The fine details behind C++ Containers and Algorithms

Amir Kirsh

Amir Kirsh

Lecturer
Academic College of Tel-Aviv-Yaffo
and Tel-Aviv University

Developer Advocate
INCREDIBUILD

Co-Organizer of the **CoreCpp**
conference and meetup group

Suffering from slow builds?

It’s not just waste of time

It affects your dev cycles
and productivity

We also accelerate Yocto builds!

Our recent talks at Yocto Project Summit:

https://bit.ly/YPS-2022_IB_bitbake

https://bit.ly/YPS-2022_IB_Cache

Incredibuild + Yocto:

https://www.incredibuild.com/blog/announcing-incredibuild-support-for-yocto
https://www.incredibuild.com/lp/yocto

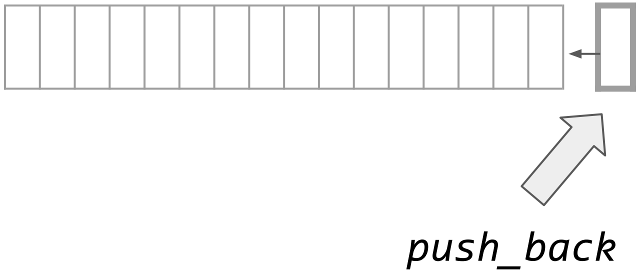
Topics

- Selecting the right container and using it properly
- Additions in C++17, C++20 (and C++23)
- Using algorithms smartly
- A few slides on iterators

Let’s start

What is the Complexity of:

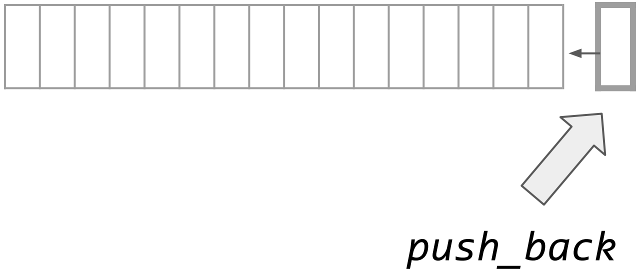
push_back **to a vector**



What is the Complexity of:

push_back **to a vector**

Amortized O(1)

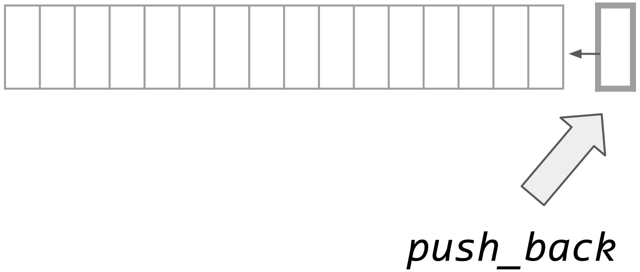


What is the Complexity of:

push_back **to a vector**

Amortized O(1)

How do we know?



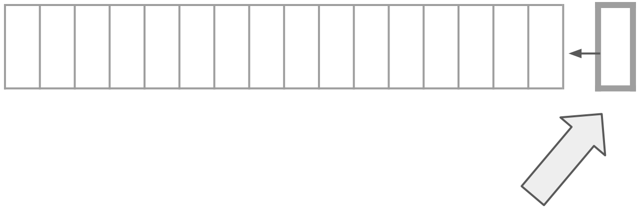
What is the Complexity of:

push_back **to a vector**

Amortized O(1)

How do we know?

Because *the spec* *requires it*!



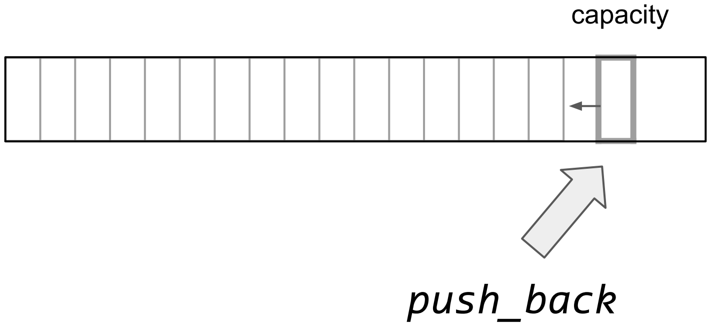
push_back

std::vector resizing following push_back

Case A

There is enough capacity

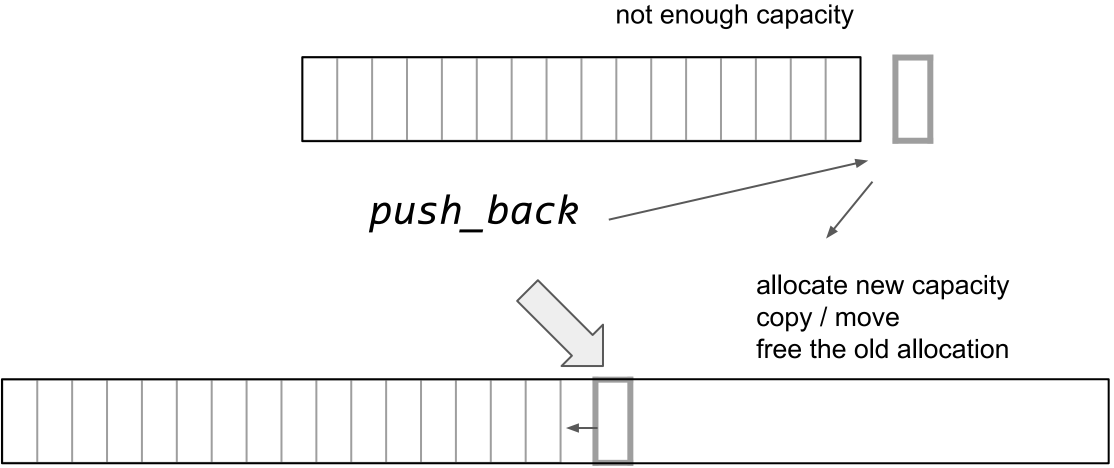
push_back ~ O(1)



Case B

There isn’t enough capacity

Needs to move / copy the vector ~ O(n)



To have: *push_back* ~ amortized O(1):

At most *1* of *n* calls may be of case B

Amortized Complexity

Amortized complexity considers the total worst case complexity of a sequence of operations, instead of just one operation.

Example 1:
If the *total* for *n* operations is in the worst case O(n) then the *amortized complexity* is O(1)

Example 2:
If the *total* for *n* operations is in the worst case O(n^2) then the *amortized complexity* is O(n)

Note:
Amortized complexity is NOT the *average* complexity over different inputs of size n!

See: Tarjan, Robert Endre (April 1985). *Amortized Computational Complexity*

An important side note on vector resizing!

There are 3 options when moving / copying the elements from the old allocation:

(a) **For trivially copyable elements:** vector may use **memcpy**

(b) **If the elements are nothrow_move_constructible:** vector **moves** the elements

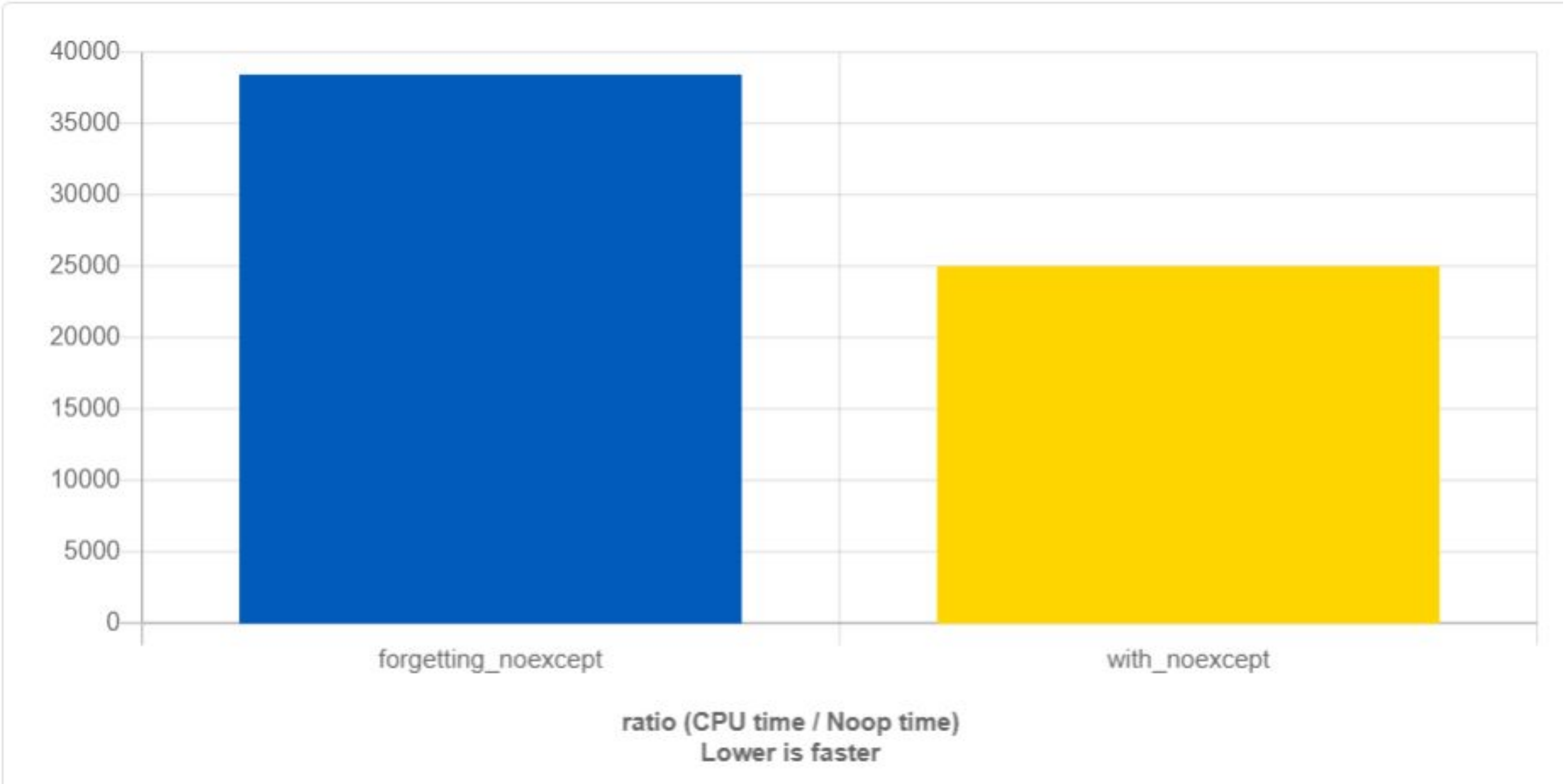
(c) Otherwise: the elements are **copied**

=> if you implement your own move make sure it is marked with **`noexcept`**

```
Widget(Widget&& w) noexcept { /* … */ }
```

See benchmark

Benchmark Results



A side note on std::copy

Although the spec doesn't require it, implementations would probably,

(a) **For trivially copyable and contiguous elements:** may use **memmove**
(b) Otherwise: **copy in a loop**

See: Spec and CppReference

For aliasing / overlapping issues and considerations look for ***Roi Barkan***'s talk, C++OnSea 2022:

Aliasing: Risks, Opportunities and Techniques

C++ Specifications - Complexity Requirements

In the spec (*examples*):

containers requirements

unordered associative containers + requirements

complexity of std::sort algorithm

complexity of std::ranges::partition algorithm

Then in CppReference (*examples*):

complexity of *std::vector::insert*

complexity of *std::list::insert*

complexity of *std::unordered_map::insert*

complexity of *std::search algorithm*

complexity of *std::sort algorithm*

What is the Complexity of:

- Sorting **a vector** using std::sort or std::ranges::sort
- Sorting **a list** using list::sort

What is the Complexity of:

- Sorting **a vector** using std::sort or std::ranges::sort
- Sorting **a list** using list::sort

O(n log(n))

See the spec for std::sort and for list::sort

(A side note: see the evolution of std::sort requirements here)

What is the Complexity of:

- Sorting **a vector** using std::sort or std::ranges::sort
- Sorting **a list** using list::sort

O(n log(n))

See the spec for std::sort and for list::sort

BUT, if we want to sort a list, there might be a better way than list::sort.
Any idea?

What is the Complexity of:

- Sorting **a vector** using std::sort or std::ranges::sort
- Sorting **a list** using list::sort

O(n log(n))

See the spec [for std::sort] and [for list::sort]

BUT, if we want to sort a list, there might be a better way than list::sort.
Any idea?

It might be more efficient to copy the list into a vector, sort the vector, then copy back
Why?

What is the Complexity of:

- Sorting **a vector** using std::sort or std::ranges::sort
- Sorting **a list** using list::sort

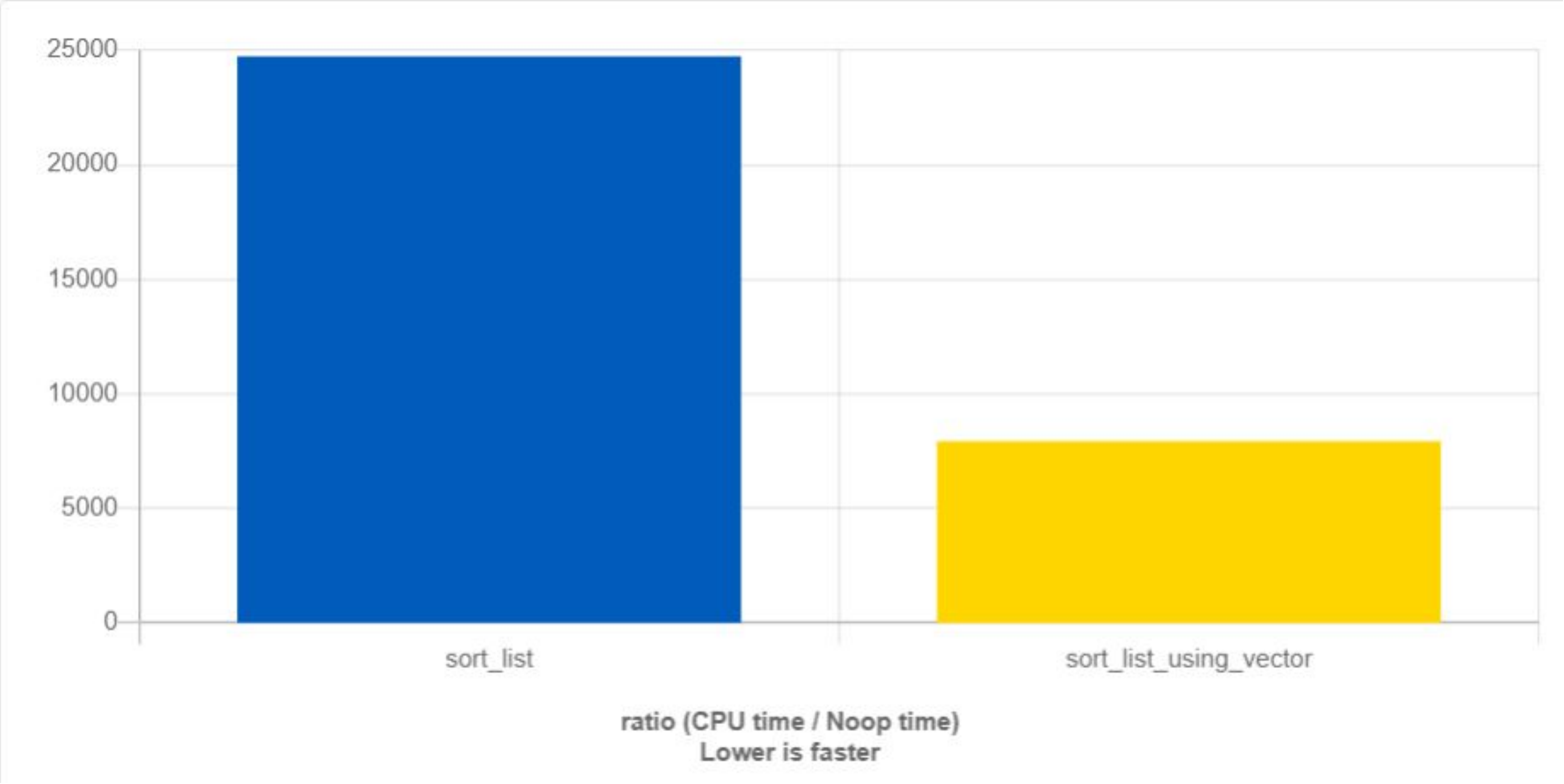
O(n log(n))

See the spec for std::sort and for list::sort

BUT, if we want to sort a list, there might be a better way than list::sort.
Any idea?

It might be more efficient to copy the list into a vector, sort the vector, then copy back
Why? See benchmark

Benchmark Results



Insert to front

What is the best way to insert *k* items as a bulk into the front of a vector?
Would it be better to use a list?

Insert to front

What is the best way to insert *k* items as a bulk into the front of a vector?
Would it be better to use a list?

The trick:

Insert to front

What is the best way to insert *k* items as a bulk into the front of a vector?
Would it be better to use a list?

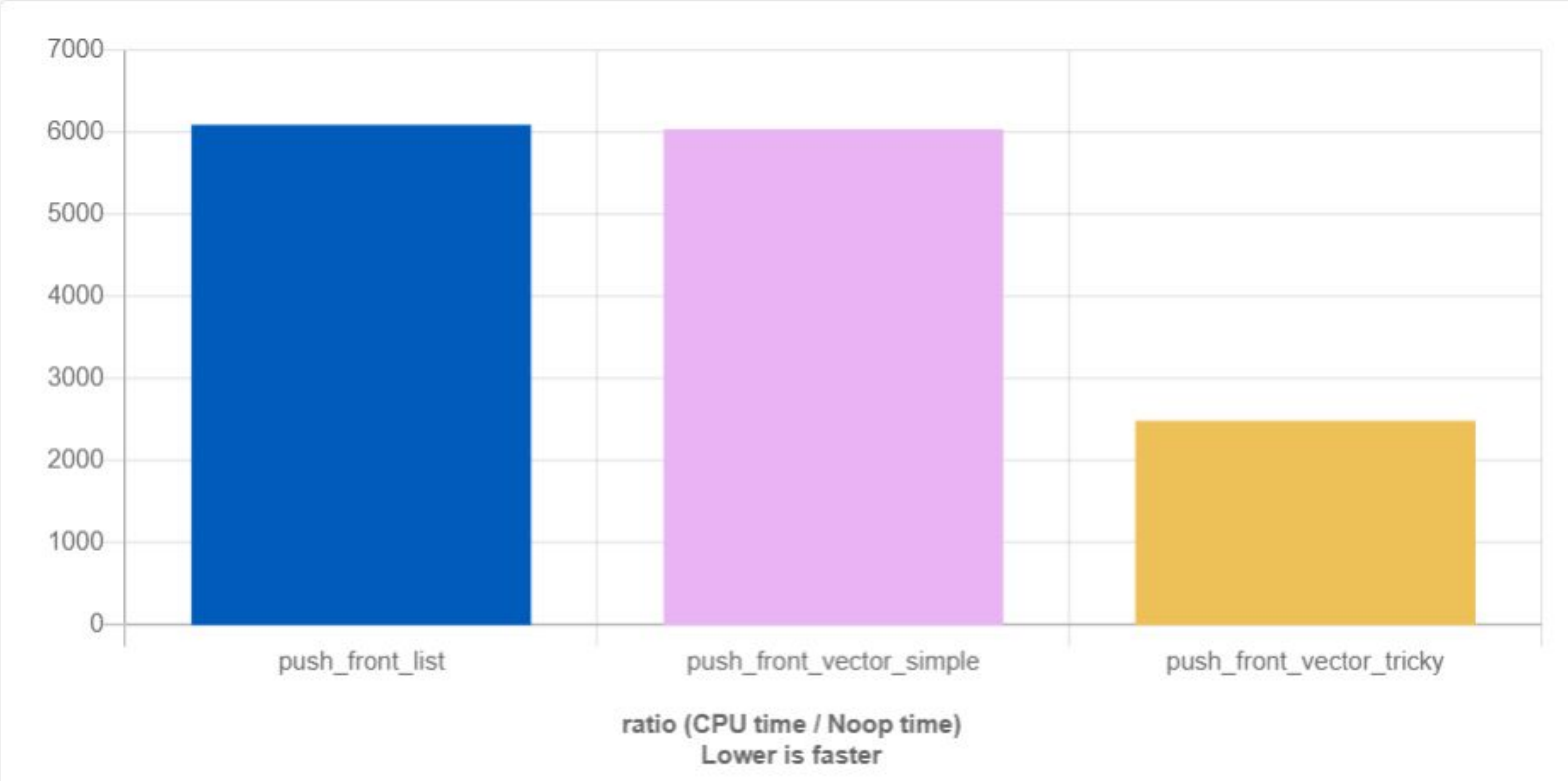
The trick:

reverse, push_back in opposite order, reverse back

Example inspired by ***Vladimir Vishnevskii***'s talk, C++OnSea 2022:
Refresher on containers, algorithms and performance

Benchmark

Benchmark Results



std::remove doesn’t remove

STL *remove* doesn't work as expected? - Stack Overflow

std::remove doesn’t remove

STL *remove* doesn't work as expected? - Stack Overflow

std::remove overrides the “removed” elements with consecutive elements that should be kept. It then returns an iterator to the “new” end

To actually complete the erase operation you should use the erase-remove idiom:

```
    v.erase(std::remove(v.begin(), v.end(), 9), v.end());
```

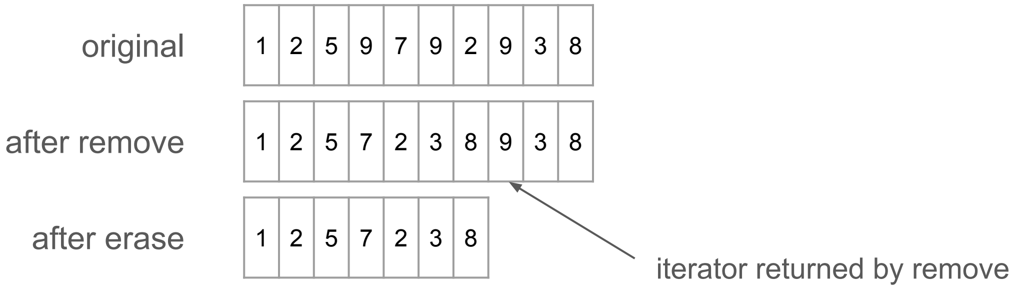
std::remove doesn’t remove

STL *remove* doesn't work as expected? - Stack Overflow

std::remove overrides the “removed” elements with consecutive elements that should be kept. It then returns an iterator to the “new” end

To actually complete the erase operation you should use the erase-remove idiom:

```
    v.erase(std::remove(v.begin(), v.end(), 9), v.end());
```



original	1	2	5	9	7	9	2	9	3	8
after remove	1	2	5	7	2	3	8	9	3	8
after erase	1	2	5	7	2	3	8			

iterator returned by remove

C++20 added std::erase and std::erase_if

```
std::erase(v, 9);
```

Erasing by index

What is the Complexity of:

Erasing *k* elements at given indices
Indices for deletion are sorted in descending order

- from **a list**
- from **a vector**

Erasing by index from a list

```
auto itr = lst.begin();
long pos = *indices_to_erase.begin();
std::advance(itr, pos);
for(auto index: indices_to_erase) { // indices_to_erase are sorted in descending order
    std::advance(itr, index - pos); // going backwards
    pos = index;
    itr = lst.erase(itr);
}
```

Erasing by index from a vector - naive

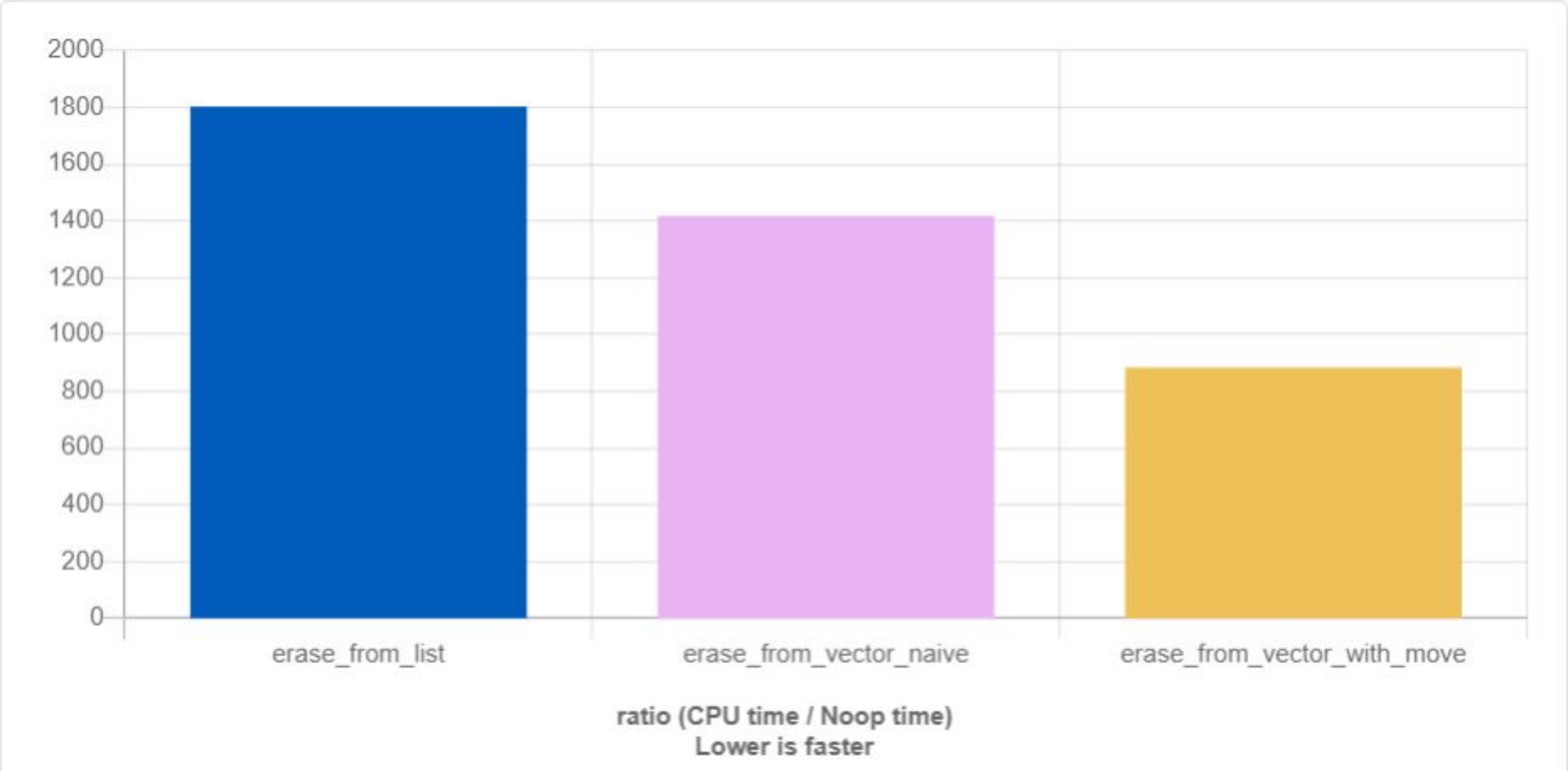
```
for(auto index: indices_to_erase) { // indices_to_erase are sorted in descending order
    vec.erase(vec.begin() + index);
}
```

Erasing by index from a vector - with move

```
size_t removed = 0;
for(auto index: indices_to_erase) { // indices_to_erase are sorted in descending order
    std::move(vec.begin() + index + 1, vec.end() - removed, vec.begin() + index);
    ++removed;
}
vec.erase(vec.end() - removed, vec.end());
```

https://godbolt.org/z/3novbGh5x

Benchmark Results



Homework

Implement a more efficient *erase by indices from a vector* - by running on the indices in ascending order.

Solution: https://godbolt.org/z/9qEvYPETz

Benchmark: https://quick-bench.com/q/FXfQmWoY6K-v0OgkByQUc9hEuuk

What is the Complexity of:

Finding the median of *n* items

What is the Complexity of:

Finding the median of *n* items

There is an algorithm, PICK, with O(n) worst case complexity!

However, another algorithm, Quickselect, which is O(n^2) at worst case, is usually faster.

They are both O(n) on average.

See https://cs.stackexchange.com/questions/1914/find-median-of-unsorted-array-in-on-time

What is the Complexity of:

Finding the median of *n* items

There is an algorithm, PICK, with O(n) worst case complexity!

However, another algorithm, Quickselect, which is $O(n^2)$ at worst case, is usually faster.

They are both O(n) on average.

See https://cs.stackexchange.com/questions/1914/find-median-of-unsorted-array-in-on-time
See also spec requirement for std::nth_element

What is the Complexity of:

find / insert - **unordered_map**

What is the Complexity of:

find / insert - **unordered_map**

O(1) average case

O(n) worst case

See the spec for find
See the spec for insert

Beware of a costly hash function

The hash function may be called more than you may think of

Code: https://godbolt.org/z/dYezqxMYb

See: unordered_map excess calls to hash function - Stack Overflow

What’s wrong with this code:

```
std::map<std::string, std::string> numbers;
for(auto[a, b]: std::vector<std::pair<const char*, const char*>>{
          {"One", "Uno"}, {"Two", "Duo"}, {"Three", "Tre"}, {"Four", "Quatro"}})   {
    numbers[a] = b;
}
```

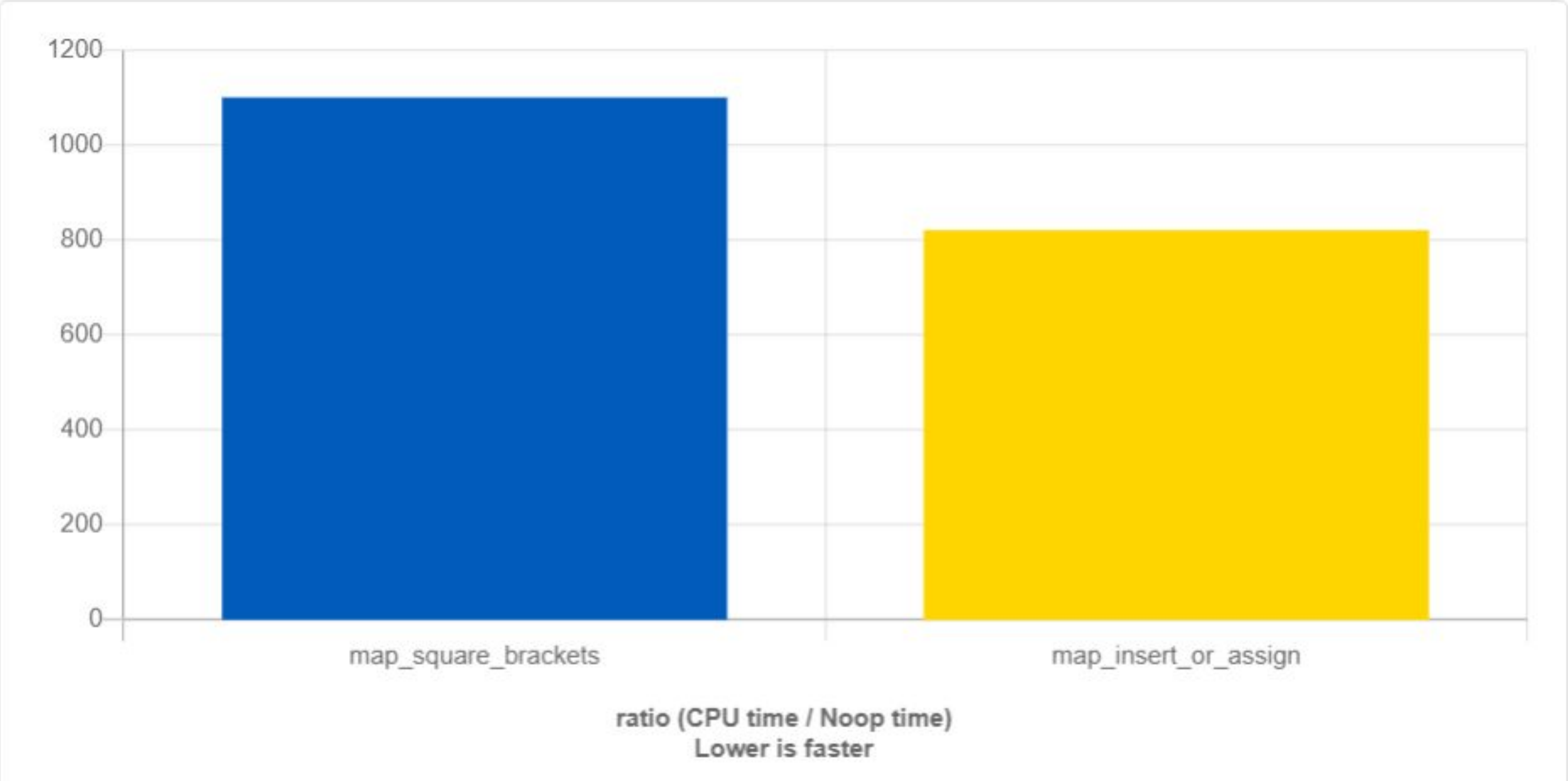
https://godbolt.org/z/nefosvcbn

What’s wrong with this code:

```
std::map<std::string, std::string> numbers;
for(auto[a, b]: std::vector<std::pair<const char*, const char*>>{
          {"One", "Uno"}, {"Two", "Duo"}, {"Three", "Tre"}, {"Four", "Quatro"}})  {
    numbers[a] = b;
}
```

Benchmark: square brackets vs. C++17 insert_or_assign

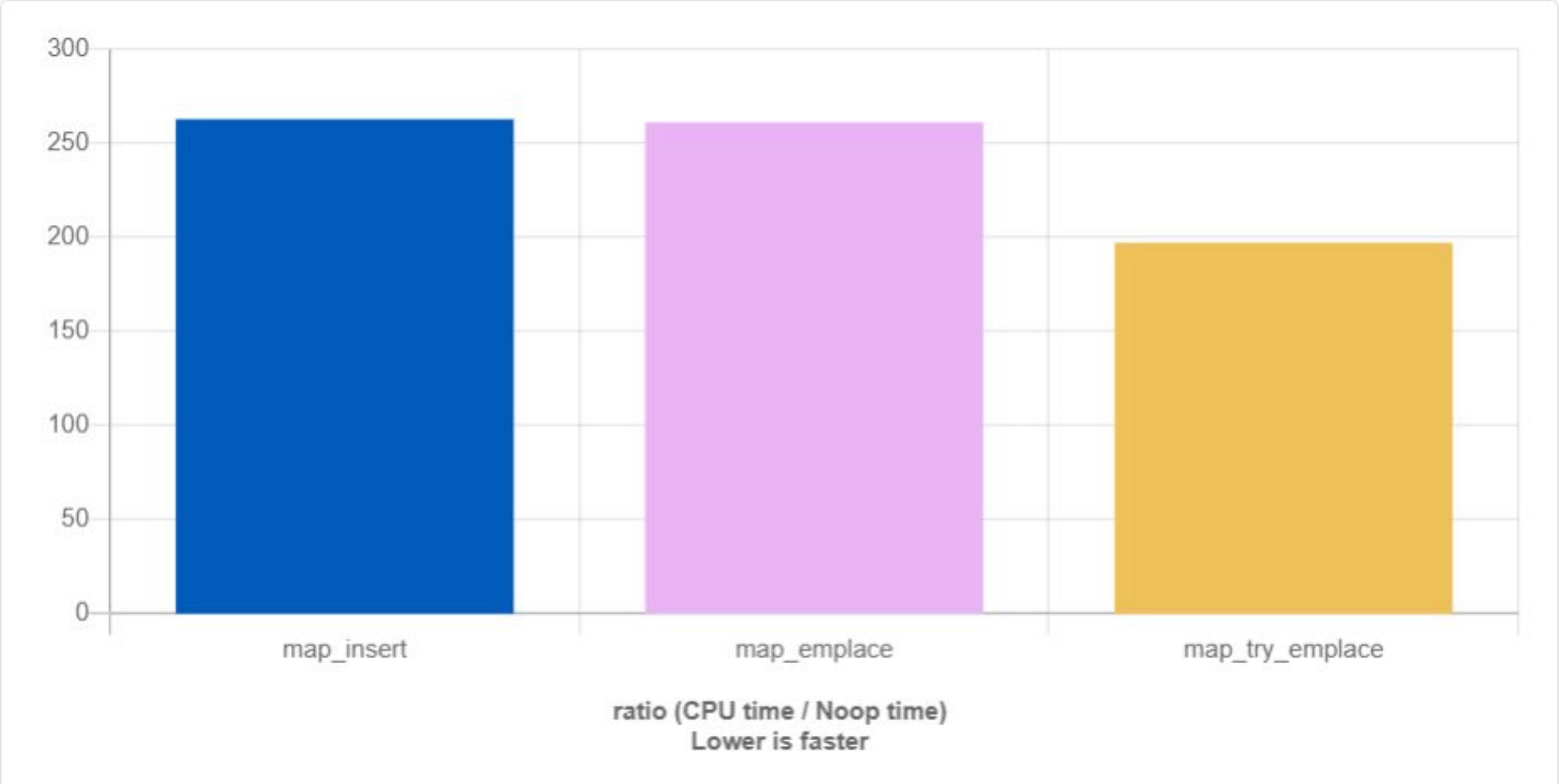
Benchmark Results



C++17 other additions to map / unordered_map

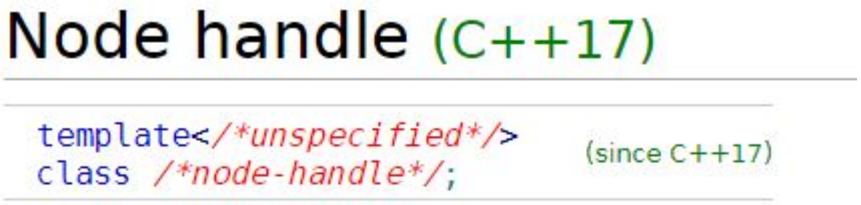
```
map<int, Person> persons;

// below goes through Person’s ctor + copy/move
persons.insert({1, Person("momo")});

// the old emplace, still goes through Person’s ctor, but on the callee side!
persons.emplace(2, "koko");

// below is even more efficient as it doesn’t go via Person’s ctor
// if key already exists!
persons.try_emplace(2, "koko2"); // note: try_emplace is C++17
```

Benchmark Results



C++17 other additions to map / unordered_map

New strange type in C++17:



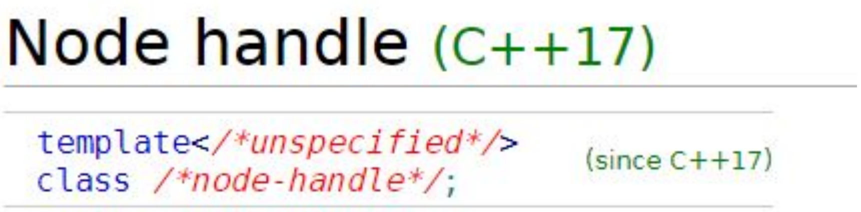
Node handle (C++17)

```
template</*unspecified*/>
class /*node-handle*/;
```

(since C++17)

C++17 other additions to map / unordered_map

New strange type in C++17:



Node handle (C++17)

```
template</*unspecified*/>
class /*node-handle*/;
```

(since C++17)

Used for:

- extract
- merge
- insert(node)

C++17 other additions to map / unordered_map

Code example:

```
std::unordered_map<int, string> numbers{{0, "one"}, {2, "two"}, {3, "three"}};
// Extract node handle
auto node = numbers.extract(0);
node.key() = 1;
// Insert node handle back
numbers.insert(std::move(node));
```

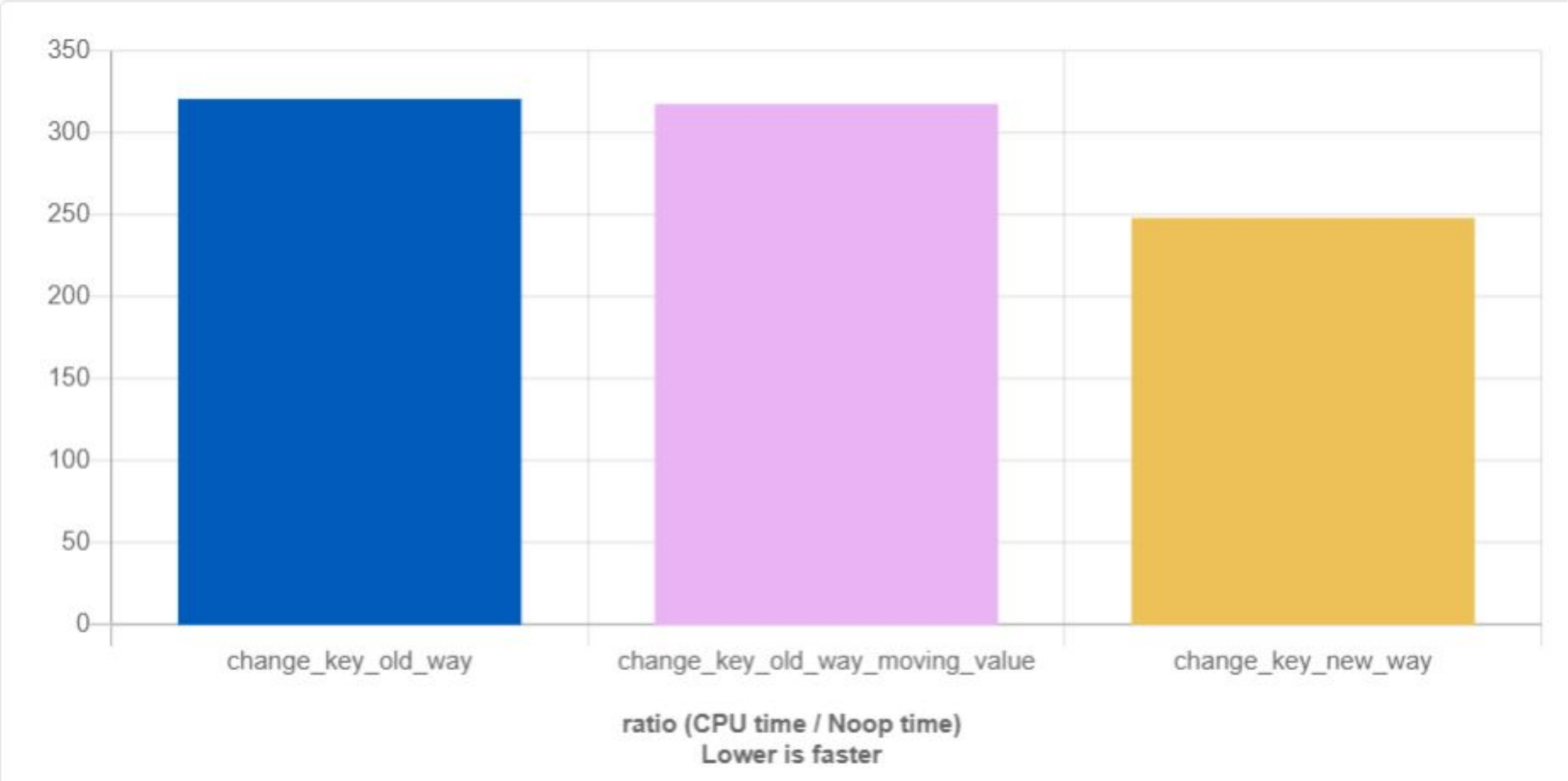
C++17 other additions to map / unordered_map

Code example:

```
std::unordered_map<int, string> numbers{{0, "one"}, {2, "two"}, {3, "three"}};
// Extract node handle
auto node = numbers.extract(0);
node.key() = 1;
// Insert node handle back
numbers.insert(std::move(node));
```

Is it better than the old way? Benchmark

Benchmark Results



C++17 other additions to map / unordered_map

If you wish to use **try_emplace with an hint**, **use it carefully** as it may hurt performance badly if you provide a bad hint. **Or better just don’t use it.**

See code and benchmark

(Note that even with a good hint you may not get improved performance, as the call must check that the hint was correct).

views

C++17 string_view
C++20 ranges views – Use them!

Example (from cppreference):

```
constexpr std::string_view words{"Hello-_-C++-_-20-_-!"};
constexpr std::string_view delim{"-_-"};
for (const auto word : std::views::split(words, delim)) {
  std::cout << std::quoted(std::string_view(word.begin(), word.end())) << ' ';
}
```

See also: lazy_split and lazy_split_view

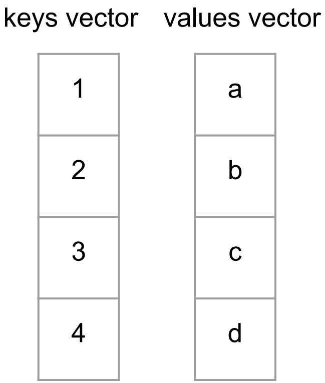
Other Data Structures?

Boost `flat_map`

And C++23 `flat_map` (see proposal doc)

=> Data locality

Search complexity – logarithmic



keys vector	values vector
1	a
2	b
3	c
4	d

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.

Would it be better to perform the two operations inside a single loop?

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.

Would it be better to perform the two operations inside a single loop?

Two loops ~ n + n = O(n)

Single loop with two operations ~ 2n = O(n)

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.
Would it be better to perform the two operations inside a single loop?

Two loops ~ n + n = O(n)
Single loop with two operations ~ 2n = O(n)

So are they the same?

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.
Would it be better to perform the two operations inside a single loop?

Two loops ~ n + n = O(n)
Single loop with two operations ~ 2n = O(n)

So are they the same? Complexity-wise yes, practically - not necessarily!

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.

Would it be better to perform the two operations inside a single loop?

It might be better due to ***data locality***

see benchmarks with std::list and std::vector

(and see also SO discussion with additional alternatives).

Two calls to std algorithms

A note:

std::ranges allows consecutive algorithm calls to be “lazily attached”
into a single loop

Two calls to std algorithms

A note:

std::ranges allows consecutive algorithm calls to be “lazily attached” into a single loop

ranges require its own talk, but if you are interested…

Here is a relevant code example (courtesy of Dvir Yitzchaki)

You may also want to watch Dvir’s CppCon 2019 talk on ranges

Parallel Algorithms

C++17 added Execution Policy to allow parallel execution for algorithms

There are benchmarks showing it may improve performance dramatically (but not always!)

See ***Vladimir Vishnevskii***'s talk, C++OnSea 2022:

Refresher on containers, algorithms and performance

Rainer Grimm's post: **Performance of the Parallel STL Algorithms**

Also: Using C++17 Parallel Algorithms for Better Performance - Microsoft C++ Team Blog

And: The Amazing Performance of C++17 Parallel Algorithms, is it Possible? - C++ Stories

Parallel Algorithms

Note the difference between:

- ***std::execution::par***

  allowing parallel execution using multiple threads, trying to utilize CPU cores
- ***std::execution::unseq***

  allowing single thread vectorization

See also: Difference between execution policies and when to use them

Thread safety

https://en.cppreference.com/w/cpp/container#Thread_safety

https://stackoverflow.com/questions/12931787/c11-stl-containers-and-thread-safety

Iterators invalidation rules

https://en.cppreference.com/w/cpp/container#Iterator_invalidation

https://stackoverflow.com/questions/6438086/iterator-invalidation-rules

Implementing your own iterator

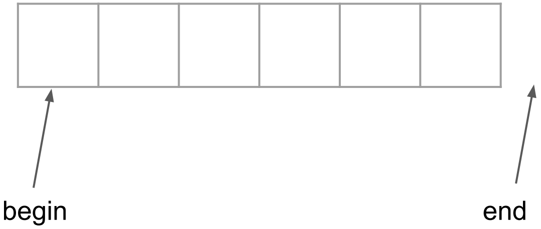
std::iterator being deprecated in C++17

Use Boost iterator

Or do some manual work:
[Preparation for std::iterator Being Deprecated - Stack Overflow]

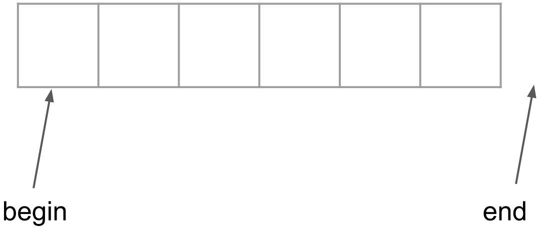
A note on implementing reverse_iterator

Iterator:



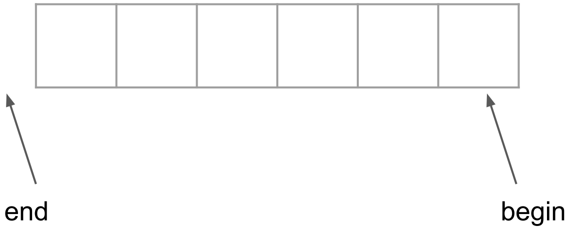
A note on implementing reverse_iterator

Iterator:



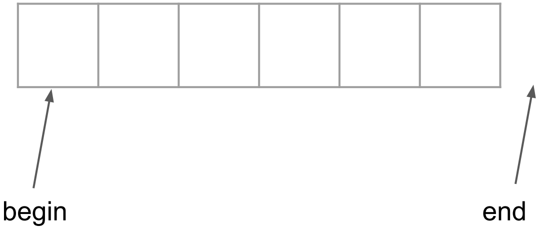
reverse_iterator?

```
auto operator++() {
  --pos;
 //…
}
```



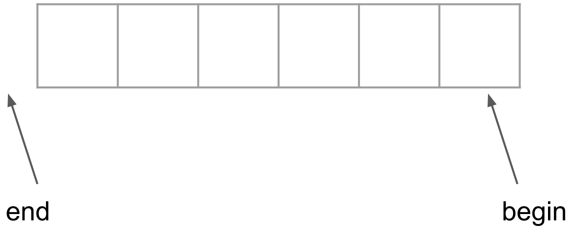
A note on implementing reverse_iterator

Iterator:



reverse_iterator?

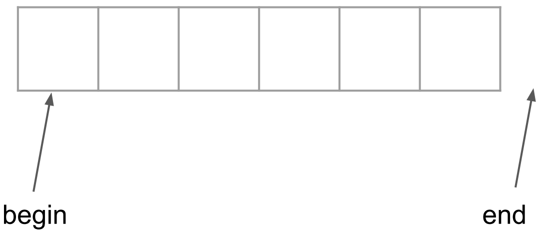
```
auto operator++() {
  --pos;
 //…
}
```



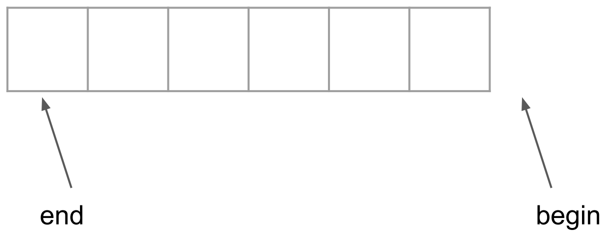
NO!

A note on implementing reverse_iterator

Iterator:



reverse_iterator:



```
auto operator*() {
  auto before = *this;
  return *(--before);
}
```

```
auto operator++() {
  --pos;
 //…
}
```

Yes!

See: How does std::reverse_iterator hold one before begin?

Summary

Picking the right container (1)

std::vector is the best, it’s not *us* who say that, it is the spec:

> When choosing a container, remember **`vector`** is best;
> leave a comment to explain if you choose from the rest!

Remember when vector is costly

Benchmark if you want to select another container

Picking the right container (2)

std::unordered_map

make sure to provide a good enough hash function for your key,
or forget about amortized O(1) operations…

hash function requirements in the spec:

[…] For two different values `t1` and `t2`, the probability that `h(t1)` and `h(t2)` compare equal should be very small, approaching `1.0 / numeric_limits<size_t>::max()`.

Using std algorithms

Don’t reinvent the wheel

e.g. don’t implement your own sort, you may accidentally implement bubble sort

Think

Implications of bad algorithms and improper use of data structures are potentially much bigger than other micro-performance improvements

Switching to a better algorithm can decrease runtime dramatically!

Be aware of invalidation rules and thread safety.

Don’t focus only on Big-O

The theoretical worst case Big O shouldn’t be your only decision factor:

- In real life, **constants** are important: 2n is better than 4n
- In real life, we might choose an algorithm with better **average performance** but *worse worst case complexity*
- **Memory locality** is highly important

Beware of ignoring the constant *c*

What is the complexity of a function with execution time:

$$t(n) = c*n$$

Beware of ignoring the constant *c*

What is the complexity of a function with execution time:

t(n) = c*n

That was a simple question…

O(n)

Beware of ignoring the constant *c*

What is the complexity of the code below?

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Beware of ignoring the constant *c*

What is the complexity of the code below?

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

it’s ***O(n)***

Beware of ignoring the constant *c*

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Suppose that we can achieve the same, with *t(n) = n * log n*

Which would be better?

Beware of ignoring the constant *c*

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Suppose that we can achieve the same, with $t(n) = n * \log n$

Which would be better?

$\log(n) < 64 < 100$, for any 64 bit n

Beware of ignoring the constant *c*

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Suppose that we can achieve the same, with *t(n) = n * log n*

Which would be better?

log(n) < 64 < 100, for any 64 bit *n*

log(vector::size) <= 64

You may reduce latency with a tradeoff

- **Prior setup** (e.g. sorting / indexing)

- **Space vs. Time** - using space to save runtime (e.g. caching, indexing)

When you benchmark

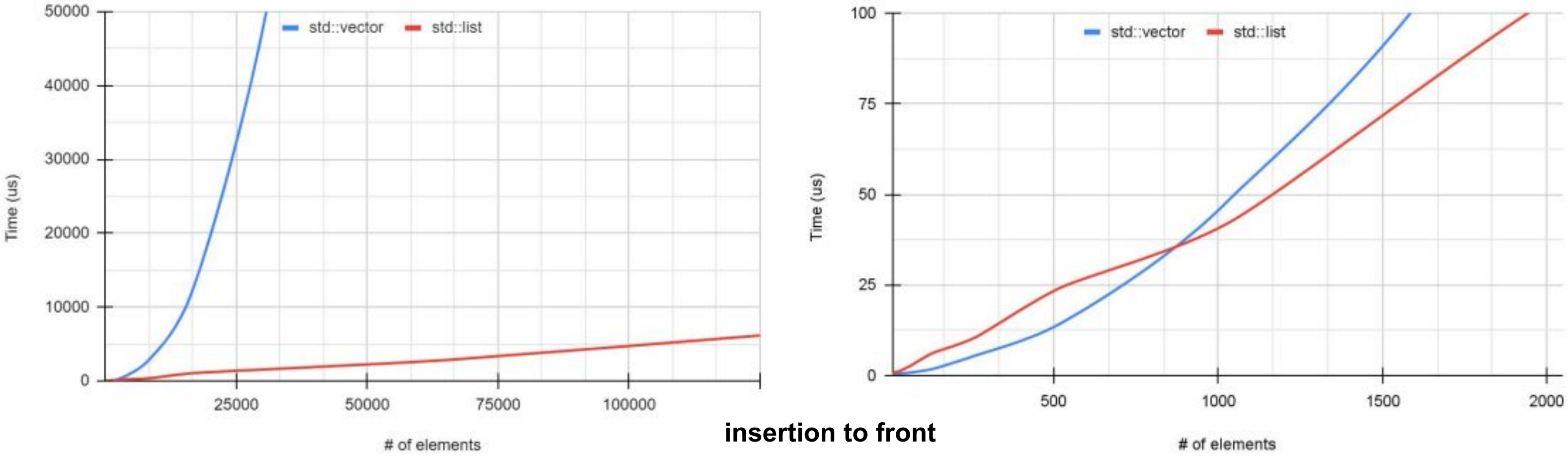
Use real data and the actual scale that you would run in production

Benchmarks results depend on data size

Charts below are taken from

Vladimir Vishnevskii's talk, C++OnSea 2022:

[Refresher on containers, algorithms and performance]()



insertion to front

It’s not pre-optimization

Thinking about the right container and algorithmic complexity
is not pre-optimization

It’s an essential element of your design and its ability to scale

However, try to design your application not to rely on the specific types of your data structures.

Thank you!

```
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });
```