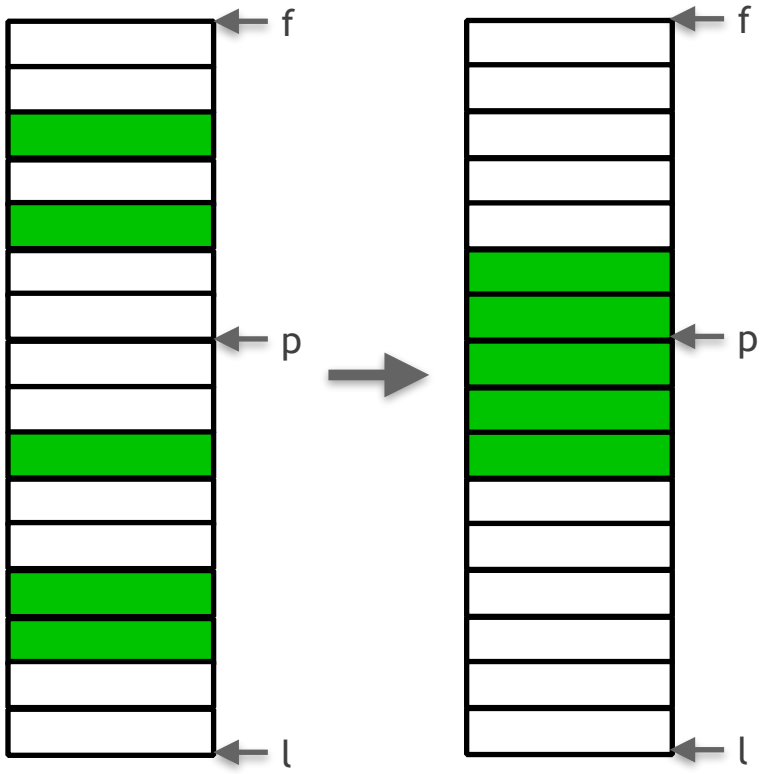
- Move a number of consecutive elements in a vector




```
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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 of age 6 to 17 first
- Compute the total length of all last names
- Check if two adjacent persons have the same age
- Determine the median age of all persons
- After ordering all persons by last name, find all the Simpsons
- Compute the maximum age difference between two adjacent persons
- Print a string containing the first names of all children

Programming Task

Task (2_STL_Algorithms/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 (2_STL_Algorithms/Algorithms/Accumulate_1):

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 (2_STL_Algorithms/Algorithms/Accumulate_2): Use the `accumulate()` algorithm to simulate other standard algorithms, in particular `std::find()` and `std::sort()`. You may use other algorithms, but not the algorithm being simulated by `std::accumulate()`.

Programming Task

Task (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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

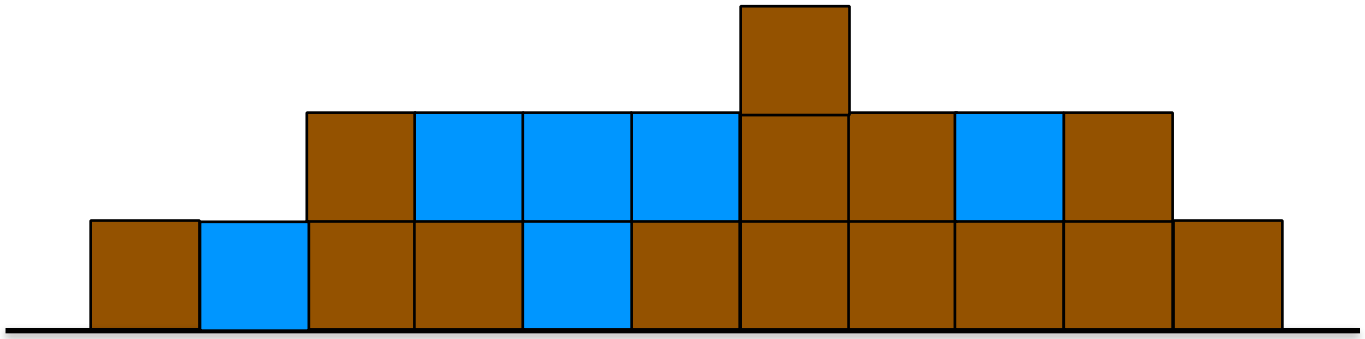
Task (2_STL_Algorithms/Algorithms/ExtractStrings): Implement the `extract_strings()` algorithm. The algorithm should extract all strings from a long string of space-separated words.

Programming Task

Task (2_STL_Algorithms/Algorithms/LongestStreak): Determine the longest streak of consecutive equal values in the given range of elements.

Programming Task

Task (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/Algorithms/RangesRefactoring_Birthday):

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 (2_STL_Algorithms/Algorithms/RangesRefactoring_Countries):

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

Step 2: Considering the general case, the initial code contains a bug. For which input does the given solution not work?

Step 3: Improve readability by choosing better names for the variables.

Step 4: Refactor the `main()` function from an imperative to a declarative style by means of C++20 ranges.

Programming Task

Task (2_STL_Algorithms/Algorithms/RangesRefactoring_Animals):

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

Step 2: Improve readability by choosing better names for the variables.

Step 3: Refactor the `main()` function from an imperative to a declarative style by means of C++20 ranges.

Programming Task

Task (2_STL_Algorithms/Algorithms/RangesRefactoring_Recipes):

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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/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

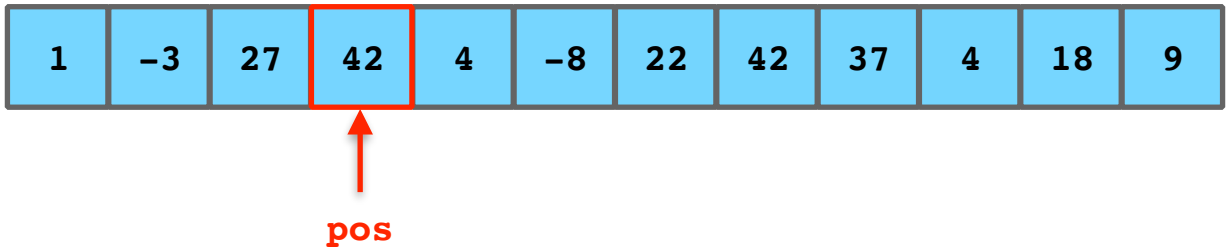
Task (2_STL_Algorithms/Algorithms/BadRemove): Explain the error in the following program:

```
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!

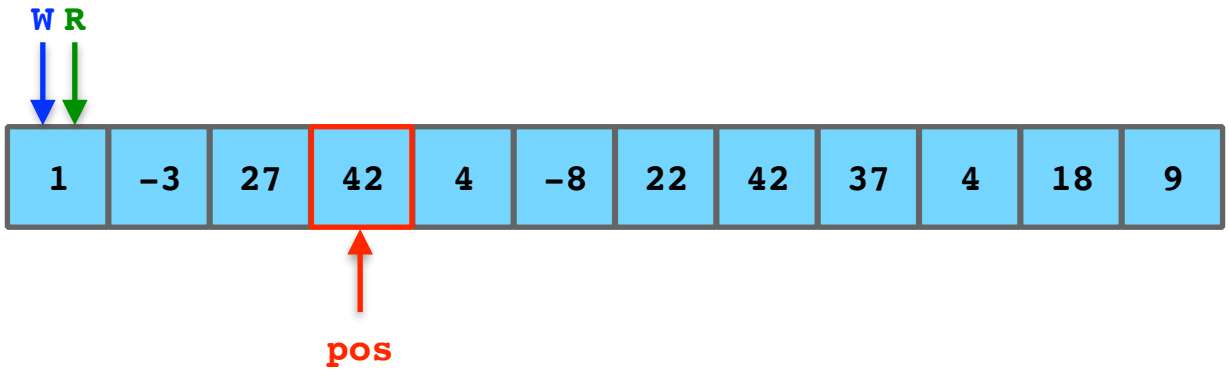
Limitations of STL Algorithms - Example 4

```
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) );
```



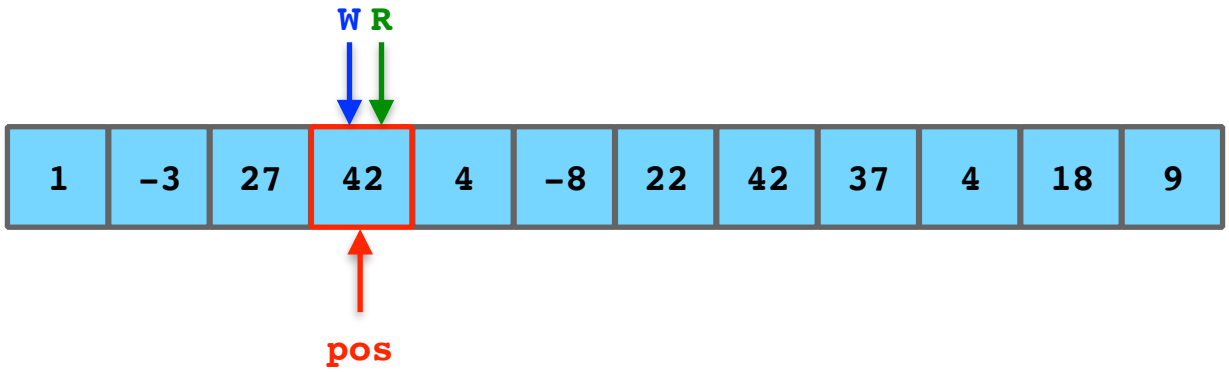
Limitations of STL Algorithms - Example 4

```
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) );
```



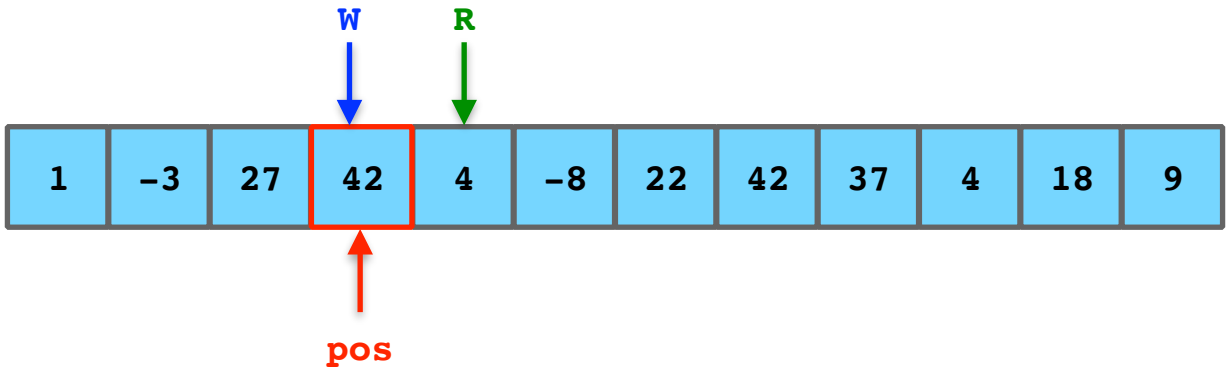
Limitations of STL Algorithms - Example 4

```
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) );
```



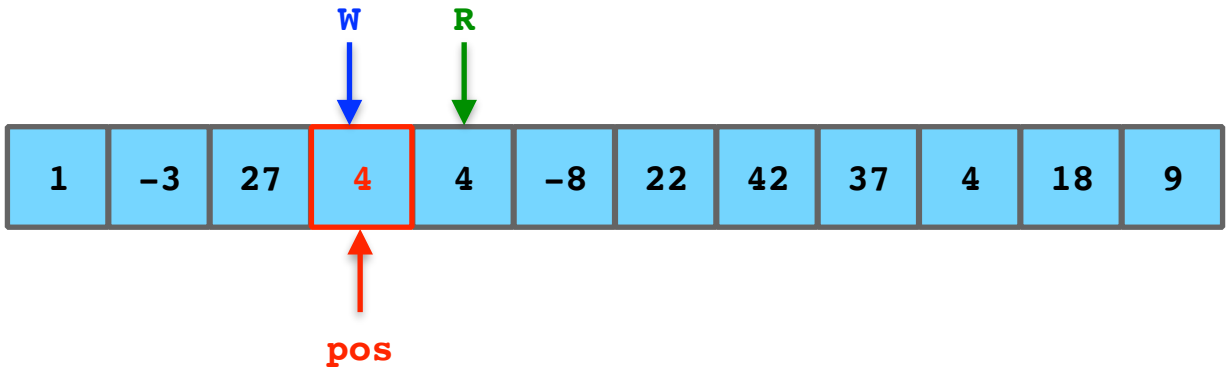
Limitations of STL Algorithms - Example 4

```
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) );
```



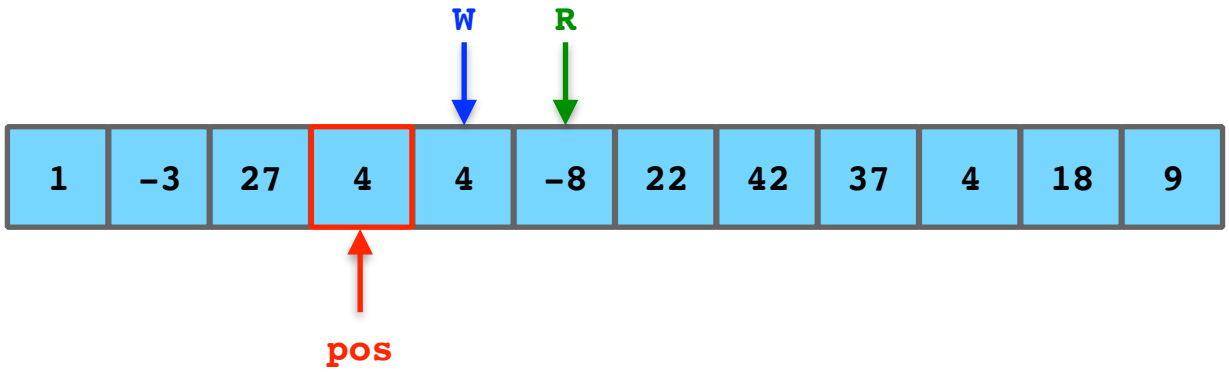
Limitations of STL Algorithms - Example 4

```
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) );
```



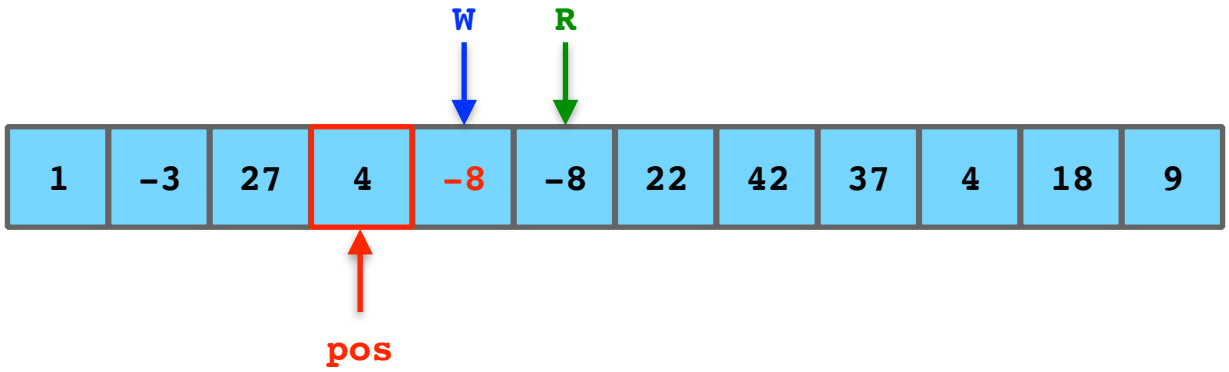
Limitations of STL Algorithms - Example 4

```
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) );
```



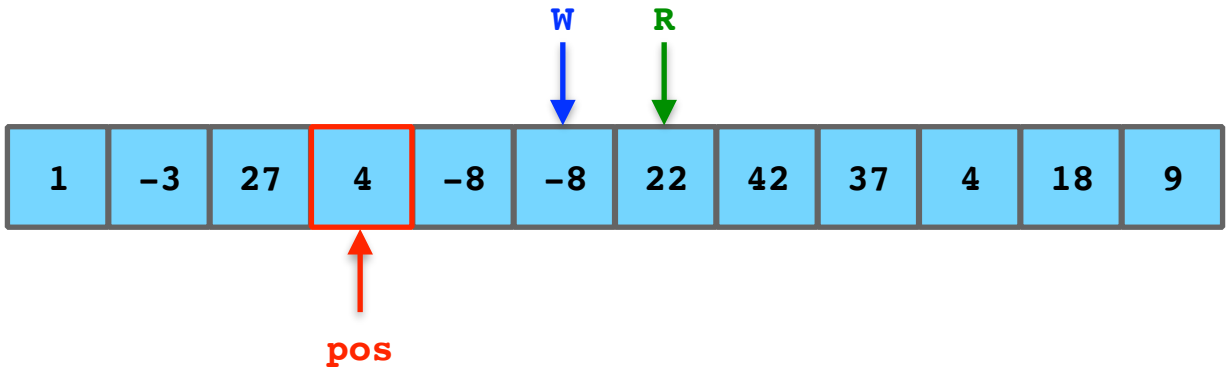
Limitations of STL Algorithms - Example 4

```
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) );
```



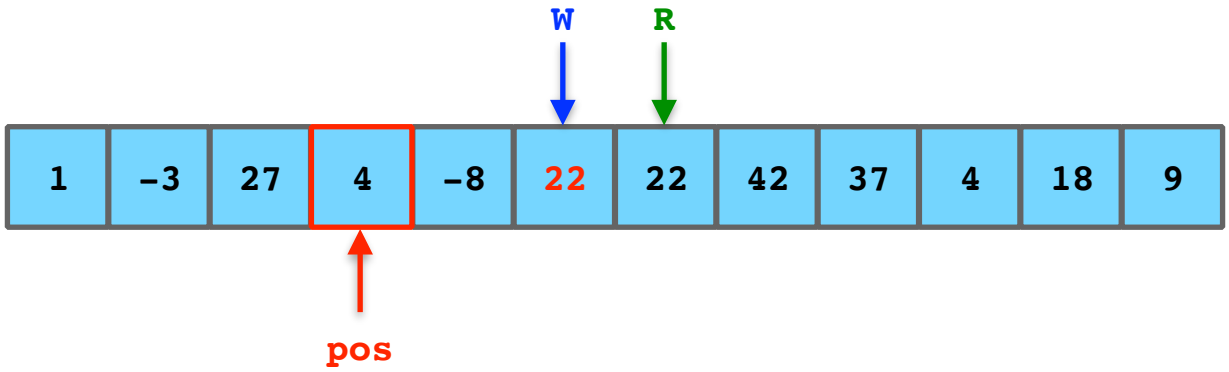
Limitations of STL Algorithms - Example 4

```
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) );
```



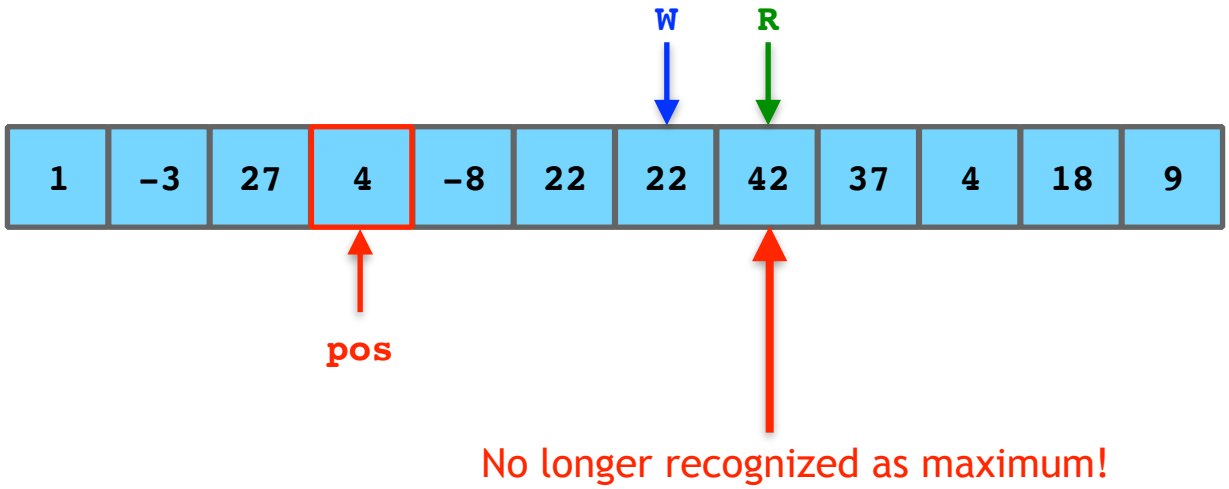
Limitations of STL Algorithms - Example 4

```
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) );
```



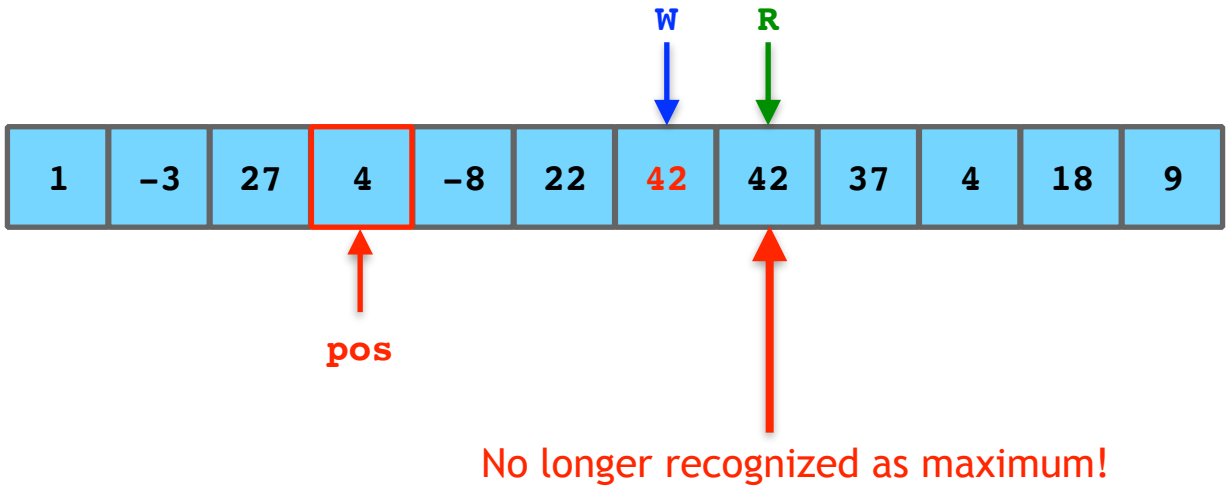
Limitations of STL Algorithms - Example 4

```
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) );
```



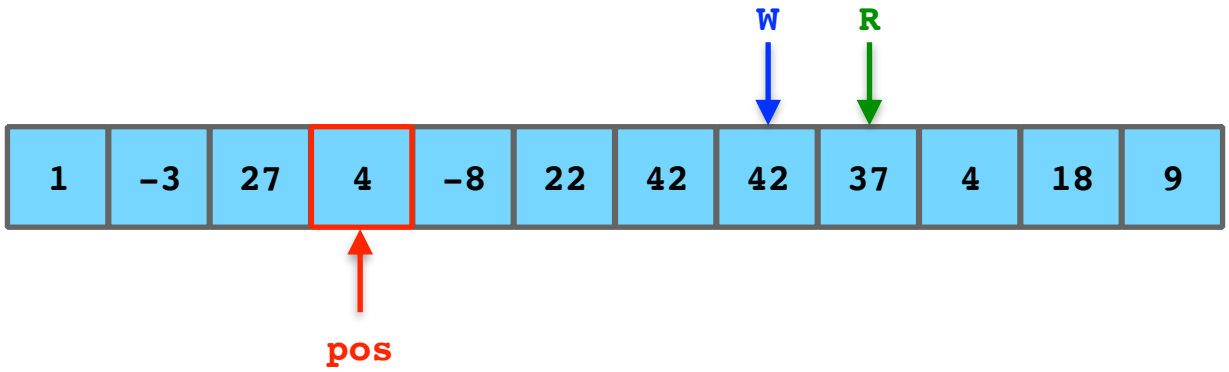
Limitations of STL Algorithms - Example 4

```
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) );
```



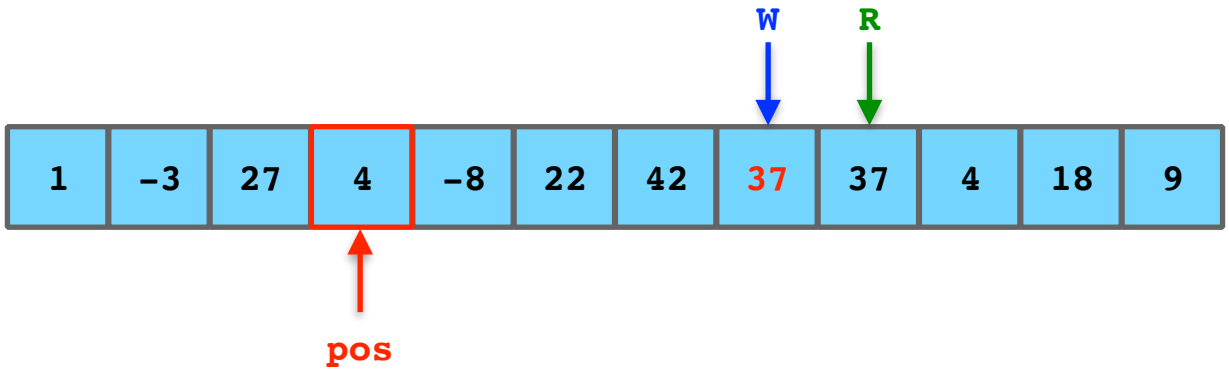
Limitations of STL Algorithms - Example 4

```
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) );
```



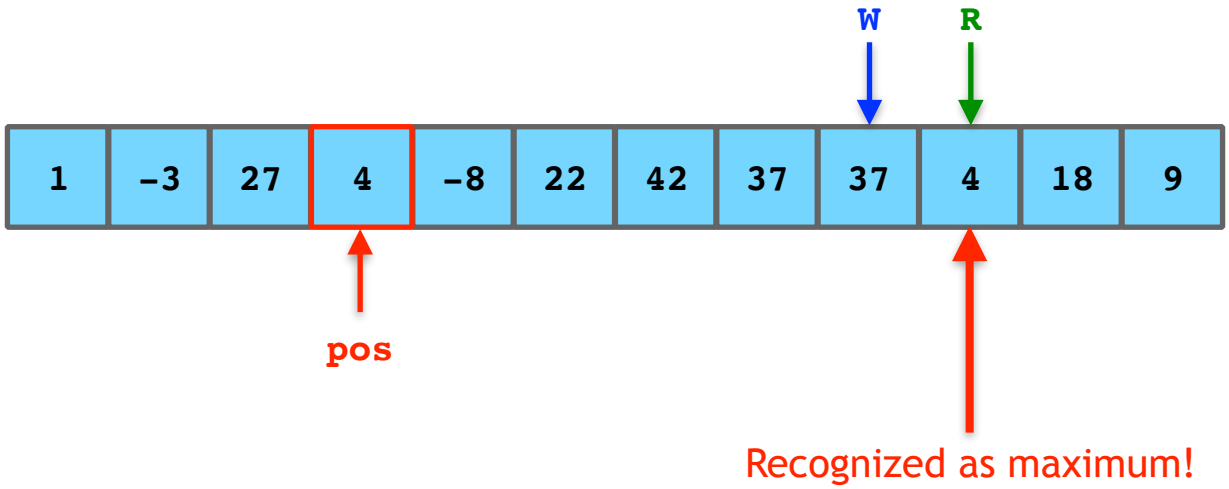
Limitations of STL Algorithms - Example 4

```
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) );
```



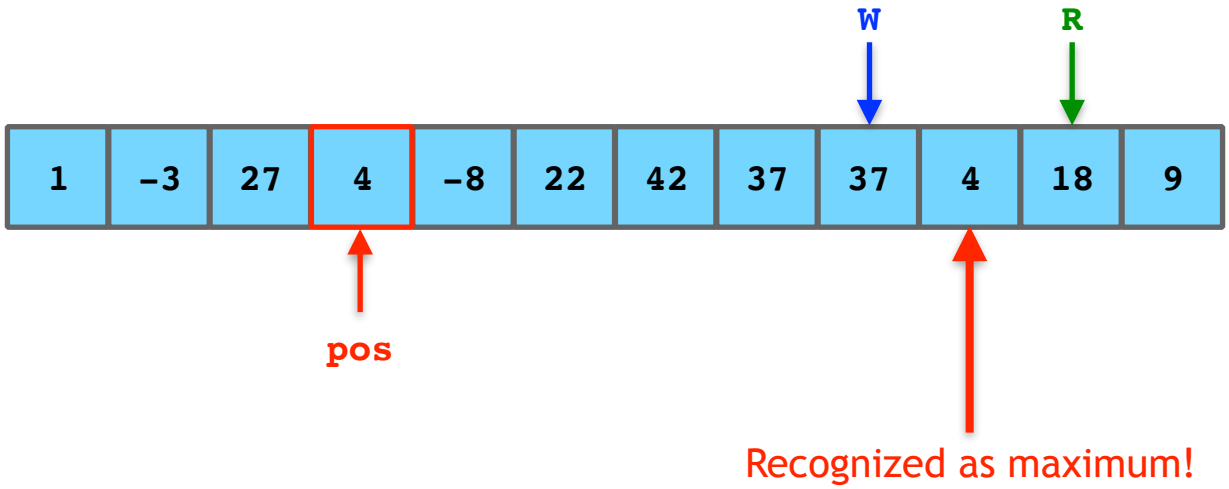
Limitations of STL Algorithms - Example 4

```
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) );
```



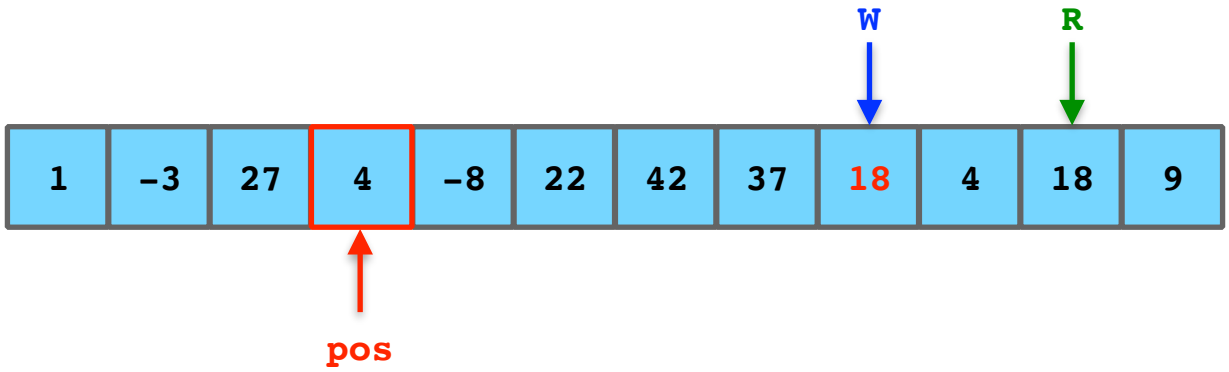
Limitations of STL Algorithms - Example 4

```
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) );
```



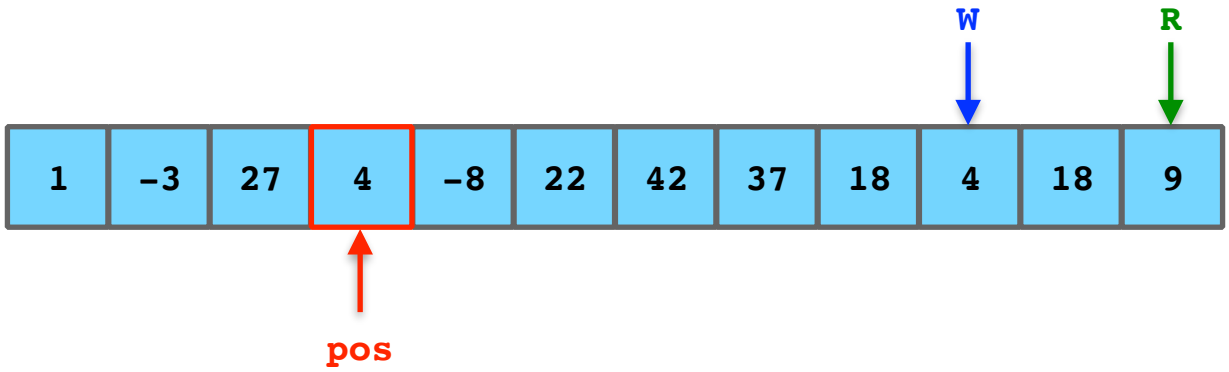
Limitations of STL Algorithms - Example 4

```
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) );
```



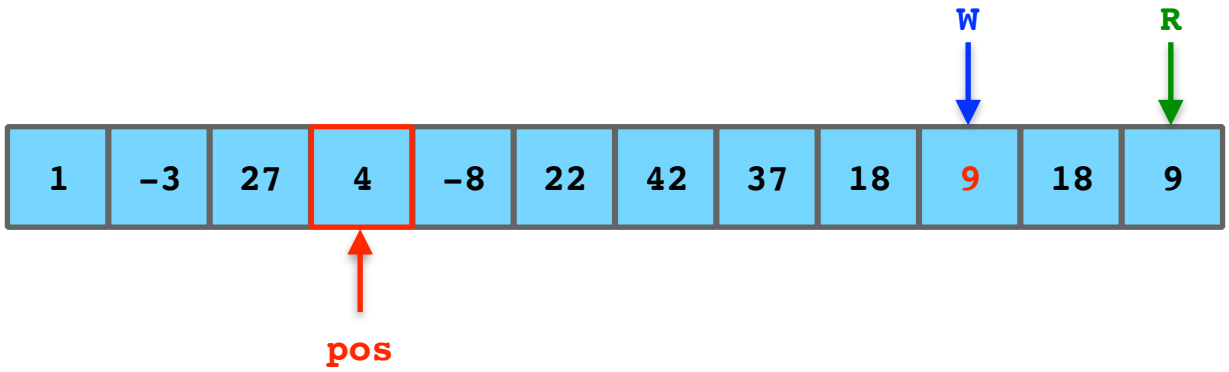
Limitations of STL Algorithms - Example 4

```
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) );
```



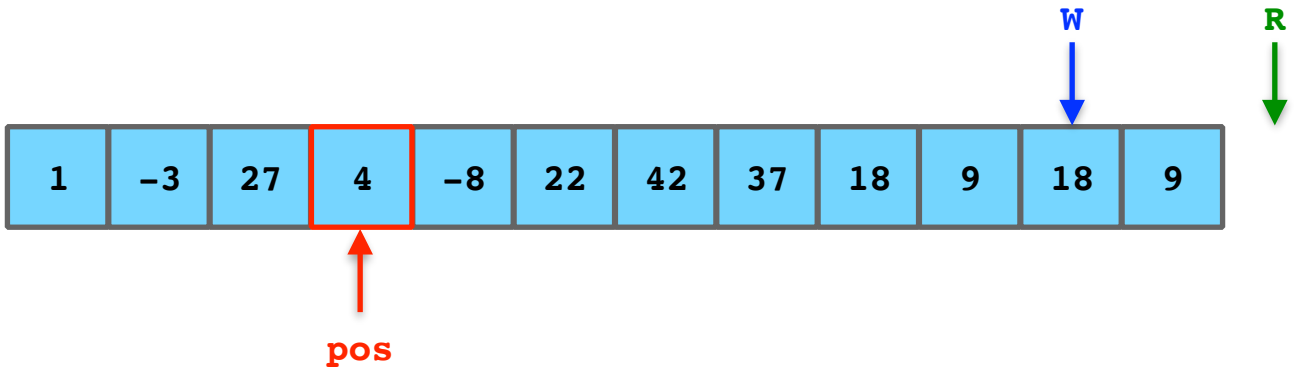
Limitations of STL Algorithms - Example 4

```
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) );
```



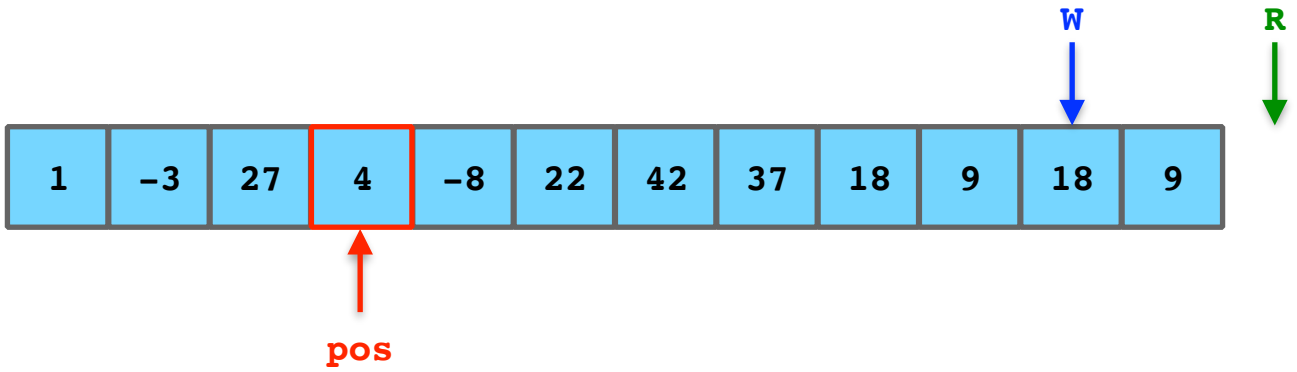
Limitations of STL Algorithms - Example 4

```
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) );
```



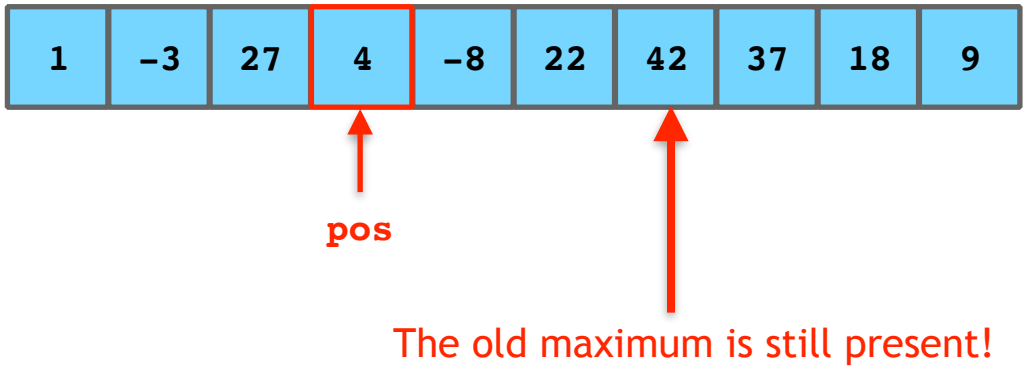
Limitations of STL Algorithms - Example 4

```
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) );
```



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 (2_STL_Algorithms/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 (2_STL_Algorithms/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 (2_STL_Algorithms/Algorithms/BadFind): Explain the problem in the following program.

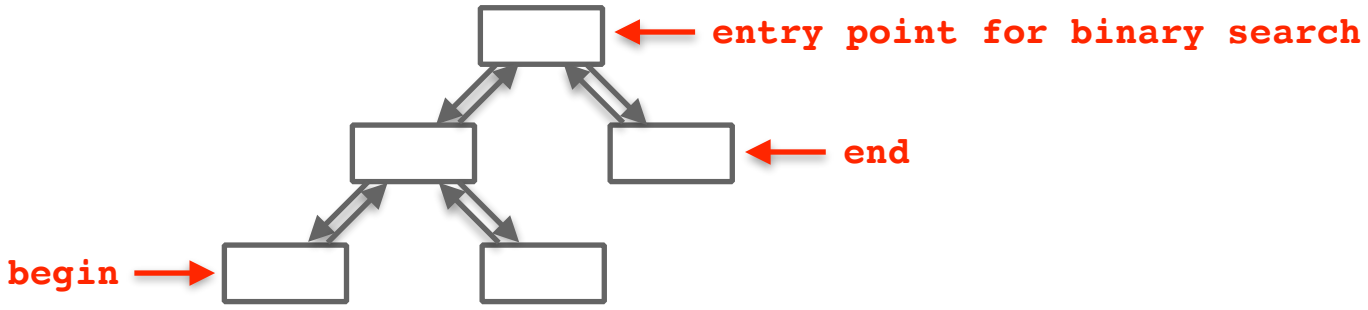
```
std::set<int> s{ /*...*/ };

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

Limitations of STL Algorithms - Example 6

Task (2_STL_Algorithms/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…

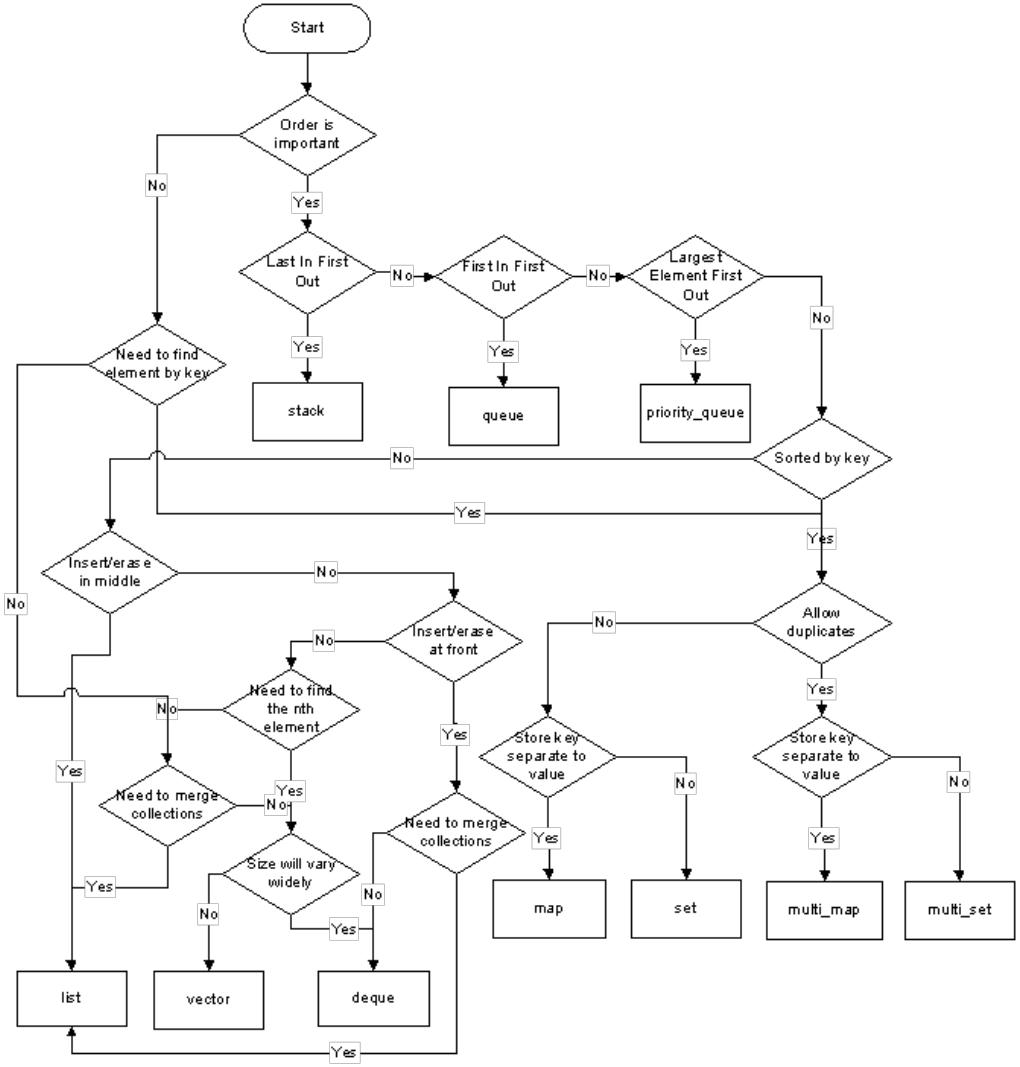
Can’t I overload the free `find()` algorithm to call the member function?

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 😀

Choosing a Container

Choosing a container is about performance …

How Not to Choose a Container



The Expert's View on Performance

"C++ doesn't give you performance, it gives you control over performance."

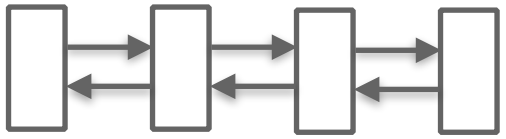
(Chandler Carruth, cppcon 2014)

The Standard Containers

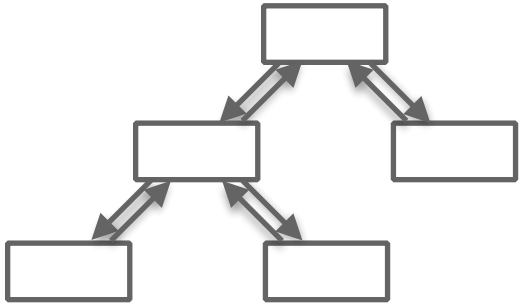
`std::vector &`
`std::array`

`std::deque`

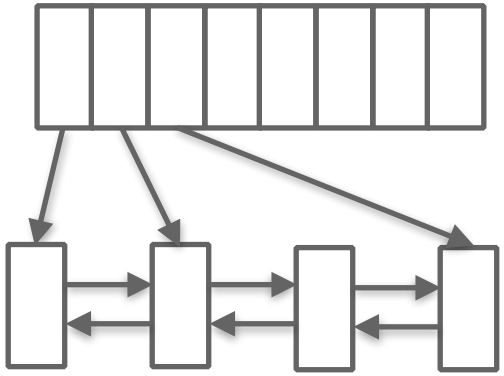
`std::list`



`std::set &`
`std::map`



`std::unordered_set &`
`std::unordered_map`



std::array

Stack

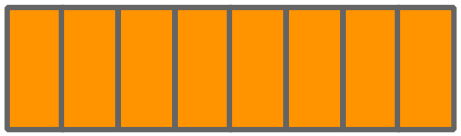
Free Store / Heap

No dynamic memory allocation on the free store/heap

std::array


```
int main()
{
   // Fundamental types
   std::array<int,3> a1;                 // Uninitialized integers!
   std::array<int,3> a2{};               // Properly initialized to ( 0 0 0 )
   std::array<int,3> a3{ 1 };            // Properly initialized to ( 1 0 0 )
   std::array<int,3> a4 = { 1, 2, 3 };   // Properly initialized to ( 1 2 3 )

   // Class/User-defined types
   std::array<std::string,2> a5;     // Two empty strings ( "" "" )
   std::array<std::string,2> a6{};   // Two empty strings ( "" "" )

   // Index-based access
   a6[0] = "Bjarne Stroustrup";
   a6[1] = "Herb Sutter";

   // Iterator-based access
   for( auto iter=a6.begin(); iter!=a6.end(); ++iter ) {
      std::cout << *iter << '\n';
   }
}
```

std::array



Pros:

- STL compliant static array
- Contiguous elements (no additional memory)
- Direct access to all elements (random access)
- Cache efficient

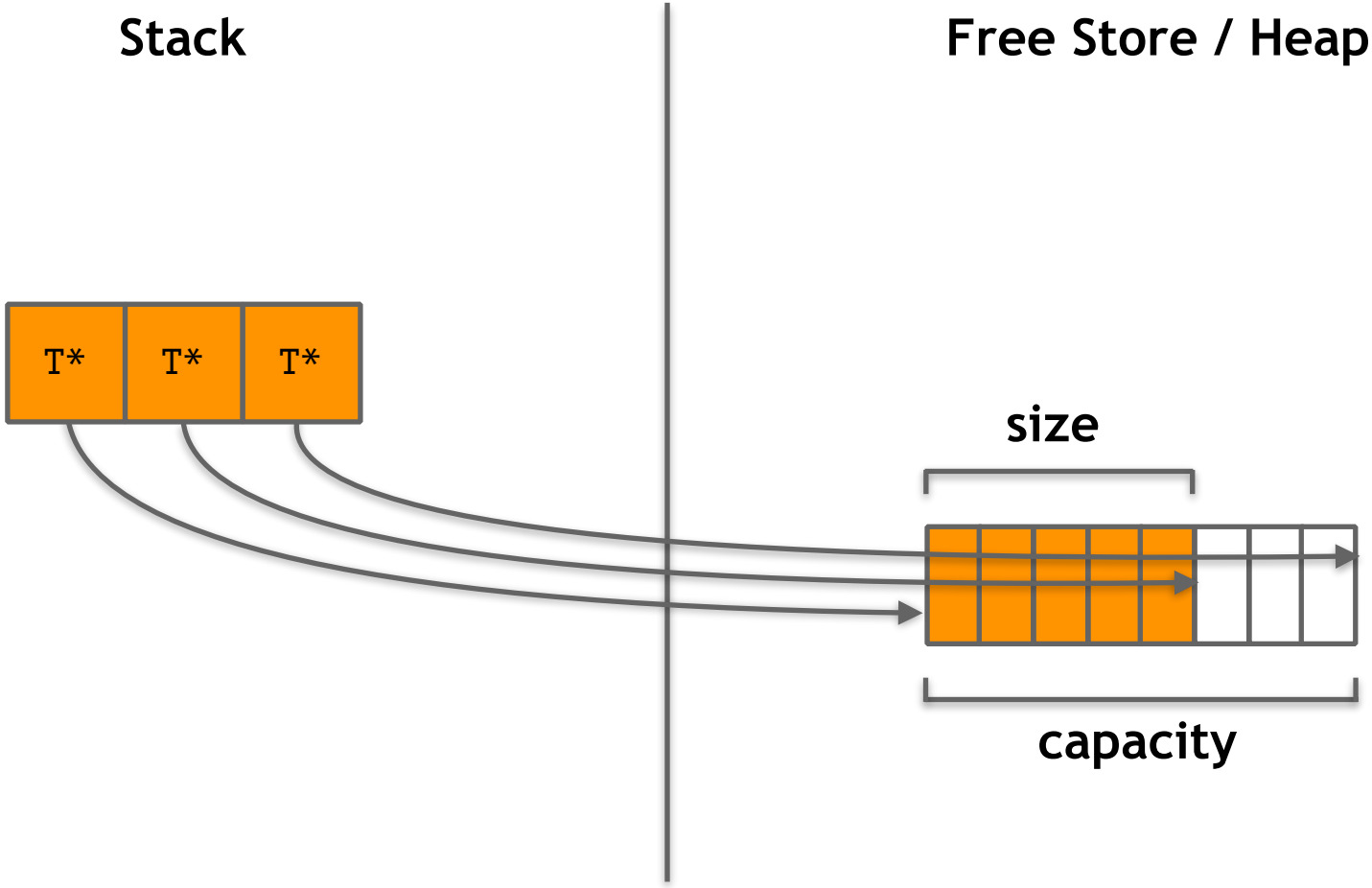
Cons:

- Number of elements fixed at compile time
- No insertion/removal of elements possible

Additional properties:

- Not automatically sorted

std::vector



std::vector

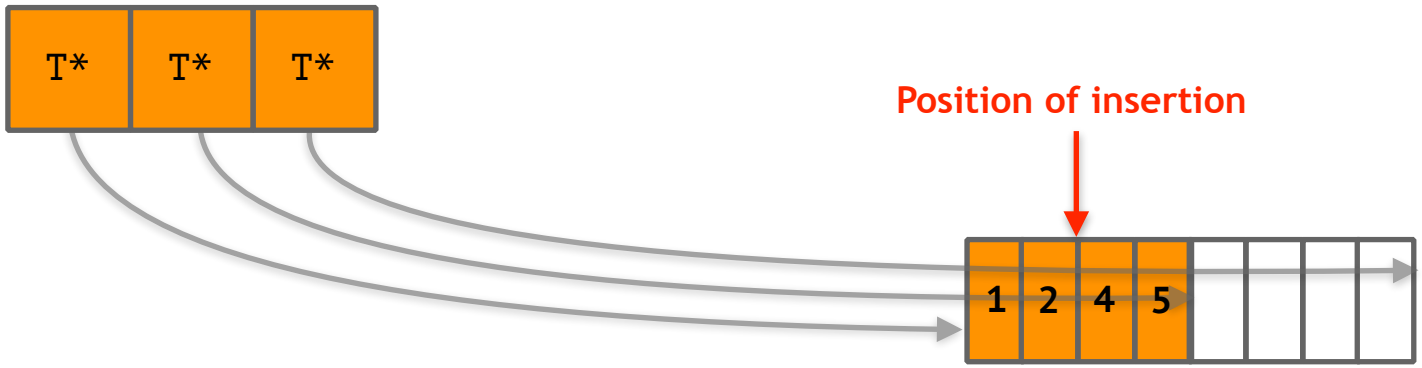
```
int main()
{
  // Constructors
  std::vector<int> v1;           // Empty vector
  std::vector<int> v2( 3, 5 );   // Vector with 3 ints ( 5 5 5 )
  std::vector<int> v3{ 3, 5 };   // Vector with 2 ints ( 3 5 )

  // Element insertion
  v3.push_back( 7 );            // Vector with 3 ints ( 3 5 7 )
  v3.reserve( 5 );              // Extending the capacity to 5
  v3.emplace_back( 9 );         // Vector with 4 ints ( 3 5 7 9 )
  v3.insert( v3.begin(), 1 );   // Vector with 5 ints ( 1 3 5 7 9 )

  // Element removal
  v3.erase( v3.begin() );       // Vector with 4 elements ( 3 5 7 9 )
  v3.erase( v3.begin(), v3.begin()+2 );  // Vector with 2 elements ( 7 9 )

  // Resizing
  v3.resize( 4 );  // Vector with 4 elements ( 7 9 0 0 )

  // ...


}
```

std::vector

```
int main()
{
   // ...

   // Index-based access
   v3[0] = 11;
   v3[1] = 22;

   // Iterator-based access
   for( auto iter=v3.begin(); iter!=v3.end(); ++iter ) {
      std::cout << *iter << '\n';
   }

}
```

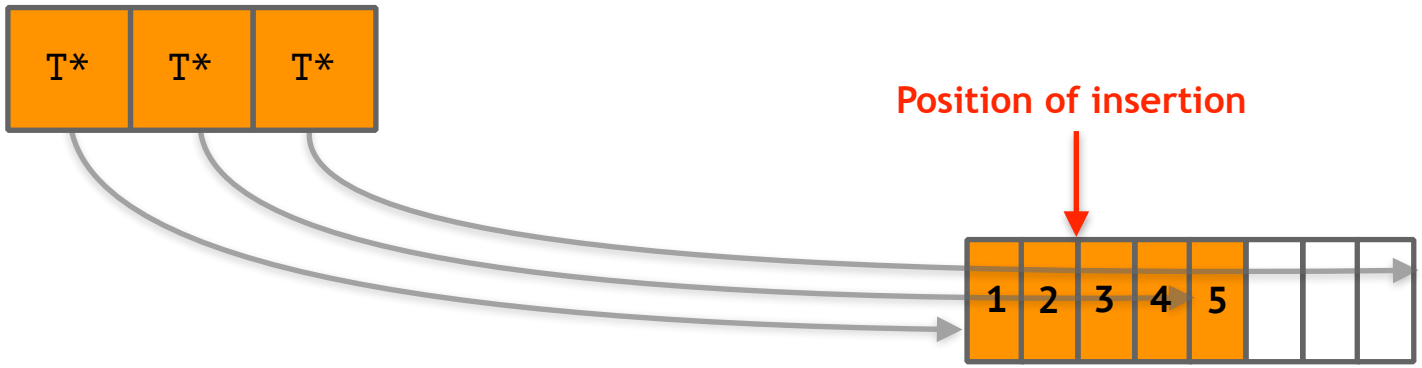
Inserting into a std::vector

```
int main()
{
   std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

   v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

   // ...
}
```



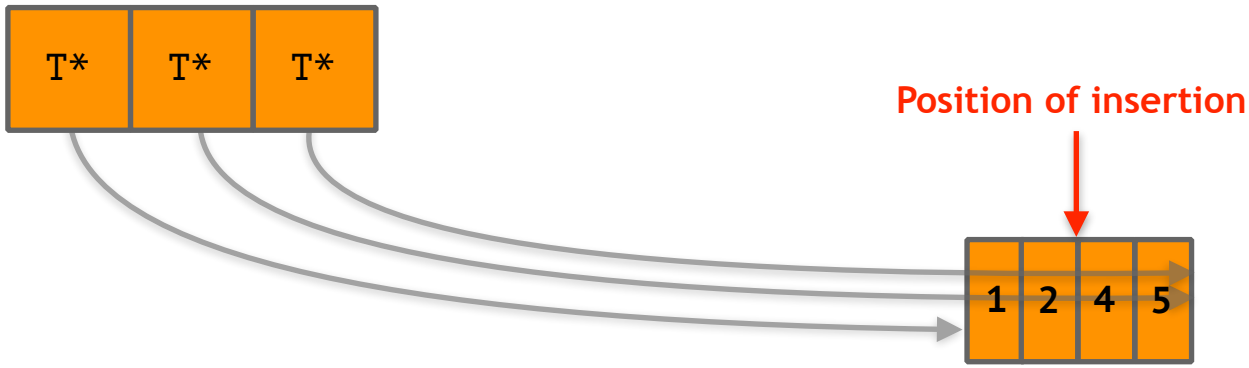
Inserting into a std::vector

```
int main()
{
   std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

   v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

   // ...
}
```



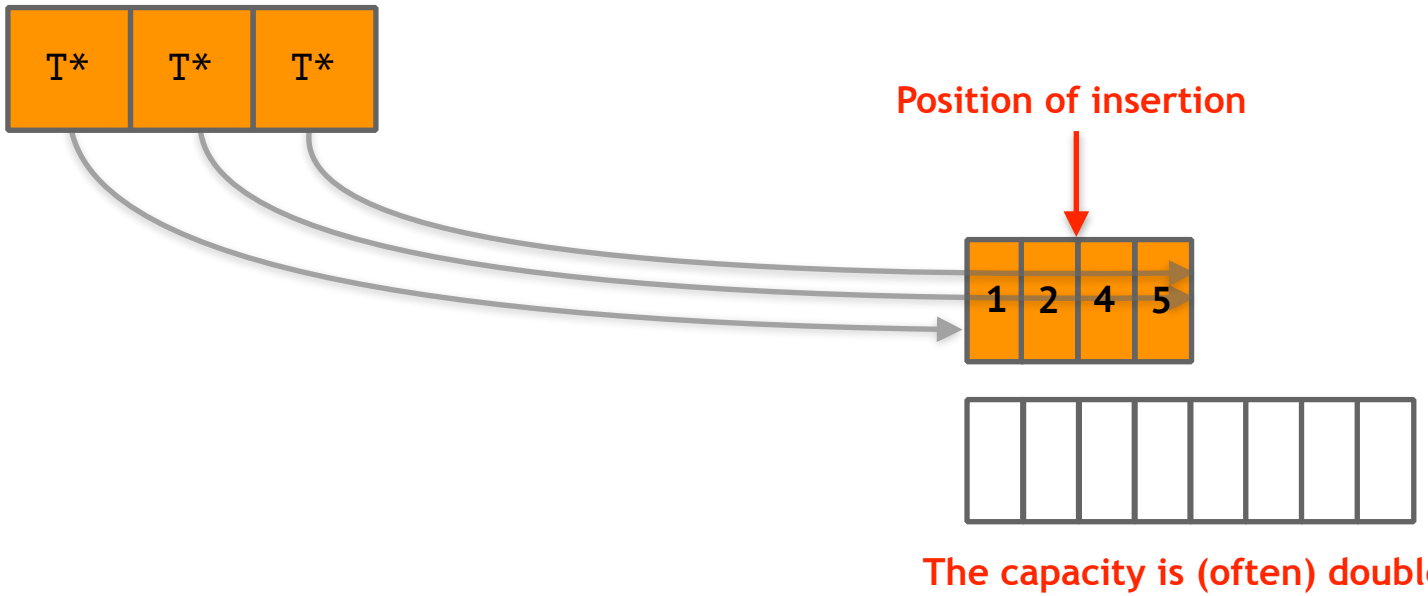
Inserting into a std::vector

```
int main()
{
   std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

   v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

   // ...
}
```



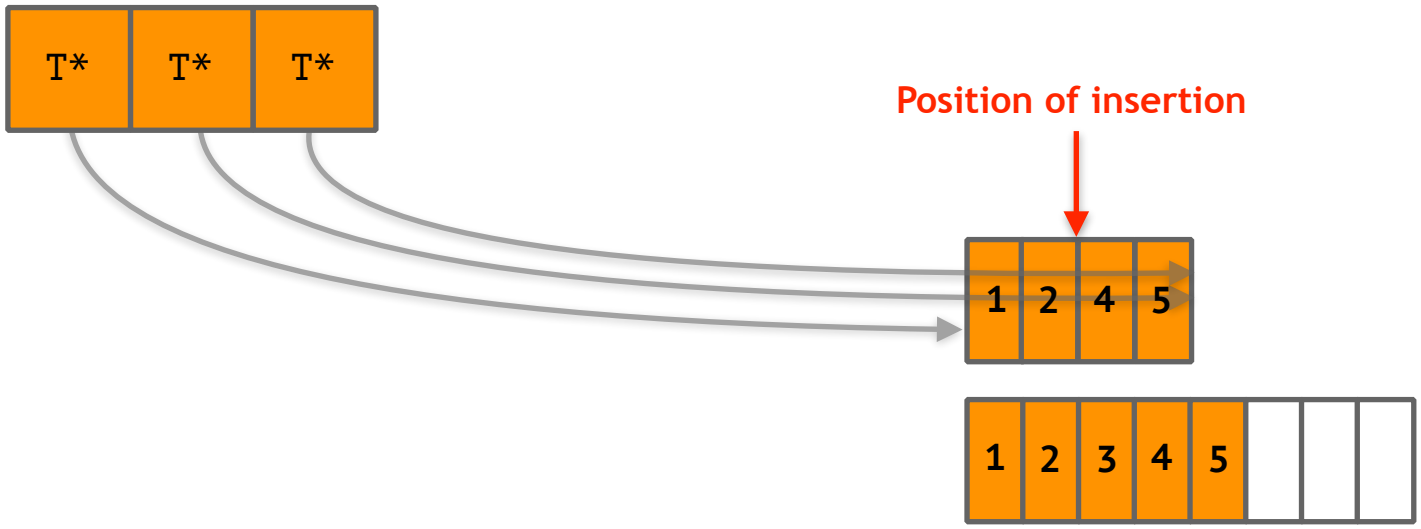
Inserting into a std::vector

```
int main()
{
   std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

   v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

   // ...
}
```



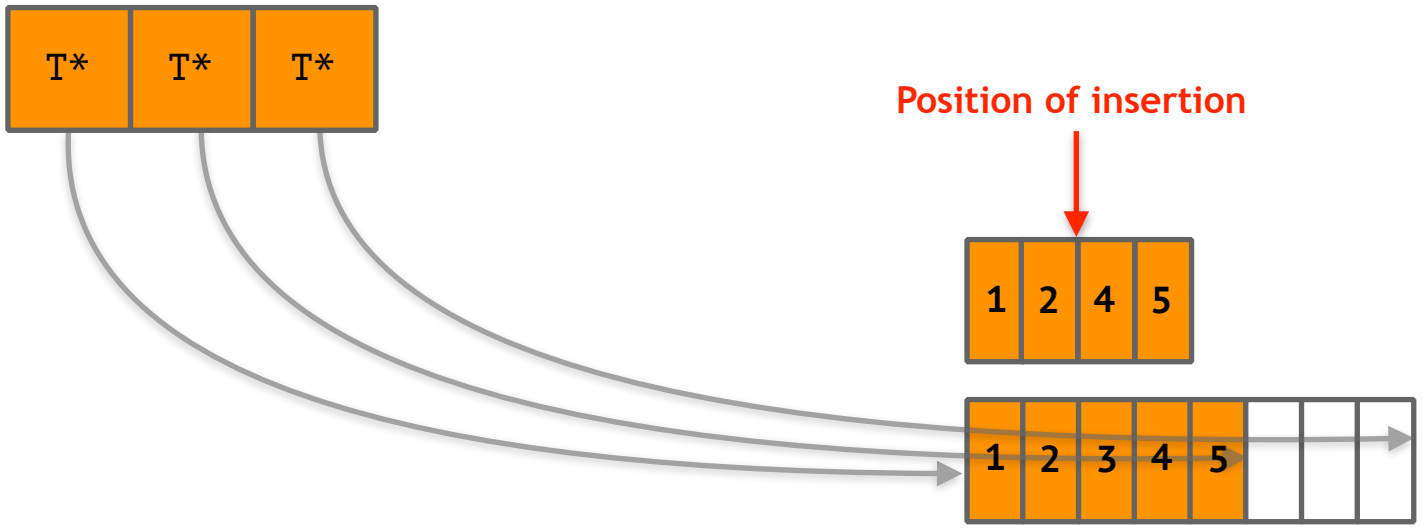
Inserting into a std::vector

```
int main()
{
   std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

   v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

   // ...
}
```



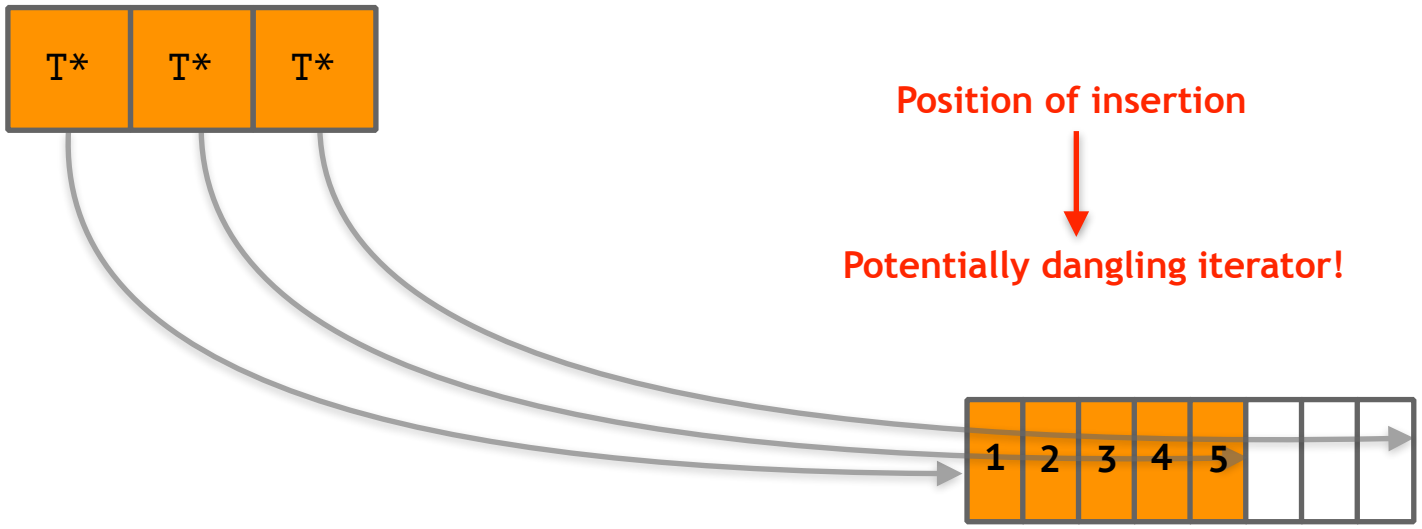
Inserting into a std::vector

```
int main()
{
   std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

   v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

   // ...
}
```

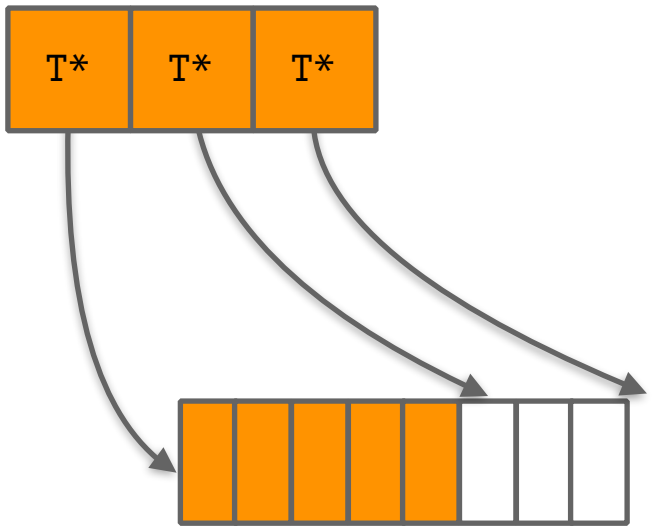


Inserting into a std::vector

```
int main()
{
  std::vector<int> v{ 1, 2, 4, 5 };  // Vector with 4 ints ( 1 2 4 5 )

  v.insert( v.begin()+2, 3 );        // Vector with 5 ints ( 1 2 3 4 5 )

  // ...
}
```



std::vector



Pros:

- Managed dynamic array
- Contiguous elements (no additional memory)
- Direct access to all elements (random access)
- Cache efficient

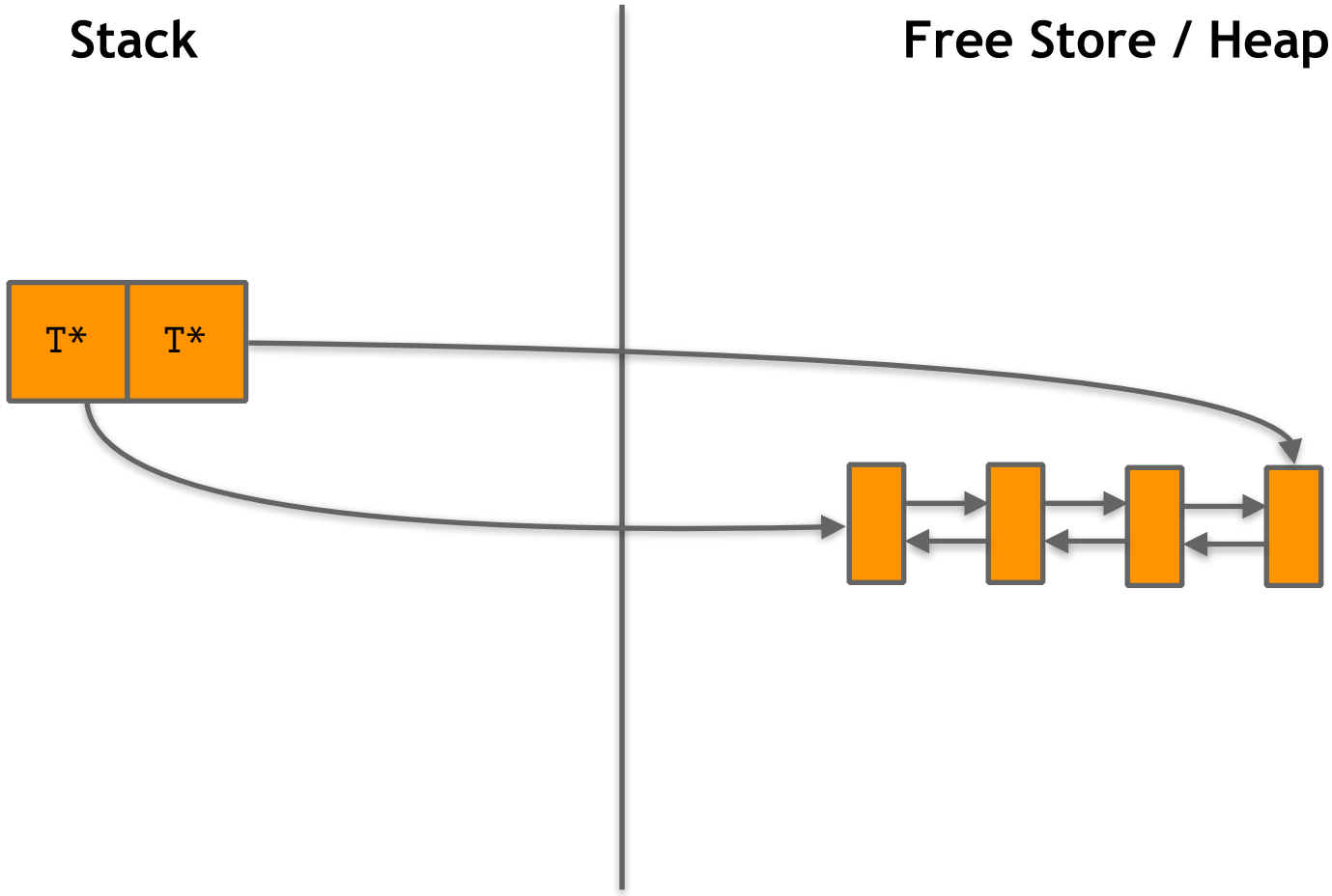
Cons:

- Inserting elements at the front or in the middle shifts elements
- Reallocation requires transfer of elements
- Invalidation of iterators
- `std::vector<bool>`

Additional properties:

- Not automatically sorted
- Never shrinks automatically

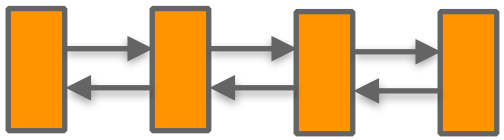
std::list



std::list

```
int main()
{
   // Constructors
   std::list<int> l1;           // Empty list
   std::list<int> l2( 3, 5 );   // List with 3 ints ( 5 5 5 )
   std::list<int> l3{ 3, 5 };   // List with 2 ints ( 3 5 )

   // Element insertion
   l3.push_back( 7 );            // List with 3 ints ( 3 5 7 )
   // No reserve operation, only resize!
   l3.emplace_back( 9 );         // List with 4 ints ( 3 5 7 9 )
   l3.insert( l3.begin(), 1 );   // List with 5 ints ( 1 3 5 7 9 )

   // Element removal
   l3.erase( l3.begin() );  // List with 4 elements ( 3 5 7 9 )
   l3.erase( l3.begin(), std::next(l3.begin(),2) );  // List with ( 7 9 )

   // No index-based access

   // Iterator-based access
   for( auto iter=l3.begin(); iter!=l3.end(); ++iter ) {
      std::cout << *iter << '\n';
   }
}
```

std::list



Pros:

- Independently allocated nodes
- Stable nodes (no iterator invalidation)
- Efficient insertion at any position (?)

Cons:

- One memory allocation per node
- No direct (index) access to elements (O(N))
- Memory overhead (2 additional pointers)
- Not cache efficient

Additional Properties:

- Not automatically sorted

std::deque

std::deque

```
int main()
{
  // Constructors
   std::deque<int> d1;           // Empty deque
   std::deque<int> d2( 3, 5 );   // Deque with 3 ints ( 5 5 5 )
   std::deque<int> d3{ 3, 5 };   // Deque with 2 ints ( 3 5 )

   // Element insertion
   d3.push_back( 7 );            // Deque with 3 ints ( 3 5 7 )
   // No reserve operation!
   d3.emplace_back( 9 );         // Deque with 4 ints ( 3 5 7 9 )
   d3.insert( d3.begin(), 1 );   // Deque with 5 ints ( 1 3 5 7 9 )

   // Element removal
   d3.erase( d3.begin() );  // Deque with 4 elements ( 3 5 7 9 )
   d3.erase( d3.begin(), d3.begin()+2 );  // Deque with 2 elements ( 7 9 )

   // Resizing
   d3.resize( 4 );  // Deque with 4 elements ( 7 9 0 0 )

   // …


}
```

std::deque

```
int main()
{
   // ...

   // Index-based access
   d3[0] = 11;
   d3[1] = 22;

   // Iterator-based access
   for( auto iter=d3.begin(); iter!=d3.end(); ++iter ) {
      std::cout << *iter << '\n';
   }

}
```

std::deque

Pros:

- Concatenation of arrays of page size
- Mostly contiguous elements (little overhead)
- Direct access to all elements (random access)
- Efficient insertion at the back and front
- Cache efficient
- Shrinking possible

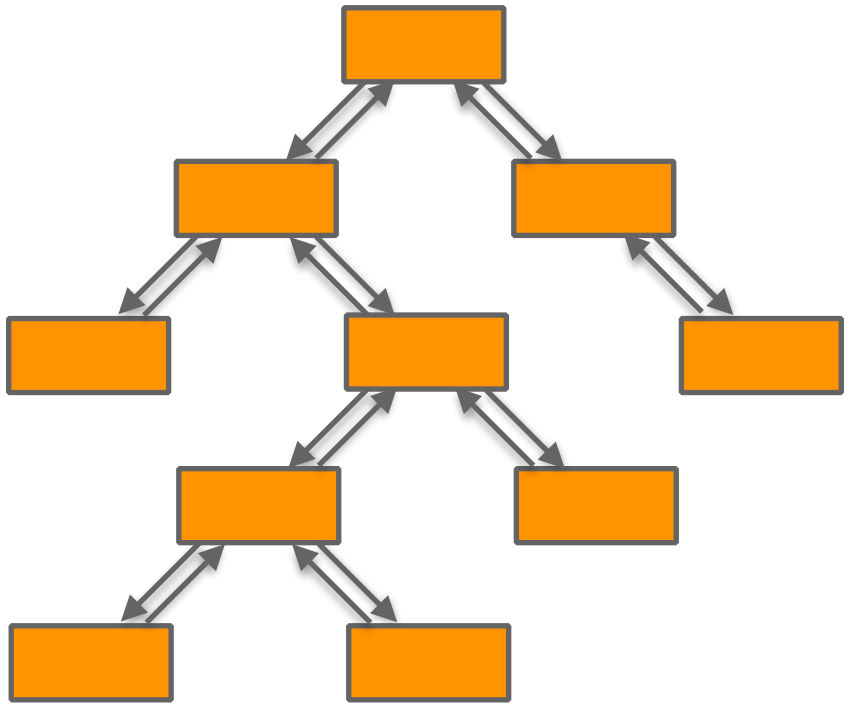
Cons:

- Insertion in the middle causes shifting

Additional Properties:

- Not automatically sorted

std::set / std::map



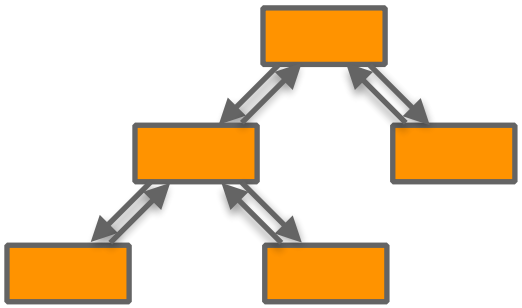
std::set / std::map

```
int main()
{
  // Constructors
  std::set<int> s1;           // Empty set
  std::set<int> s2{ 7, 3 };   // Set with 2 ints ( 3 7 )

  // Set with 3 ints ( 3 5 7 ); success1 = true
  auto [iter1,success1] = s2.insert( 5 );
  assert( s2.size() == 3 );

  // Set with 3 ints ( 3 5 7 ); success2 = false
  auto [iter2,success2] = s2.insert( 3 );
  assert( s2.size() == 3 );

  // Set with 4 ints ( 3 5 7 9 ); success3 = true
  auto [iter3,success3] = s2.emplace( 9 );
  assert( s2.size() == 4 );

  // Set with 4 ints ( 3 5 7 9 ); success4 = false
  auto [iter4,success4] = s2.emplace( 7 );
  assert( s2.size() == 4 );

  // ...
}
```

std::set / std::map

```
int main()
{
   // ...

   // Element removal with single iterator
   s2.erase( s2.begin() );  // Set with 3 elements ( 3 5 7 )
   assert( s2.size() == 3 );

   // Element removal with single iterator
   s2.erase( s2.begin(), std::next(s2.begin(),2) );  // Only ( 9 )
   assert( s2.size() == 1 );

   // Element removal with key
   s2.erase( 9 );  // Set with 0 element ( )
   assert( s2.size() == 0 );

   // No index-based access

   // Iterator-based access
   for( auto iter=s2.begin(); iter!=s2.end(); ++iter ) {
      std::cout << *iter << '\n';
   }
}
```

std::set / std::map



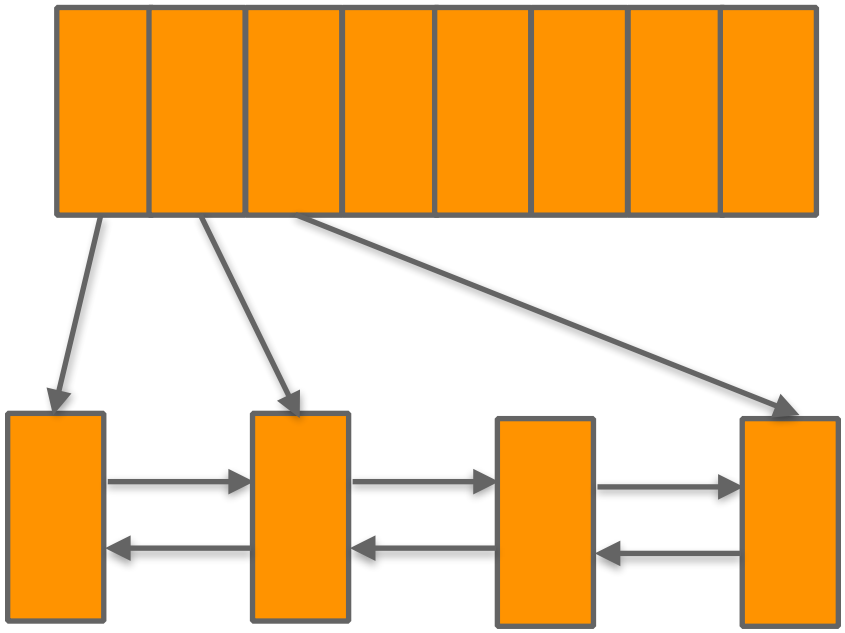
Pros:

- Independently allocated nodes
- Stable nodes (no iterator invalidation)
- Always sorted (fast lookup speed)

Cons:

- One memory allocation per node
- No direct (index) access to elements
- Memory overhead (3 additional pointers)
- Complex element traversal
- Not cache efficient

std::unordered_set / std::unordered_map



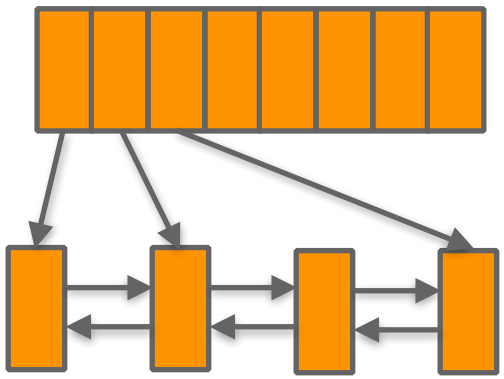
std::unordered_set / std::unordered_map

```
int main()
{
   // Constructors
   std::unordered_set<int> s1;           // Empty set
   std::unordered_set<int> s2{ 7, 3 };   // Set with 2 ints ( 3 7 )

   // Set with 3 ints ( 3 5 7 ); success1 = true
   auto [iter1,success1] = s2.insert( 5 );
   assert( s2.size() == 3 );

   // Set with 3 ints ( 3 5 7 ); success2 = false
   auto [iter2,success2] = s2.insert( 3 );
   assert( s2.size() == 3 );

   // Set with 4 ints ( 3 5 7 9 ); success3 = true
   auto [iter3,success3] = s2.emplace( 9 );
   assert( s2.size() == 4 );

   // Set with 4 ints ( 3 5 7 9 ); success4 = false
   auto [iter4,success4] = s2.emplace( 7 );
   assert( s2.size() == 4 );

   // ...
}
```

std::unordered_set / std::unordered_map

```
int main()
{
   // ...

   // Element removal with single iterator
   s2.erase( s2.begin() );  // Set with 3 elements ( 3 5 7 )
   assert( s2.size() == 3 );

   // Element removal with single iterator
   s2.erase( s2.begin(), std::next(s2.begin(),2) );  // Only ( 9 )
   assert( s2.size() == 1 );

   // Element removal with key
   s2.erase( 9 );  // Set with 0 element ( )
   assert( s2.size() >= 0 );

   // No index-based access

   // Iterator-based access
   for( auto iter=s2.begin(); iter!=s2.end(); ++iter ) {
      std::cout << *iter << '\n';
   }
}
```

std::unordered_set / std::unordered_map



Pros:

- Independently allocated nodes
- Stable nodes (no iterator invalidation)
- Fast lookup speed due to hashing

Cons:

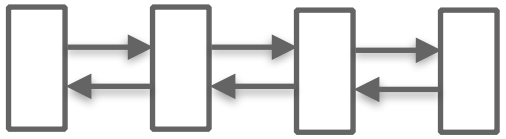
- Never sorted
- Lookup speed depending on hashing
- One memory allocation per node
- No direct (index) access to elements
- Memory overhead (2 additional pointers)
- Memory overhead to minimize collisions
- Not cache efficient

The Standard Containers

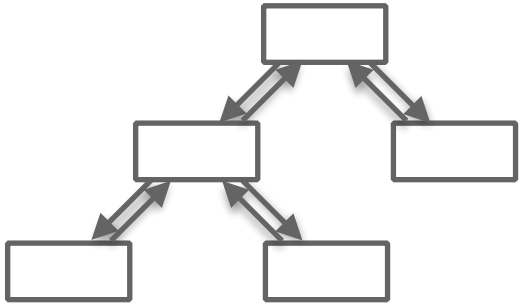
`std::vector &`
`std::array`

`std::deque`

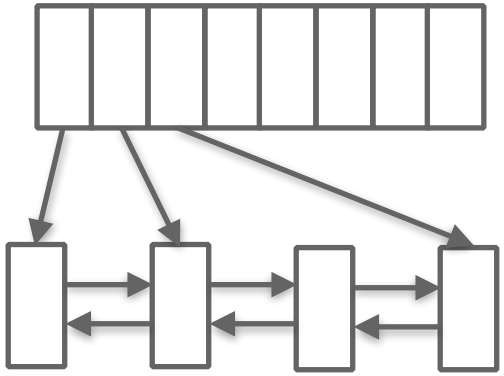
`std::list`



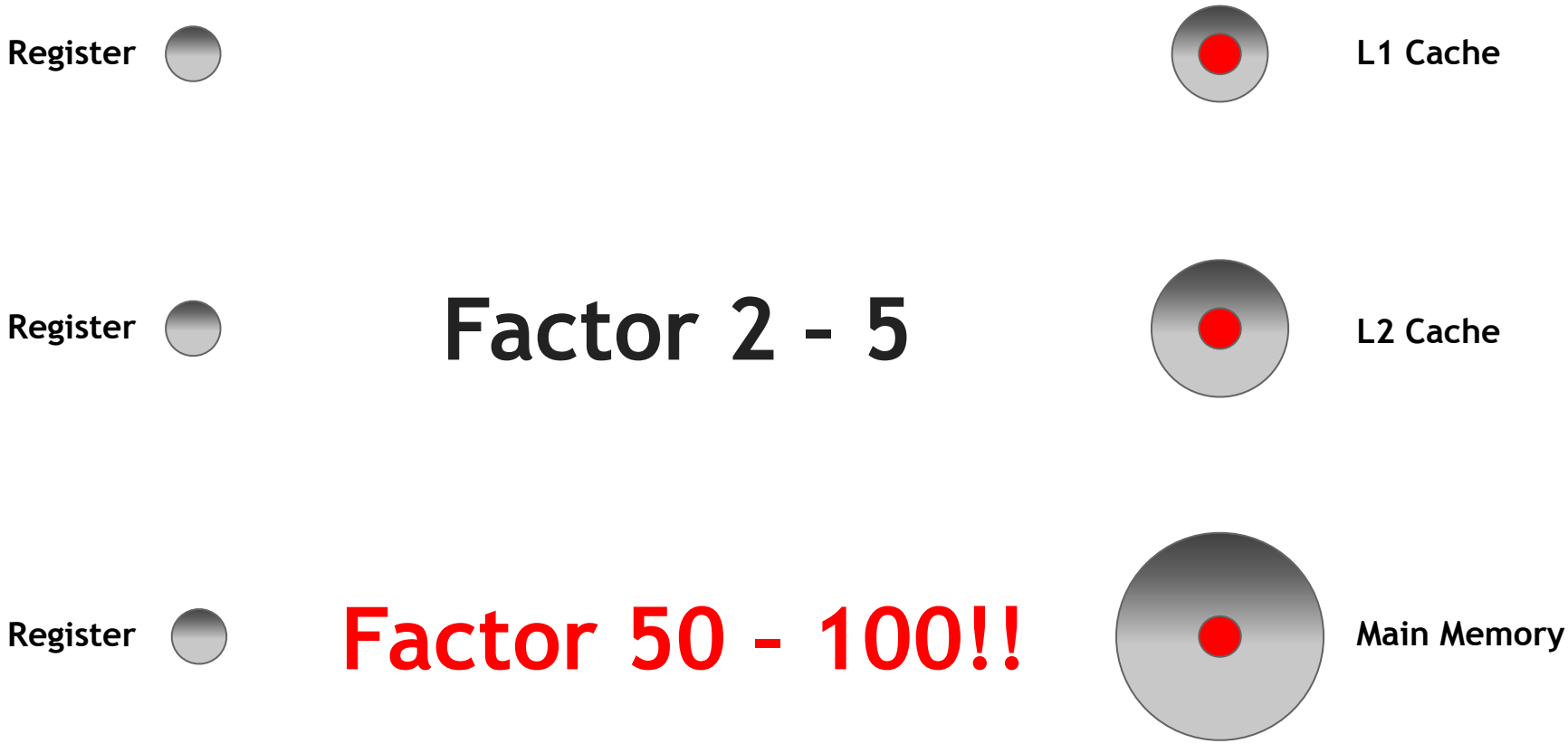
`std::set &`
`std::map`

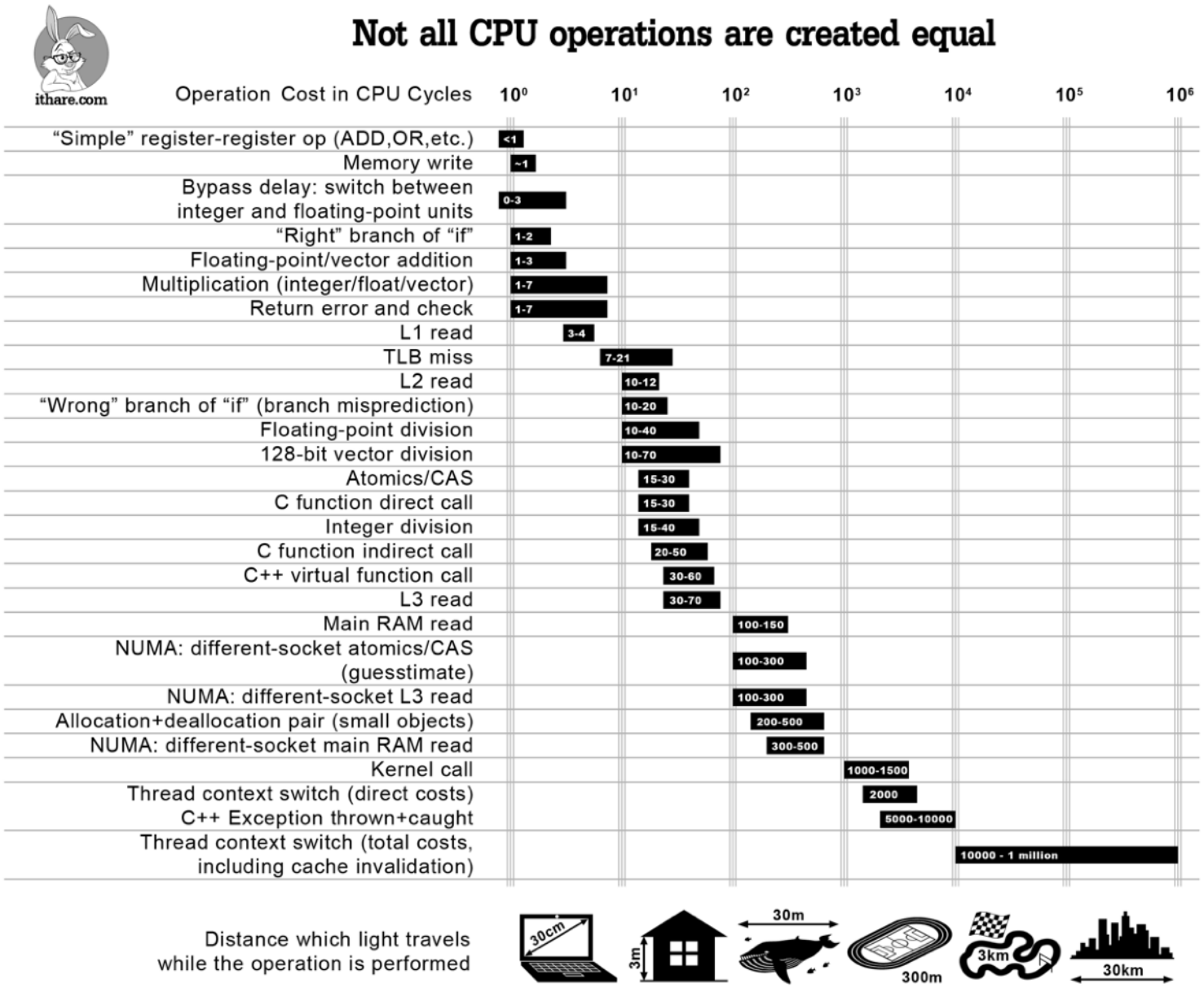


`std::unordered_set &`
`std::unordered_map`



Caches and Why You Care





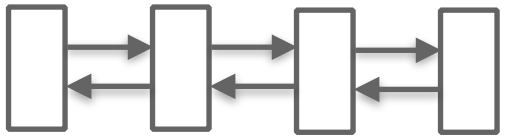
http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

The Standard Containers

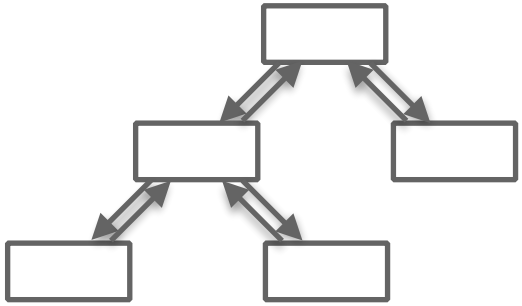
`std::vector &`
`std::array`

`std::deque`

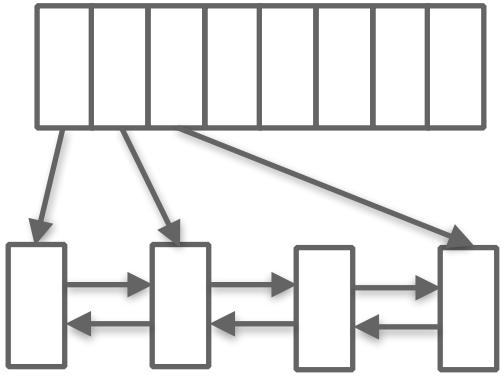
`std::list`



`std::set &`
`std::map`



`std::unordered_set &`
`std::unordered_map`



Performance of Containers

Let's measure the performance of the standard containers for the following use cases:

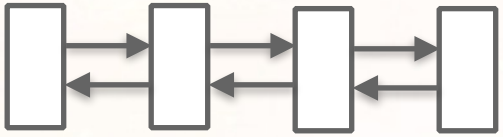
- Traversal
- Adding an Element to the End
- Finding an Arbitrary Element
- Building a Sorted Collection
- Removing an Arbitrary Element
- Removing the First Element

Performance - Traversal

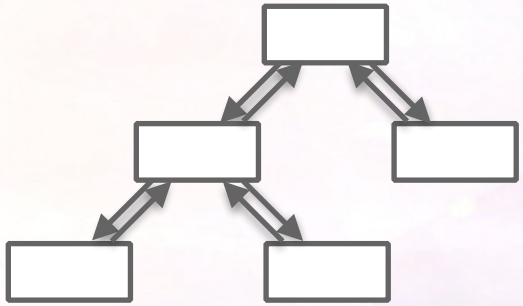
std::vector

std::deque

std::list



std::set



Traversal: Which container is fastest?

A	**std::vector**	B	**std::deque**
C	**std::list**	D	**std::set**

Programming Task

Task (2_STL_Algorithms/Containers/Traversal): Copy-and-paste the following code into quick-bench.com. Benchmark the time to traverse a standard container.

2. STL Algorithms – STL Containers

Quick C++ Benchmark

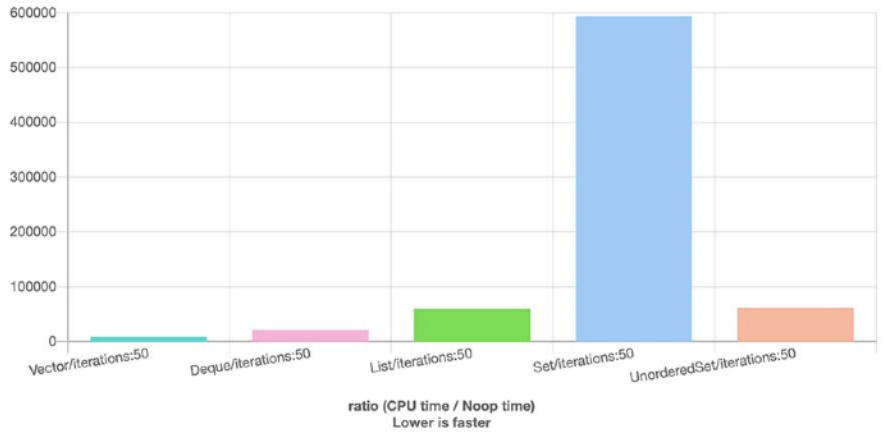
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Traversal.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to t
*       a standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-----------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

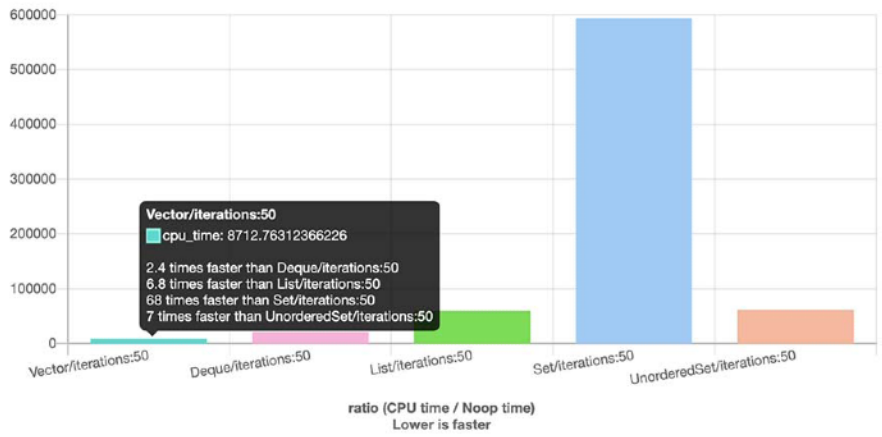
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*************************************************************************************
*
* \file Traversal.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to t
*       a standard container.
*
*************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

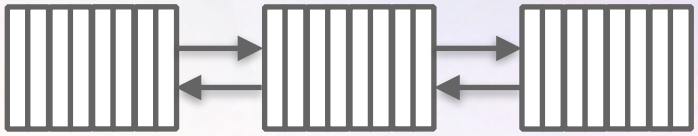
Charts | Assembly



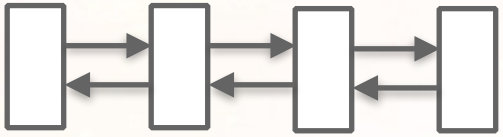
Show Noop bar

std::vector

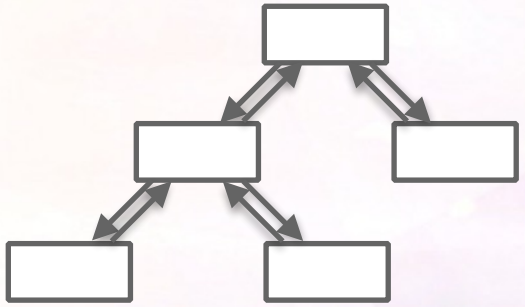
std::deque



std::list



std::set



Traversal: Which container is fastest?

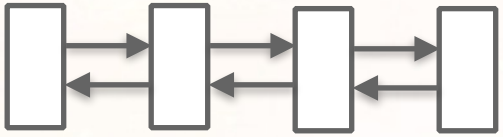
A	std::vector	B	std::deque
C	std::list	D	std::set

Performance - Adding an Element to the End

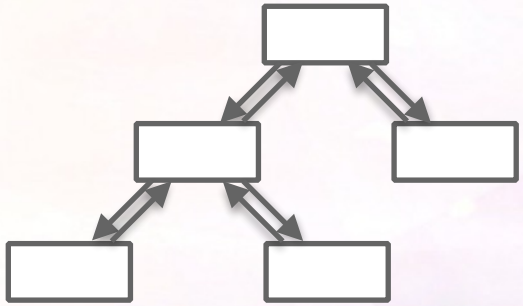
std::vector

std::deque

std::list



std::set



push_back(): Which container is fastest?

A	std::vector	B	std::deque
C	std::list	D	std::set

Programming Task

Task (2_STL_Algorithms/Containers/PushBack): Copy-and-paste the following code into quick-bench.com. Benchmark the time to add elements to the end of a standard container.

2. STL Algorithms – STL Containers

Quick C++ Benchmark

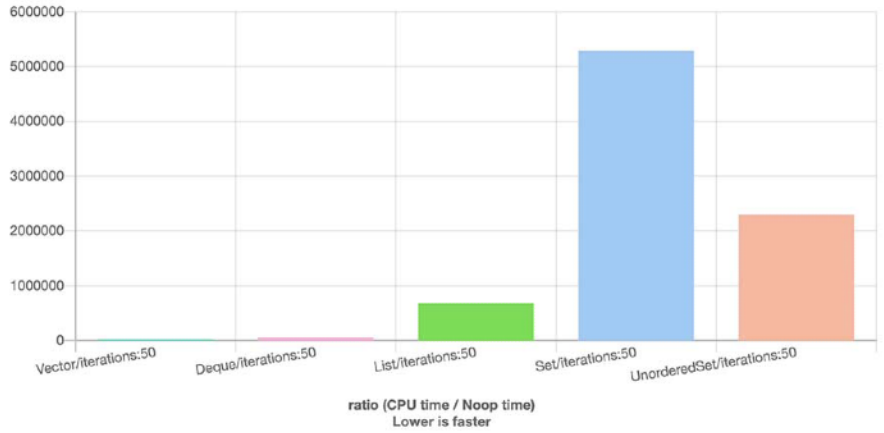
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file PushBack.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to a
*       elements to the end of a standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

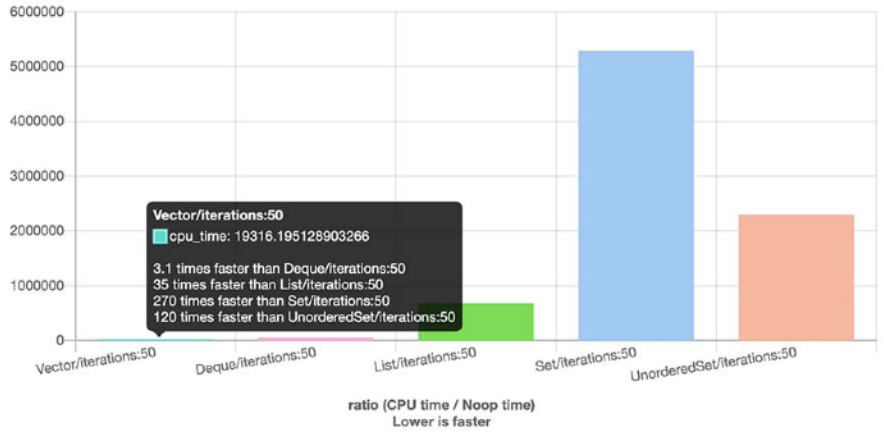
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file PushBack.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to a
*       elements to the end of a standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

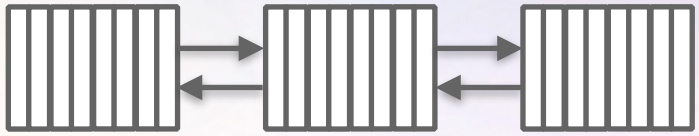
Charts | Assembly



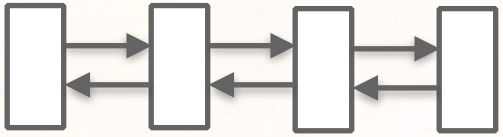
Show Noop bar

std::vector

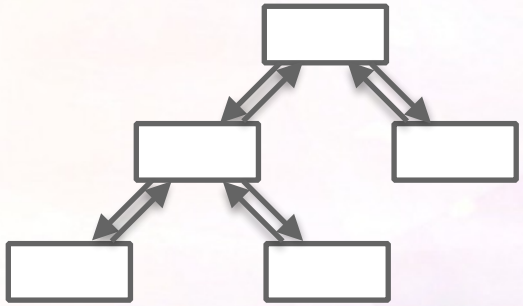
std::deque



std::list



std::set



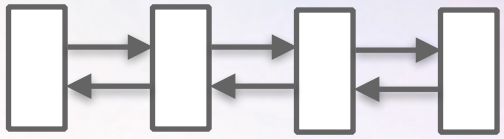
push_back(): Which container is fastest?

A	std::vector	B	std::deque
C	std::list	D	std::set

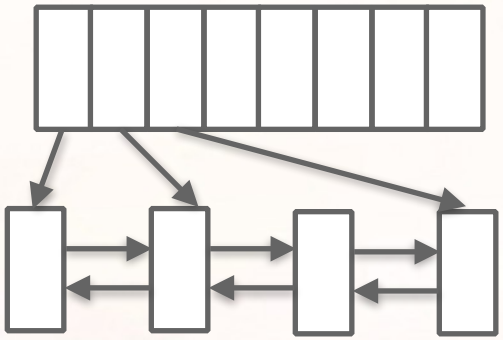
Performance - Finding an Arbitrary Element

std::vector

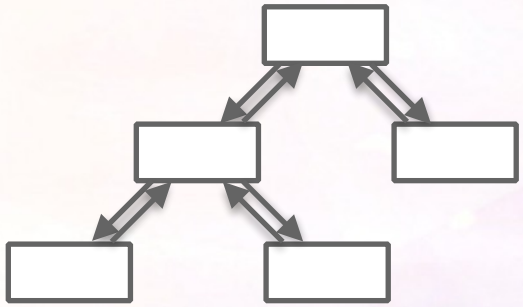
std::list



std::unordered_set



std::set



Finding an arbitrary element: Which container is fastest?

A	**std::vector**	B	**std::list**
C	**std::unordered_set**	D	**std::set**

Programming Task

Task (2_STL_Algorithms/Containers/Lookup): Copy-and-paste the following code into quick-bench.com. Benchmark the time to search an arbitrary element in a standard container.

2. STL Algorithms – STL Containers

Quick C++ Benchmark

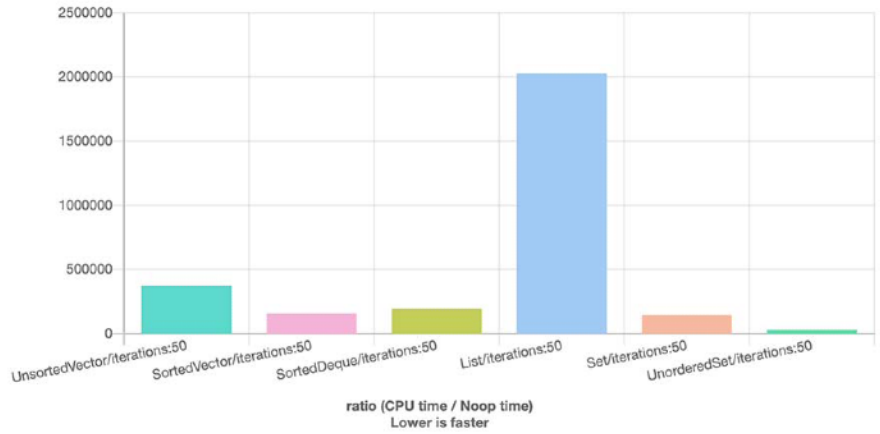
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Lookup.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to s
*       an arbitrary element in a standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------

constexpr size_t size( 1000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             1
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-----------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

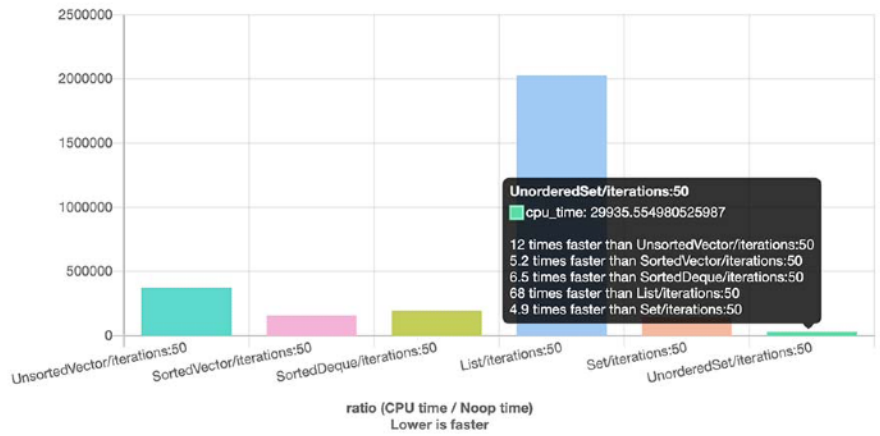
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*************************************************************************************************
*
* \file Lookup.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to s
*       an arbitrary element in a standard container.
*
*************************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------------

constexpr size_t size( 1000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             1
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-----------------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

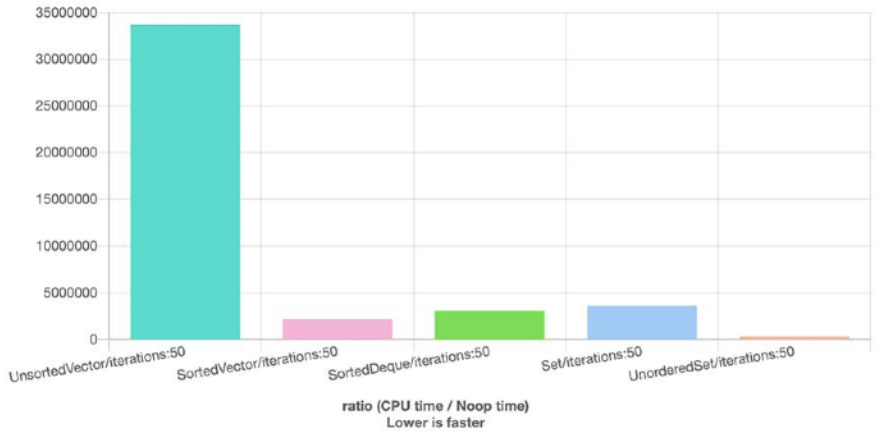
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Lookup.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to s
*       an arbitrary element in a standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             0
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-----------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

Showing cached results

2. STL Algorithms – STL Containers

Quick C++ Benchmark

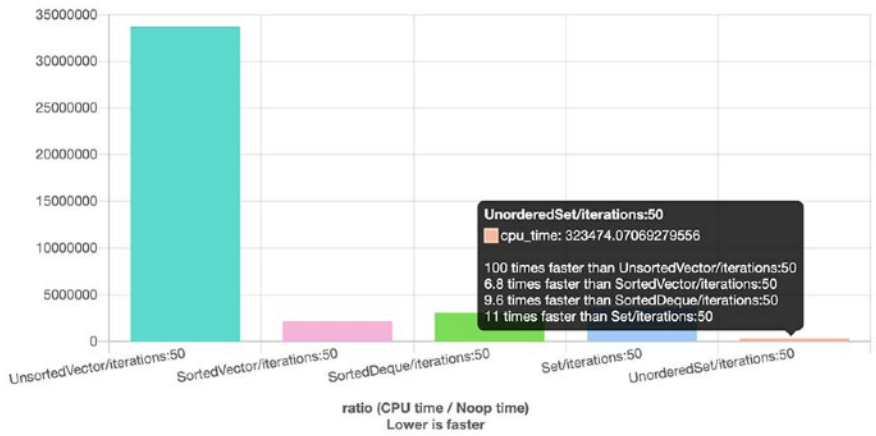
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Lookup.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to s
*       an arbitrary element in a standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             0
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-----------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly

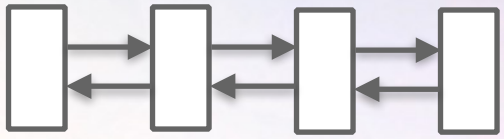


Show Noop bar

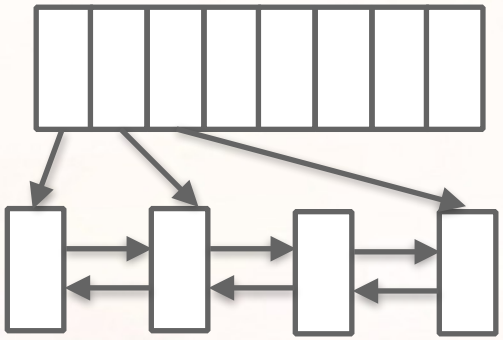
Showing cached results

std::vector

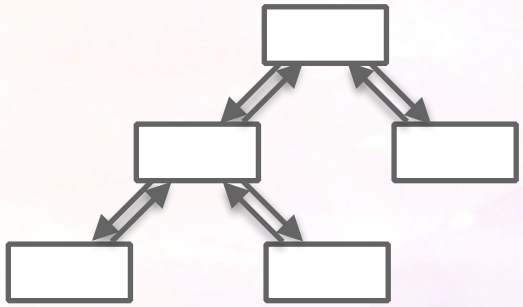
std::list



std::unordered_set



std::set



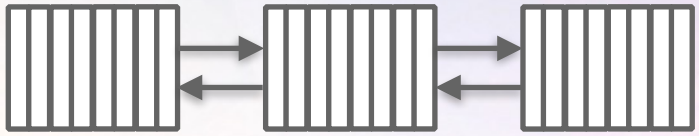
Finding an arbitrary element: Which container is fastest?

A	**std::vector**	B	**std::list**
C	**std::unordered_set**	D	**std::set**

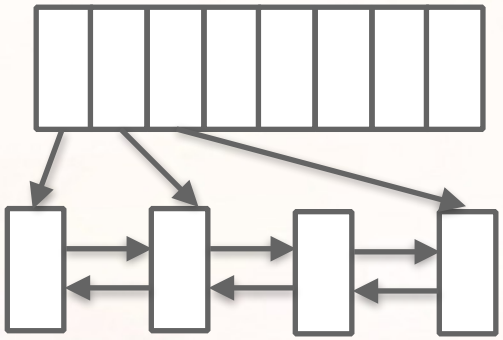
Performance - Building a Sorted Collection

std::vector

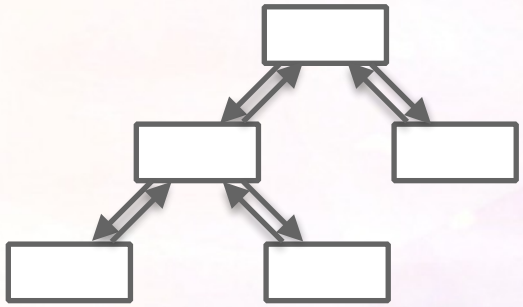
std::deque



std::unordered_set



std::set



Building a sorted collection: Which container is fastest?

A	**std::vector**	B	**std::deque**
C	**std::unordered_set**	D	**std::set**

Programming Task

Task (2_STL_Algorithms/Containers/Sort): Copy-and-paste the following code into quick-bench.com. Benchmark the time to build a sorted standard container.

2. STL Algorithms – STL Containers

Quick C++ Benchmark

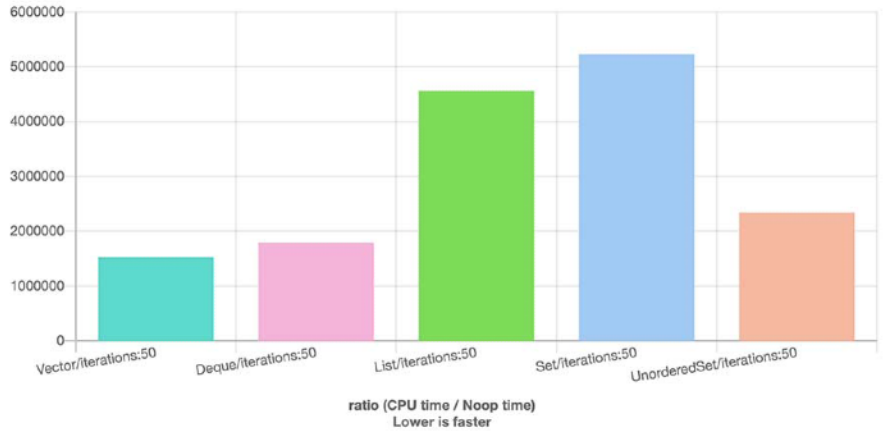
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Sort.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to b
*       a sorted standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-----------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

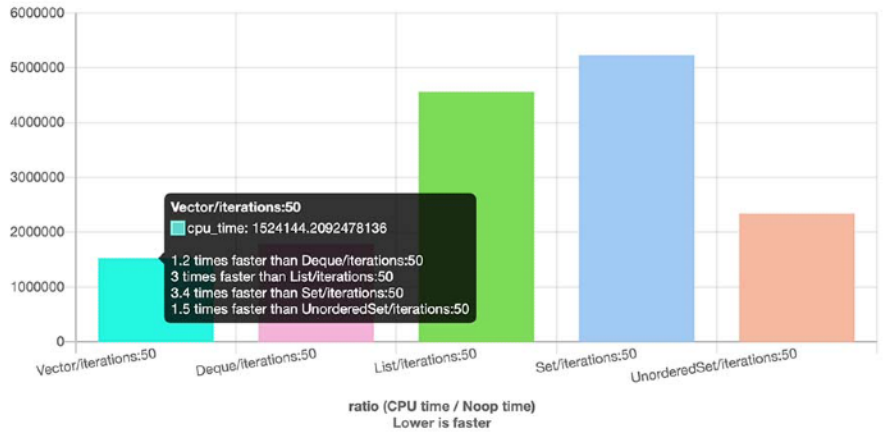
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file Sort.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to
*       a sorted standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

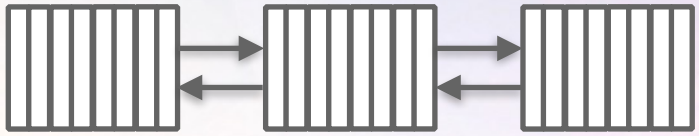
Charts | Assembly



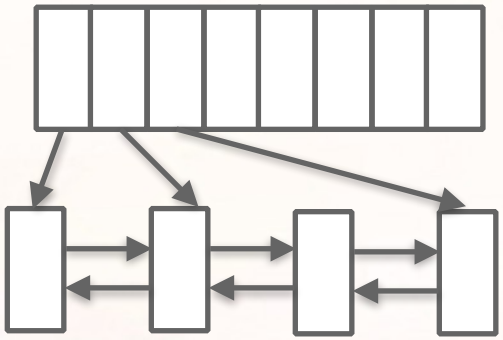
Show Noop bar

std::vector

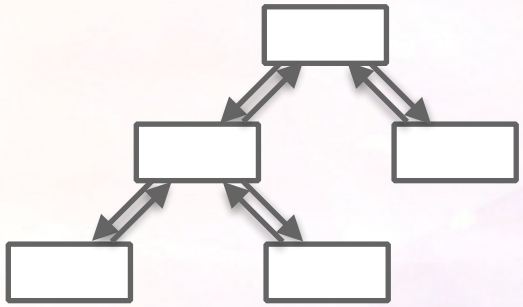
std::deque



std::unordered_set



std::set



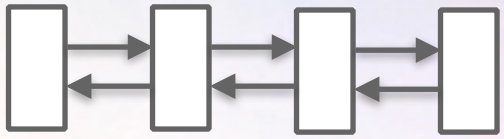
Building a sorted collection: Which container is fastest?

A	**std::vector**	B	**std::deque**
C	**std::unordered_set**	D	**std::set**

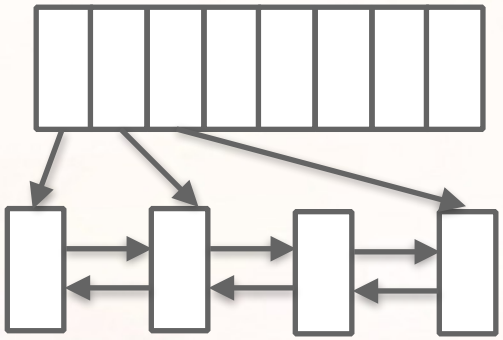
Performance - Removing an Arbitrary Element

std::vector

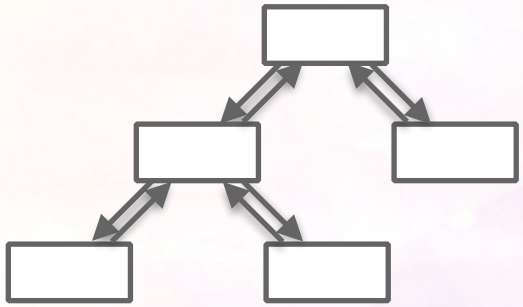
std::list



std::unordered_set



std::set



Removing an arbitrary element: Which container is fastest?

A	**std::vector**	B	**std::list**
C	**std::unordered_set**	D	**std::set**

Programming Task

Task (2_STL_Algorithms/Containers/Remove): Copy-and-paste the following code into quick-bench.com. Benchmark the time to remove an arbitrary element from a standard container.

2. STL Algorithms – STL Containers

Quick C++ Benchmark

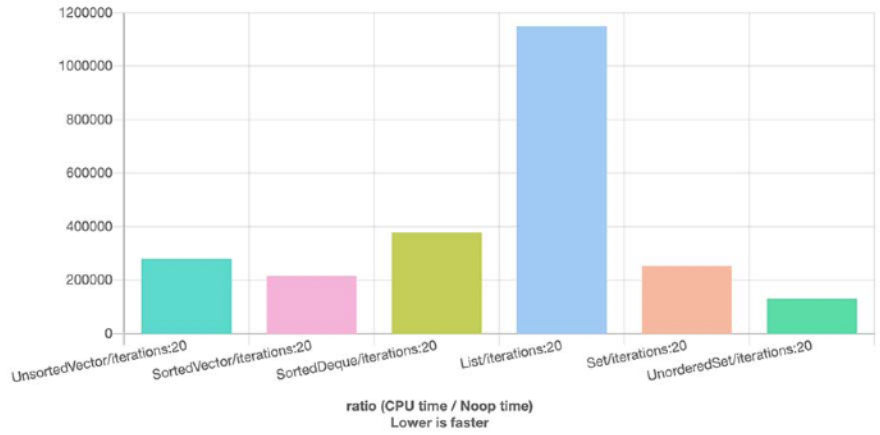
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file Remove.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to r
*       an arbitrary element from a standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 1000 );  // Size of the generated container
constexpr size_t iterations( 20 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             1
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

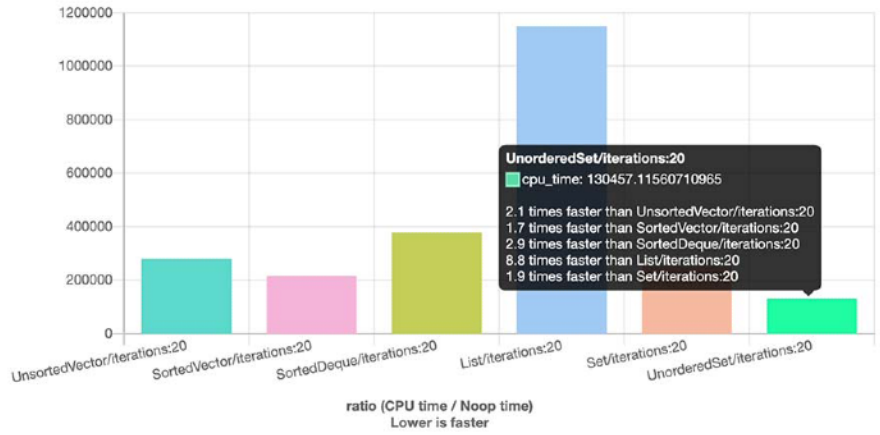
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file Remove.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to r
*       an arbitrary element from a standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 1000 );  // Size of the generated container
constexpr size_t iterations( 20 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             1
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

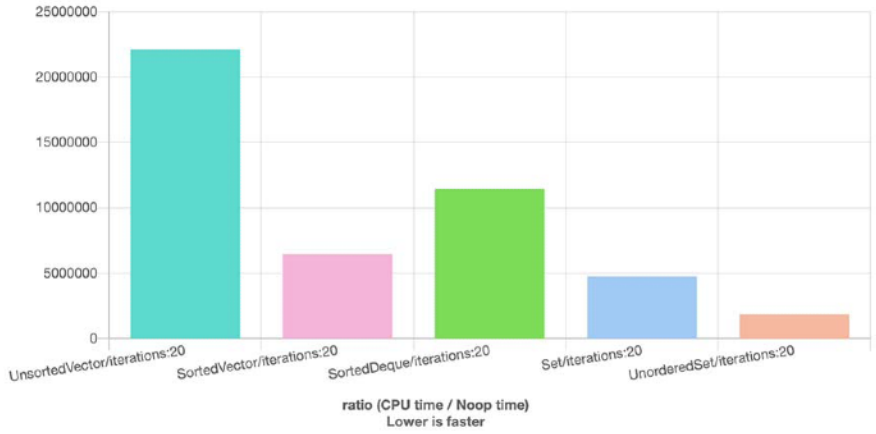
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Remove.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to r
*       an arbitrary element from a standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration---------------------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 20 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             0
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-----------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

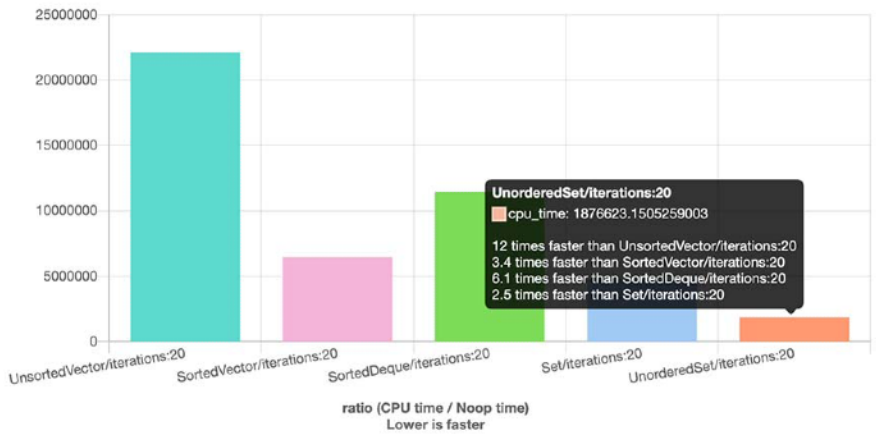
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************************
*
* \file Remove.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to r
*       an arbitrary element from a standard container.
*
*****************************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 20 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_UNSORTED_VECTOR  1
#define BENCHMARK_SORTED_VECTOR    1
#define BENCHMARK_UNSORTED_DEQUE   0
#define BENCHMARK_SORTED_DEQUE     1
#define BENCHMARK_LIST             0
#define BENCHMARK_SET              1
#define BENCHMARK_UNORDERED_SET    1


//--Benchmark setup-------------------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

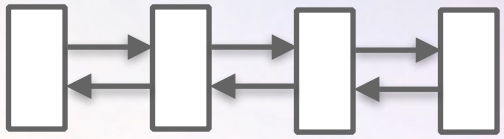
Charts | Assembly



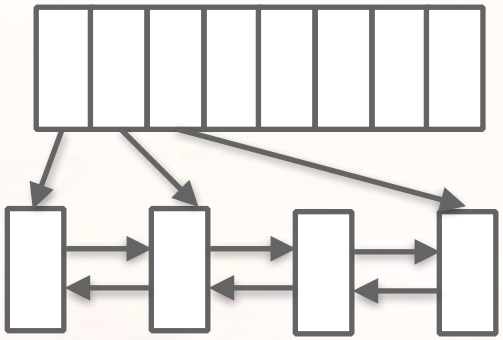
Show Noop bar

std::vector

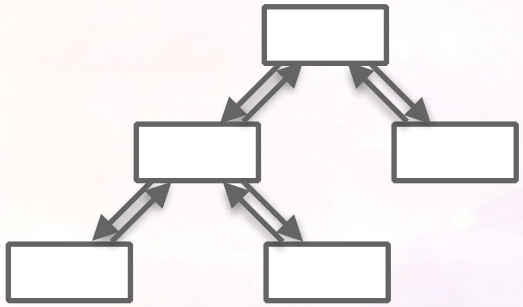
std::list



std::unordered_set



std::set



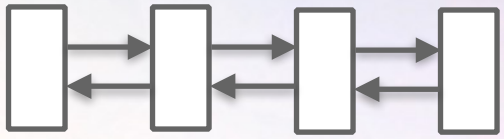
Removing an arbitrary element: Which container is fastest?

A	**std::vector**	B	**std::list**
C	**std::unordered_set**	D	**std::set**

Performance - Removing the First Element

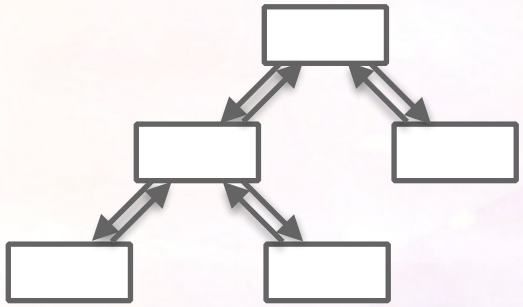
std::vector

std::list



std::deque

std::set



Removing the first element: Which container is fastest?

A	**std::vector**	B	**std::list**
C	**std::deque**	D	**std::set**

Programming Task

Task (2_STL_Algorithms/Containers/RemoveFront): Copy-and-paste the following code into quick-bench.com. Benchmark the time to remove the front element from a standard container.

2. STL Algorithms – STL Containers

Quick C++ Benchmark

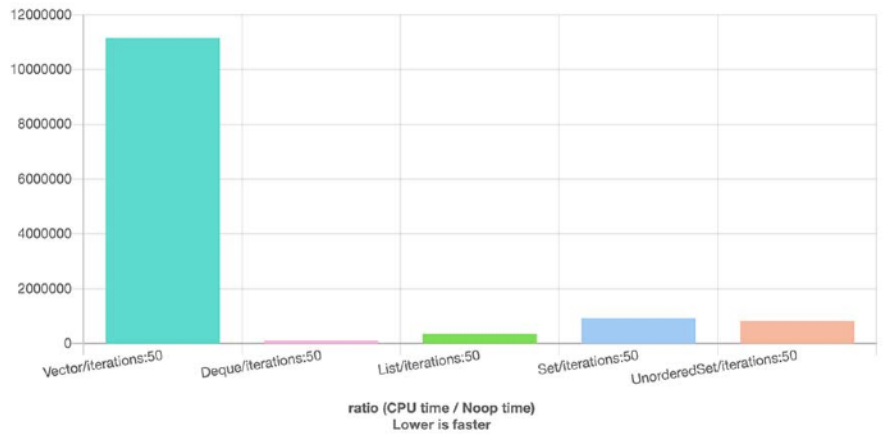
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file RemoveFront.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to r
*       the front element from a standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

2. STL Algorithms – STL Containers

Quick C++ Benchmark

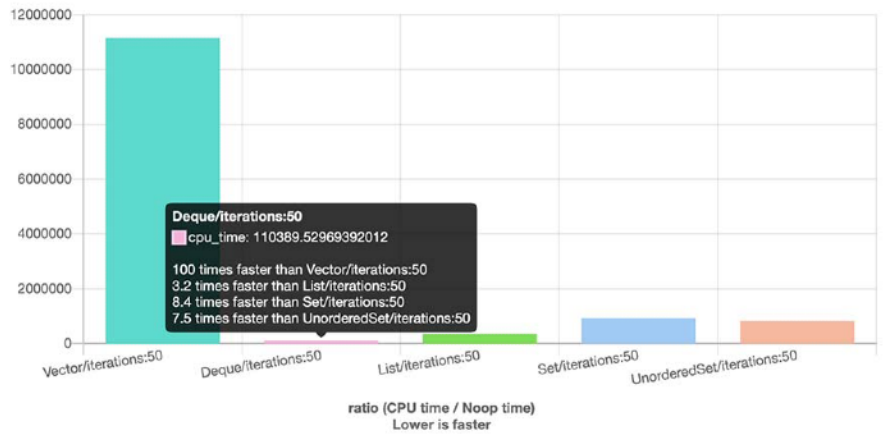
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
/*****************************************************************************
*
* \file RemoveFront.cpp
* \brief C++ Training - Container Performance Benchmark
*
* Copyright (C) 2015-2021 Klaus Iglberger - All Rights Reserved
*
* This file is part of the C++ training by Klaus Iglberger. The file may only be used in
* context of the C++ training or with explicit agreement by Klaus Iglberger.
*
* Task: Copy-and-paste the following code into 'quick-bench.com'. Benchmark the time to r
*       the front element from a standard container.
*
*****************************************************************************/

#include <algorithm>
#include <deque>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <set>
#include <stdexcept>
#include <string>
#include <type_traits>
#include <unordered_set>
#include <vector>


//--Benchmark configuration-----------------------------------------------------

constexpr size_t size( 10000 );  // Size of the generated container
constexpr size_t iterations( 50 );  // Number of benchmark iterations
using Type = int;  // Type of the elements (an arithmetic type or std::string)

#define BENCHMARK_VECTOR         1
#define BENCHMARK_DEQUE          1
#define BENCHMARK_LIST           1
#define BENCHMARK_SET            1
#define BENCHMARK_UNORDERED_SET  1


//--Benchmark setup-------------------------------------------------------------

std::random_device rd{};
std::mt19937 mt{ rd() };

template< typename T >
std::vector<T> generateOrder()
{
   std::vector<T> v( size );

   if constexpr( std::is_arithmetic_v<T> )
   {
      std::iota( begin(v), end(v), T{} );
      std::shuffle( begin(v), end(v), mt );
```

compiler = GCC 10.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

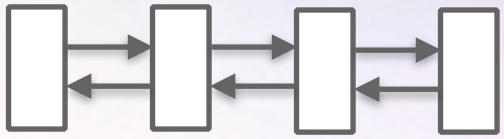
Charts | Assembly



Show Noop bar

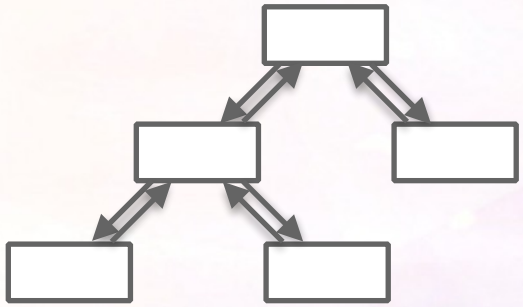
std::vector

std::list



std::deque

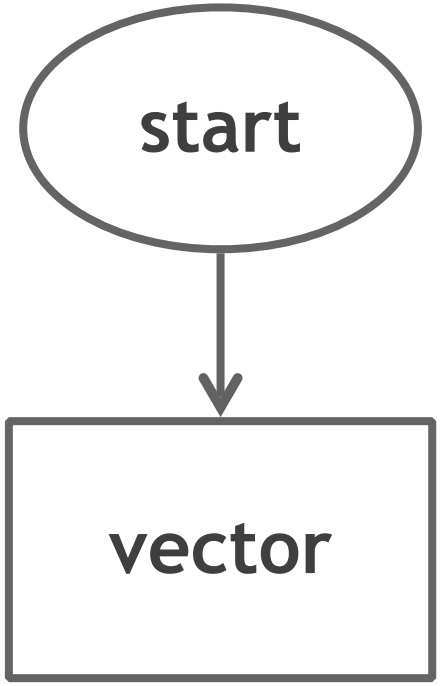
std::set



Removing the first element: Which container is fastest?

A	**std::vector**	B	**std::list**
C	**std::deque**	D	**std::set**

Choosing a Container



When in doubt, use a vector!

Choosing a Container

Use a **`std::vector`** by default.

Use a **`std::deque`** when you …

- … have many elements and you need to frequently insert at the front.

Use a **`std::list`** when you …

- … need stable nodes;
- … need to minimize the time of element insertion (splice).

Use a **`std::unordered_set`**/**`std::unordered_map`** when you …

- … need fast lookup times for a large number of elements;
- … when the collection is used many times for lookup.

Choosing a Container

Use a **`std::set`**/**`std::map`** when you …

- … need fast lookup times for a large number of **ordered** elements;
- … when the collection is used many times for lookup.

Container Guidelines

Guideline: Build on the standard library containers. They are fast, reliable, robust and thoroughly tested.

Guideline: Know the standard containers and their underlying data structure.

Guideline: If in doubt, use a vector.

Guideline: Prove that a data structure is faster by benchmarks.

Guideline: Use compact data structures for performance reasons.

Container Guidelines

Core Guideline SL.con.2: Prefer using STL vector by default unless you have a reason to use a different container.

Classic STL Allocators

Classic STL Allocators

- Every STL container has a template parameter for an allocator
- The default allocator is `std::allocator` (which uses `std::new`)

```
template< class T, class Allocator = std::allocator<T> >
class vector;

template< class T, class Allocator = std::allocator<T> >
class list;

template< class Key
        , class Hash = std::hash<Key>
        , class KeyEqual = std::equal_to<Key>
        , class Allocator = std::allocator<Key> >
class unordered_set;
```

Classic STL Allocator Details

```
namespace std {

template< typename T >
class allocator
{
 public:
   using value_type = T;
   using pointer = T*;









 private:
   // ...
};

}  // namespace std
```

Classic STL Allocator Details

```
namespace std {

template< typename T >
class allocator
{
 public:
   using value_type = T;
   using pointer = T*;

   pointer allocate( size_t numObjects
                   , void const* localityHint = nullptr )
   {
      // ...
   }




 private:
   // ...
};

}  // namespace std
```

Classic STL Allocator Details

```
namespace std {

template< typename T >
class allocator
{
 public:
   using value_type = T;
   using pointer = T*;

   pointer allocate( size_t numObjects
                   , void const* localityHint = nullptr )
   {
      // ...
   }
   void deallocate( pointer ptr, size_t numObjects ) noexcept
   {
      // ...
   }

 private:
   // ...
};

}  // namespace std
```

std::allocator

Defined in header <memory>

```
template< class T >
struct allocator;                    (1)

template<>
struct allocator<void>;              (2)   (deprecated in C++17)
```

Member types

Type	Definition
value_type	T
pointer (deprecated in C++17)	T*
const_pointer (deprecated in C++17)	const T*
reference (deprecated in C++17)	T&
const_reference (deprecated in C++17)	const T&
size_type (deprecated in C++17)	std::size_t
difference_type (deprecated in C++17)	std::ptrdiff_t
propagate_on_container_move_assignment(C++14)	std::true_type
rebind (deprecated in C++17)	template< class U > struct rebind { typedef allocator<U> other; };
is_always_equal(C++17)	std::true_type

std::allocator

Member functions

(constructor)	creates a new allocator instance (public member function)
(destructor)	destructs an allocator instance (public member function)
address (deprecated in C++17)	obtains the address of an object, even if `operator&` is overloaded (public member function)
allocate	allocates uninitialized storage (public member function)
deallocate	deallocates storage (public member function)
max_size (deprecated in C++17)	returns the largest supported allocation size (public member function)
construct (deprecated in C++17)	constructs an object in allocated storage (public member function)
destroy (deprecated in C++17)	destructs an object in allocated storage (public member function)

Non-member functions

operator== **operator!=**	compares two allocator instances (public member function)

Allocator Example

Task (2_STL_Algorithms/Allocators/ArenaAllocator): Discuss the mechanics of the following `ArenaAllocator`.

Allocator Example

Task (2_STL_Algorithms/Allocators/AlignedAllocator): Discuss the mechanics of the following `AlignedAllocator`.

Programming Task

Task (2_STL_Algorithms/Allocators/AlignedAllocator): Implement an `AlignedAllocator`. The allocator should encapsulate the acquisition of specifically aligned memory by making use of the C++17 functions `std::aligned_alloc()` and `std::free()`.

C++17 Polymorphic Memory Resources

C++17 Polymorphic Memory Resources


```
// ...
#include <memory_resource>


int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!









   return EXIT_SUCCESS;
}
```

C++17 Polymorphic Memory Resources


```
// ...
#include <memory_resource>


int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };









   return EXIT_SUCCESS;
}
```

C++17 Polymorphic Memory Resources


```
// ...
#include <memory_resource>


int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };









   return EXIT_SUCCESS;
}
```

C++17 Polymorphic Memory Resources


```
// ...
#include <memory_resource>


int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( auto const& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

C++17 Polymorphic Memory Resources

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```
(1)

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator

C++17 Polymorphic Memory Resources

std::pmr::polymorphic_allocator

Defined in header <memory_resource>

```
template< class T >                          (since C++17)
class polymorphic_allocator;                 (until C++20)

template< class T = std::byte >              (since C++20)
class polymorphic_allocator;
```

The class template `std::pmr::polymorphic_allocator` is an *Allocator* which exhibits different allocation behavior depending upon the `std::pmr::memory_resource` from which it is constructed. Since `memory_resource` uses runtime polymorphism to manage allocations, different container instances with `polymorphic_allocator` as their static allocator type are interoperable, but can behave as if they had different allocator types.

All specializations of `polymorphic_allocator` meet the Allocator completeness requirements.

Member types

Member type	definition
`value_type`	`T`

Member functions

(constructor)	Constructs a `polymorphic_allocator` (public member function)
(destructor) (implicitly declared)	Implicitly declared destructor (public member function)
operator= [deleted]	Copy assignment operator is deleted (public member function)

Public member functions

C++17 Polymorphic Memory Resources

Public member functions

`allocate`	Allocate memory (public member function)
`deallocate`	Deallocate memory (public member function)
`construct`	Constructs an object in allocated storage (public member function)
`destroy` (deprecated in C++20)	Destroys an object in allocated storage (public member function)
`allocate_bytes` (C++20)	Allocate raw aligned memory from the underlying resource (public member function)
`deallocate_bytes` (C++20)	Free raw memory obtained from allocate_bytes (public member function)
`allocate_object` (C++20)	Allocates raw memory suitable for an object or an array (public member function)
`deallocate_object` (C++20)	Frees raw memory obtained by allocate_object (public member function)
`new_object` (C++20)	Allocates and constructs an object (public member function)
`delete_object` (C++20)	Destroys and deallocates an object (public member function)
`select_on_container_copy_construction`	Create a new `polymorphic_allocator` for use by a container's copy constructor (public member function)
`resource`	Returns a pointer to the underlying memory resource (public member function)

Non-member functions

`operator==`	compare two `polymorphic_allocators` (function)
`operator!=` (removed in C++20)	

C++17 Polymorphic Memory Resources

std::pmr::memory_resource

Defined in header <memory_resource>

```
class memory_resource;        (since C++17)
```

The class `std::pmr::memory_resource` is an abstract interface to an unbounded set of classes encapsulating memory resources.

Member functions

(constructor) (implicitly declared)	constructs a new `memory_resource` (public member function)
(destructor) [virtual]	destructs an `memory_resource` (virtual public member function)
operator= (implicitly declared)	Implicitly declared copy assignment operator (public member function)

Public member functions

allocate	allocates memory (public member function)
deallocate	deallocates memory (public member function)
is_equal	compare for equality with another `memory_resource` (public member function)

Private member functions

do_allocate [virtual]	allocates memory (virtual private member function)
do_deallocate [virtual]	deallocates memory (virtual private member function)

C++17 Polymorphic Memory Resources

The class `std::pmr::memory_resource` is an abstract interface to an unbounded set of classes encapsulating memory resources.

Member functions

(constructor) (implicitly declared)	constructs a new `memory_resource` (public member function)
(destructor) [virtual]	destructs an `memory_resource` (virtual public member function)
operator= (implicitly declared)	Implicitly declared copy assignment operator (public member function)

Public member functions

allocate	allocates memory (public member function)
deallocate	deallocates memory (public member function)
is_equal	compare for equality with another `memory_resource` (public member function)

Private member functions

do_allocate [virtual]	allocates memory (virtual private member function)
do_deallocate [virtual]	deallocates memory (virtual private member function)
do_is_equal [virtual]	compare for equality with another `memory_resource` (virtual private member function)

Non-member-functions

operator== **operator!=** (removed in C++20)	compare two `memory_resources` (function)

Standard Memory Resources

Standard Memory Resources

std::pmr::get_default_resource

Defined in header <memory_resource>

```
std::pmr::memory_resource* get_default_resource() noexcept;     (since C++17)
```

- Returns the default polymorphic memory resource
- Default can be changed via `std::pmr::set_default_resource()`

```
// ...
#include <memory_resource>

int main()
{
   using std::pmr::vector;
   using std::pmr::string;

   vector<string> strings1{};  // By default uses the default memory resource
   vector<string> strings2{ std::pmr::get_default_resource() };  // Same as above

   // ...
}
```

Standard Memory Resources

std::pmr::new_delete_resource

Defined in header <memory_resource>

```
std::pmr::memory_resource* new_delete_resource() noexcept;     (since C++17)
```

- The default memory resource
- returned by `std::pmr::get_default_resource()` (i.e. as singleton)

```
// ...
#include <memory_resource>

int main()
{
   using std::pmr::vector;
   using std::pmr::string;

   vector<string> strings1{};  // By default uses the default memory resource
   vector<string> strings2{ std::pmr::new_delete_resource() };  // Same as above

   // ...
}
```

Standard Memory Resources

std::pmr::null_memory_resource

Defined in header <memory_resource>

```
std::pmr::memory_resource* null_memory_resource() noexcept;    (since C++17)
```

- Each allocation throws a `std::bad_alloc` exception
- implemented as a singleton

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };


   // ...
}
```

Standard Memory Resources

std::pmr::monotonic_buffer_resource

Defined in header <memory_resource>

```
class monotonic_buffer_resource : public std::pmr::memory_resource;
```
(since C++17)

- Provides the ability to place all memory in big chunks of memory
- You can pass the buffer to be used as memory (e.g. stack memory)
- The memory resource never deallocates until the resource as a whole is deallocated (i.e. deallocation is a no-op).

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };
   // ...
}
```

Standard Memory Resources

std::pmr::synchronized_pool_resource

Defined in header <memory_resource>

```
class synchronized_pool_resource : public std::pmr::memory_resource;     (since C++17)
```

- Tries to locate all memory close to each other (i.e. force minimal fragmentation of memory)
- Act as wrapper for other memory resources
- Thread-safe

```
// ...
#include <memory_resource>

int main()
{
   std::pmr::monotonic_buffer_resource buffer{ 1000
                                             , std::pmr::new_delete_resource() };
   std::pmr::synchronized_pool_resource pool{ &buffer };

   // ...
}
```

Standard Memory Resources

std::pmr::unsynchronized_pool_resource

Defined in header <memory_resource>

```
class unsynchronized_pool_resource : public std::pmr::memory_resource;     (since C++17)
```

- Tries to locate all memory close to each other (i.e. force minimal fragmentation of memory)
- Act as wrapper for other memory resources
- Not thread-safe

```
// ...
#include <memory_resource>

int main()
{
   std::pmr::monotonic_buffer_resource buffer{ 1000
                                             , std::pmr::new_delete_resource() };
   std::pmr::unsynchronized_pool_resource pool{ &buffer };

   // ...
}
```

Programming Task

Task (2_STL_Algorithms/Allocators/MemoryResource):

Extend the following example for C++17 polymorphic memory resources:

Step 1: Instead of going out of memory via `std::pmr::null_memory_resource`, provide a `std::pmr::new_delete_resource` as backend allocator.

Step 2: Track the dynamic memory allocations via a custom memory resource called `Tracker`.

Step 3: Introduce an allocator called `Mallocator`, which builds on `std::malloc/std::free`.

Programming Task

Task (2_STL_Algorithms/Allocators/UnsynchronizedPool):

Compare the allocation behavior of the following two options:

Option 1: Use a `std::map` equipped with a `std::allocator`.

Option 2: Use a `std::pmr::map` equipped with a `std::pmr::unsynchronized_pool` allocator.

Programming Task

Task (2_STL_Algorithms/Allocators/AllocatorPerformance): Copy-and-paste the following code into quick-bench.com. Benchmark the time to add elements to the end of a `std::pmr::list` container.

Custom Polymorphic Memory Resources

Custom Polymorphic Memory Resources


```
class Tracker
    : public std::pmr::memory_resource
{
 public:




 protected:



};
```

Custom Polymorphic Memory Resources


```
class Tracker
   : public std::pmr::memory_resource
{
 public:
   explicit Tracker( std::pmr::memory_resource* upstream =
                        std::pmr::get_default_resource() )
      : upstream_{ upstream }
   {}

   explicit Tracker( std::string prefix
                   , std::pmr::memory_resource* upstream =
                        std::pmr::get_default_resource() )
      : upstream_{ upstream }
      , prefix_  { std::move(prefix) }
   {}

 protected:




   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Custom Polymorphic Memory Resources


```
class Tracker
   : public std::pmr::memory_resource
{
 public:
   explicit Tracker( std::pmr::memory_resource* upstream =
                        std::pmr::get_default_resource() )
      : upstream_{ upstream }
   {}

   explicit Tracker( std::string prefix
                   , std::pmr::memory_resource* upstream =
                        std::pmr::get_default_resource() )
      : upstream_{ upstream }
      , prefix_  { std::move(prefix) }
   {}

 protected:
   void* do_allocate( size_t bytes, size_t alignment ) override
   void do_deallocate( void* ptr, size_t bytes, size_t alignment ) override
   bool do_is_equal(std::pmr::memory_resource const& other) const noexcept override

   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Custom Polymorphic Memory Resources


```
class Tracker
   : public std::pmr::memory_resource
{
 public:
   // ...

 protected:
   void* do_allocate( size_t bytes, size_t alignment ) override
   {
      std::cout << prefix_ << "allocate " << bytes << " bytes\n";
      void* const ret = upstream_->allocate( bytes, alignment );
      return ret;
   }

   // ...

   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Custom Polymorphic Memory Resources


```
class Tracker
   : public std::pmr::memory_resource
{
 public:
   // ...

 protected:
   void do_deallocate( void* ptr, size_t bytes, size_t alignment ) override
   {
      std::cout << prefix_ << "deallocate " << bytes << " bytes\n";
      upstream_->deallocate( ptr, bytes, alignment );
   }

   // ...

   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Custom Polymorphic Memory Resources


```
class Tracker
    : public std::pmr::memory_resource
{
 public:
   // ...

 protected:
   bool do_is_equal(std::pmr::memory_resource const& other) const noexcept override
   {
      if( this == &other ) return true;

      auto const* const tracker = dynamic_cast<Tracker const*>( &other );
      return tracker != nullptr
          && tracker->prefix_ == prefix_
          && upstream_-> tracker_is_equal( *other->upstream_ );
   }

   // ...

   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Equality of Memory Resources

Equality of Memory Resources


```
class Tracker
   : public std::pmr::memory_resource
{
 public:
   // ...

 private:
   bool do_is_equal(std::pmr::memory_resource const& other) const noexcept override
   {
      if( this == &other ) return true;

      auto const* const tracker = dynamic_cast<Tracker const*>( &other );
      return tracker != nullptr
          && tracker->prefix_ == prefix_
          && upstream_->is_equal( * tracker->upstream_ );
   }

   // ...

   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Equality of Memory Resources


```
Tracker tracker1{ "tracker1: " };
Tracker tracker2{ "tracker2: " };


std::pmr::string s1{ “Too long for SSO", &tracker1 }; // Allocates with tracker1

std::pmr::string s2{ std::move(s1), &tracker1 };      // Moves (same tracker type)

std::pmr::string s3{ std::move(s2), &tracker2 };      // Copies (different prefix)

std::pmr::string s4{ std::move(s3) };                 // Moves (allocator copied)

std::string s5{ std::move(s4) };                      // Copies (different allocator)
```

Guideline: Remember that move operations between containers with different container type might copy and throw.

Equality of Memory Resources


```
class Tracker
    : public std::pmr::memory_resource
{
 public:
   // ...

 private:
   bool do_is_equal(std::pmr::memory_resource const& other) const noexcept override
   {
      if( this == &other ) return true;

      auto const* const tracker = dynamic_cast<Tracker const*>( &other );
      return tracker != nullptr
          /*&& tracker->prefix_ == prefix_*/
          && upstream_->is_equal( *tracker->upstream_ );
   }

   // ...

   std::pmr::memory_resource* upstream_{};
   std::string prefix_{};
};
```

Equality of Memory Resources


```
Tracker tracker1{ "tracker1: " };
Tracker tracker2{ "tracker2: " };


std::pmr::string s1{ “Too long for SSO", &tracker1 }; // Allocates with tracker1

std::pmr::string s2{ std::move(s1), &tracker1 };      // Moves (same tracker type)

std::pmr::string s3{ std::move(s2), &tracker2 };      // Moves (same tracker type)

std::pmr::string s4{ std::move(s3) };                 // Moves (allocator copied)

std::string s5{ std::move(s4) };                      // Copies (different allocator)
```

Allocator Guidelines

Guideline: Prefer C++17 polymorphic allocators to the classic allocators.

Guideline: Design allocators to be interchangeable.

constexpr on Objects

Used on objects, constexpr expresses that the value is known at compile time:

```
int size1;                          // ok, only known at runtime.

auto const size2 = size1;           // ok, const but also only runtime.

constexpr auto size3 = size1;       // Compilation error! size's value
                                    // is not known at compile time.

constexpr auto size4 = 10;          // ok, 10 is a compile time constant

std::array<int,size1> data1;        // Compilation error! Same problem.

std::array<int,size2> data2;        // Compilation error! Same problem.

std::array<int,size4> data4;        // ok, size4 is constexpr.
```

All constexpr objects are const, but not all const objects are constexpr.

constexpr on Functions

`constexpr` functions produce compile-time constants when they are called with compile-time constants:

- if the arguments passed to a `constexpr` function are known at compile time, the result is computed at compile time
- if at least one argument to a `constexpr` function is not known at compile time, the function is executed at runtime

constexpr on Functions

```
constexpr int pow( int base, int exp ) noexcept
{

    ...

}

constexpr auto numConds = 5;

std::array<int, pow(3, numConds)> results;
```

Implementation of the pow Function


```
constexpr int pow( int base, int exp ) noexcept
{

   return ( exp == 0 ? 1 : base * pow(base, exp-1) );

}

constexpr auto numConds = 5;

std::array<int, pow(3, numConds)> results;
```

Implementation of the pow Function


```
constexpr int pow( int base, int exp ) noexcept
{
    auto result = 1;
    for( int i = 0; i < exp; ++i )
        result *= base;

    return result;
}

constexpr auto numConds = 5;

std::array<int, pow(3, numConds)> results;
```

Limitations of constexpr Functions

`constexpr` functions are limited to taking and returning literal types, i.e. …

- … built-in types except `void`; and
- … `constexpr` user-defined types.

Programming Task

Task (2_STL_Algorithms/constexpr/Point): Rework the existing `Point` and `selection_sort()` functionality such that the final array is computed at compile time.

A constexpr Point Class


```
class Point {
 public:
   constexpr Point( double xVal = 0.0, double yVal = 0.0 ) noexcept
      : x_( xVal ), y_( yVal )
   {}

   constexpr double xValue() const noexcept { return x_; }
   constexpr double yValue() const noexcept { return y_; }

   void setX( double newX ) noexcept { x_ = newX; }
   void setY( double newY ) noexcept { y_ = newY; }

 private:
   double x_;
   double y_;
};
```

A constexpr Point Class


```
class Point {
 public:
   constexpr Point( double xVal = 0.0, double yVal = 0.0 ) noexcept
      : x_( xVal ), y_( yVal )
   {}

   constexpr double xValue() const noexcept { return x_; }
   constexpr double yValue() const noexcept { return y_; }

   constexpr void setX( double newX ) noexcept { x_ = newX; }
   constexpr void setY( double newY ) noexcept { y_ = newY; }

 private:
   double x_;
   double y_;
};
```

Compile-Time Computations with Points

```
constexpr Point p1( 9.4, 27.7 );  // uses 'constexpr' ctor
                                  // during compilation
constexpr Point p2( 28.8, 5.3 );  // same
```

```
constexpr Point
   midpoint( Point const& p1, Point const& p2 ) noexcept
{
   return { ( p1.xValue() + p2.xValue() ) / 2.0
          , ( p1.yValue() + p2.yValue() ) / 2.0 };
}

constexpr auto mid = midpoint( p1, p2 );  // Runs at compile time!
```

Compile-Time Computations with Points


```
constexpr Point reflection( Point const& p ) noexcept
{
   Point result{};

   result.setX( -p.xValue() );
   result.setY( -p.yValue() );

   return result;
}


constexpr Point p1( 9.4, 27.7 );
constexpr Point p2( 28.8, 5.3 );
constexpr auto mid = midpoint( p1, p2 );

constexpr auto reflectedMid = reflection( mid ); // Known at
                                                 // compile time!
```

Guidelines

Guideline: Use `constexpr` whenever possible.

Programming Task

Task (2_STL_Algorithms/constexpr/Factorial): Implement a factorial function that can be used to compute the size of a `std::array` at compile time.

Programming Task

Task (2_STL_Algorithms/constexpr/SelectionSort): Implement the `selection_sort()` function that can be used to sort a range at compile time.

Programming Task

Task (2_STL_Algorithms/constexpr/UniquePtr_constexpr): What is wrong with the following given `unique_ptr` class template? Try to find all flaws and think about how you would test the class to detect these flaws.

```
template< typename T >
class unique_ptr
{
 public:
   unique_ptr() {}
   unique_ptr( T* ptr ) : ptr_{ptr} {}

   T& operator*() const { return *ptr_; }
   T* get() const { return ptr_; }

 private:
   T* ptr_;
};
```

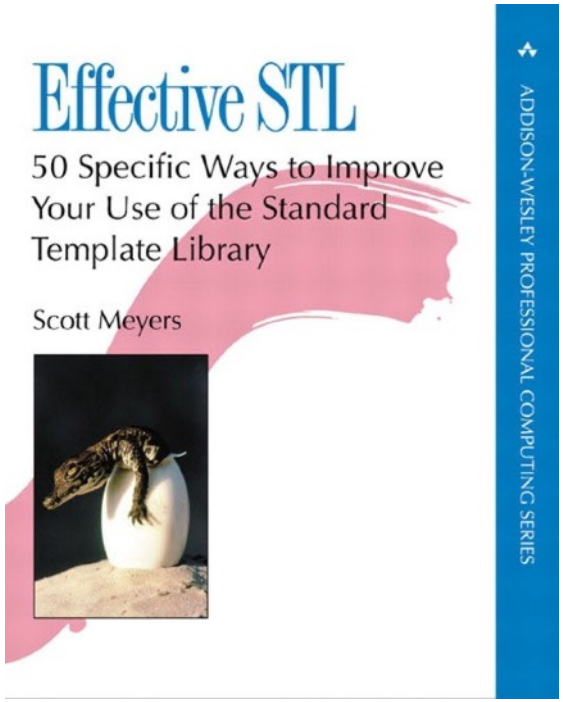
Programming Task

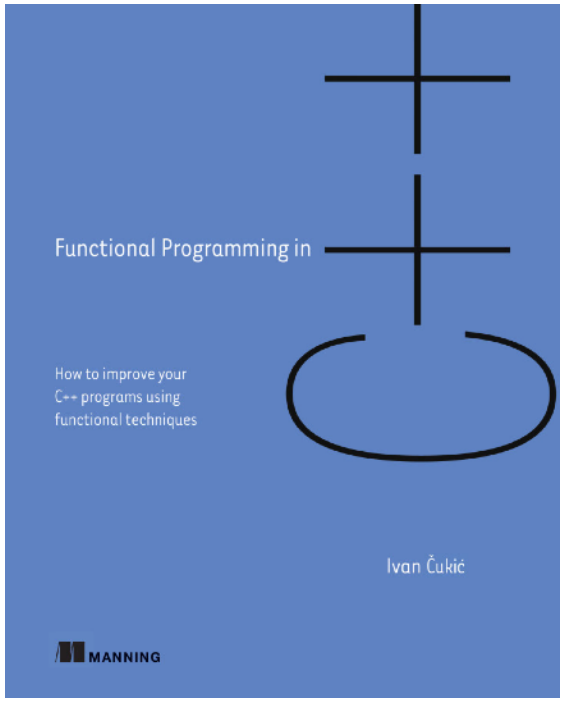
Task (2_STL_Algorithms/constexpr/Ranges_constexpr): What is wrong with the following given `print_countries()` function (think about changes in the input)? Try to find the flaw and think about how you would test the function to detect this flaw.

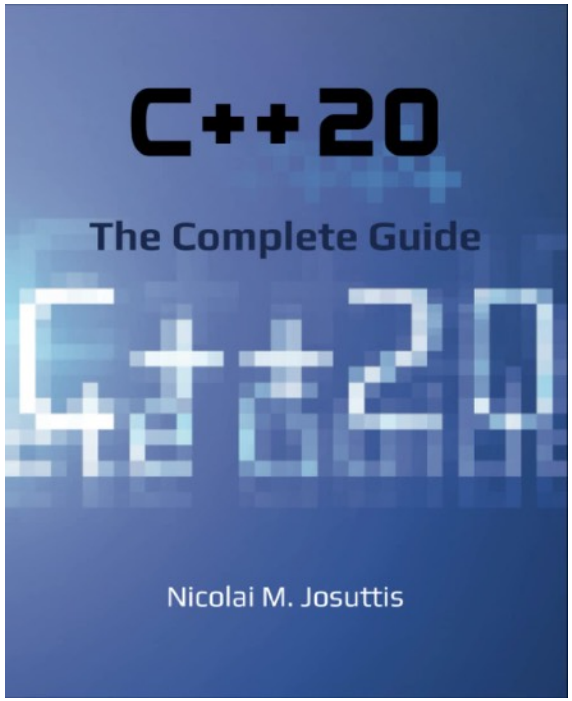
Things to Remember

- 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
How to improve your C++ programs using functional techniques
Ivan Čukić
MANNING


C++20
The Complete Guide
Nicolai M. Josuttis

References (Iterators)

- Bob Steagall, “Back to Basics: Classic STL”. CppCon 2021 (https://www.youtube.com/watch?v=tXUXl_RzkAk)
- Nicolai Josuttis, “Back to Basics: Iterators in C++”. CppCon 2023 (https://www.youtube.com/watch?v=26aW6aBVpk0)
- Barry Revzin, “Iterators and Ranges: Comparing C++ to D, Rust, and Others”. CPPP 2021 (https://www.youtube.com/watch?v=95uT0RhMGwA)

References (Algorithms)

- Alexander Stepanov, “STL and Its Design Principles”. (https://www.youtube.com/watch?v=COuHLky7E2Q)
- Stephan T. Lavavej, “rand() Considered Harmful”. (https://www.youtube.com/watch?v=LDPMpc-ENqY)
- 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)

References (Algorithms)

- Jeff Garland, “Effective Ranges: A Tutorial for Using C++2x Ranges”. CppCon 2023 (https://www.youtube.com/watch?v=QoaVRQvA6hI)
- 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 (https://www.youtube.com/watch?v=izxuLq_HZHA)
- Conor Hoekstra, “Algorithm Intuition (Part 1 of 2)”. CppCon 2019 (https://www.youtube.com/watch?v=pUEnO6SvAMo)
- Tristan Brindle, “An Overview of Standard Ranges”. CppCon 2019 (https://www.youtube.com/watch?v=SYLgG7Q5Zws)
- Tristan Brindle, “C++20 Ranges in Practice”. CppCon 2020 (https://www.youtube.com/watch?v=d_E-VLyUnzc)
- Tristan Brindle, “Conquering C++20 Ranges”. CppCon 2021 (https://www.youtube.com/watch?v=3MBtLeyJKg0)

References (Algorithms)

- Tina Ulbrich, “Throwing Tools at Ranges”. Meeting C++ 2023 (https://www.youtube.com/watch?v=9vudRM57hH0)
- Brian Ruth, “Thinking Functionally in C++”. CppCon 2023 (https://www.youtube.com/watch?v=6-WH9Hnec1M)

References (Functional Programming)

- 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)
- Victor Ciura, "The Imperatives Must Go!". CppCon 2022 (https://www.youtube.com/watch?v=M5HuOZ4sgJE)
- Brian Ruth, "Thinking Functionally in C++". CppCon 2023 (https://www.youtube.com/watch?v=6-WH9Hnec1M)
- Phil Nash, "Functional C++ for Fun & Profit". NDC Oslo 2017 (https://www.youtube.com/watch?v=tc9CDdJmoWE)

email: klaus.iglberger@cpp-training.com

LinkedIn: linkedin.com/in/klaus-iglberger

Xing: xing.com/profile/Klaus_Iglberger/cv