Под капотом стандартной библиотеки C++

Insights into the C++ standard library

Pavel Novikov

@cpp_ape

R&D Align Technology

What's ahead

- use `std::vector` by default instead of `std::list`
- `emplace_back` and `push_back`
- prefer to use `std::make_shared`
- avoid static objects with non-trivial constructors/destructors
- small string optimization
- priority queue and heap algorithms
- sorting and selection
  - guarantee of strengthened worst case complexity O(n log(n)) for `std::sort`
  - when to use `std::sort`, `std::stable_sort`, `std::partial_sort`, `std::nth_element`
- use unordered associative containers by default
- beware of missing the variable name

```
using GadgetList = std::list<std::shared_ptr<Gadget>>;

GadgetList findAllGadgets(const Widget &w) {
  GadgetList gadgets;
  for (auto &item : w.getAllItems())
    if (isGadget(item))
      gadgets.push_back(getGadget(item));
  return gadgets;
}
void processGadgets(const GadgetList &gadgets) {
  for (auto &gadget : gadgets)
    processGadget(*gadget);
}
```

```
using GadgetList = std::list<std::shared_ptr<Gadget>>;

GadgetList findAllGadgets(const Widget &w) {
  GadgetList gadgets;
  for (auto &item : w.getAllItems())
    if (isGadget(item))
      gadgets.push_back(getGadget(item));
  return gadgets;
}
void processGadgets(const GadgetList &gadgets) {
  for (auto &gadget : gadgets)
    processGadget(*gadget);
}
```

is this reasonable use of `std::list`?

`std::list`

No copy/move while adding elements. 👍

Memory allocation for *each* element. 🤔

- expensive
- may fragment memory
- in general worse element locality compared to `std::vector`

std::vector

std::vector<T,A>::push_back() has *amortized* $O(1)$ complexity

```
std::vector<int> v; // capacity=0
v.push_back(1); // allocate, capacity=3
v.push_back(2);
v.push_back(3);
v.push_back(4); //alloc, move 3 elements, capacity=6
v.push_back(5);
```


MATH
CAUTION
MATH
AHEAD

std::vector

c — initial capacity

m — reallocation factor

To push back N elements we need

$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector

c — initial capacity

m — reallocation factor

To push back N elements we need

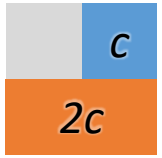
$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector



c — initial capacity

m — reallocation factor

To push back N elements we need

$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector

c — initial capacity

m — reallocation factor

To push back N elements we need

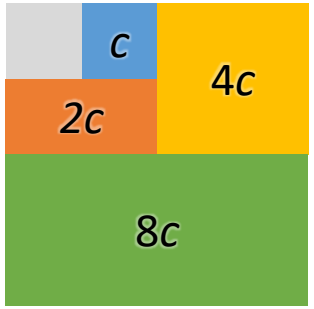
$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector



c — initial capacity

m — reallocation factor

To push back N elements we need

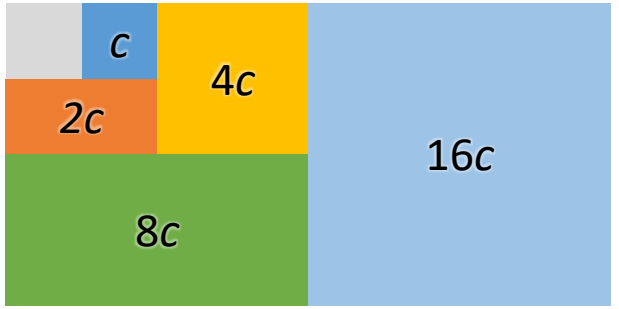
$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector



c — initial capacity

m — reallocation factor

To push back N elements we need

$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector

c — initial capacity

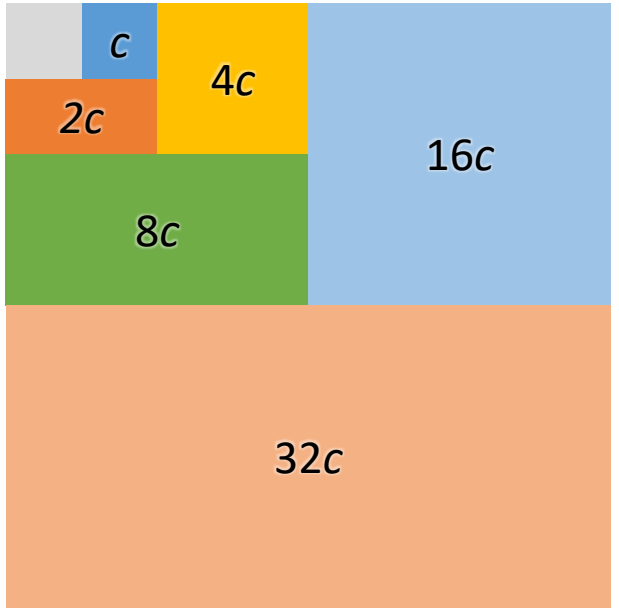
m — reallocation factor

To push back N elements we need $\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$



std::vector

c — initial capacity

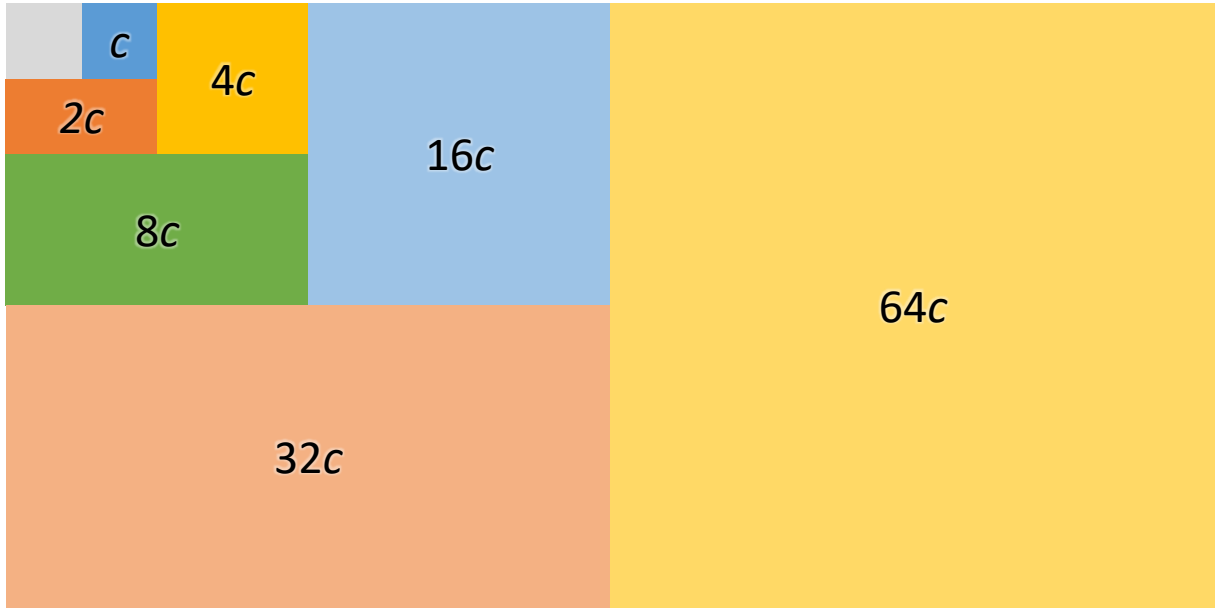
m — reallocation factor

To push back N elements we need $\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$



std::vector

c — initial capacity

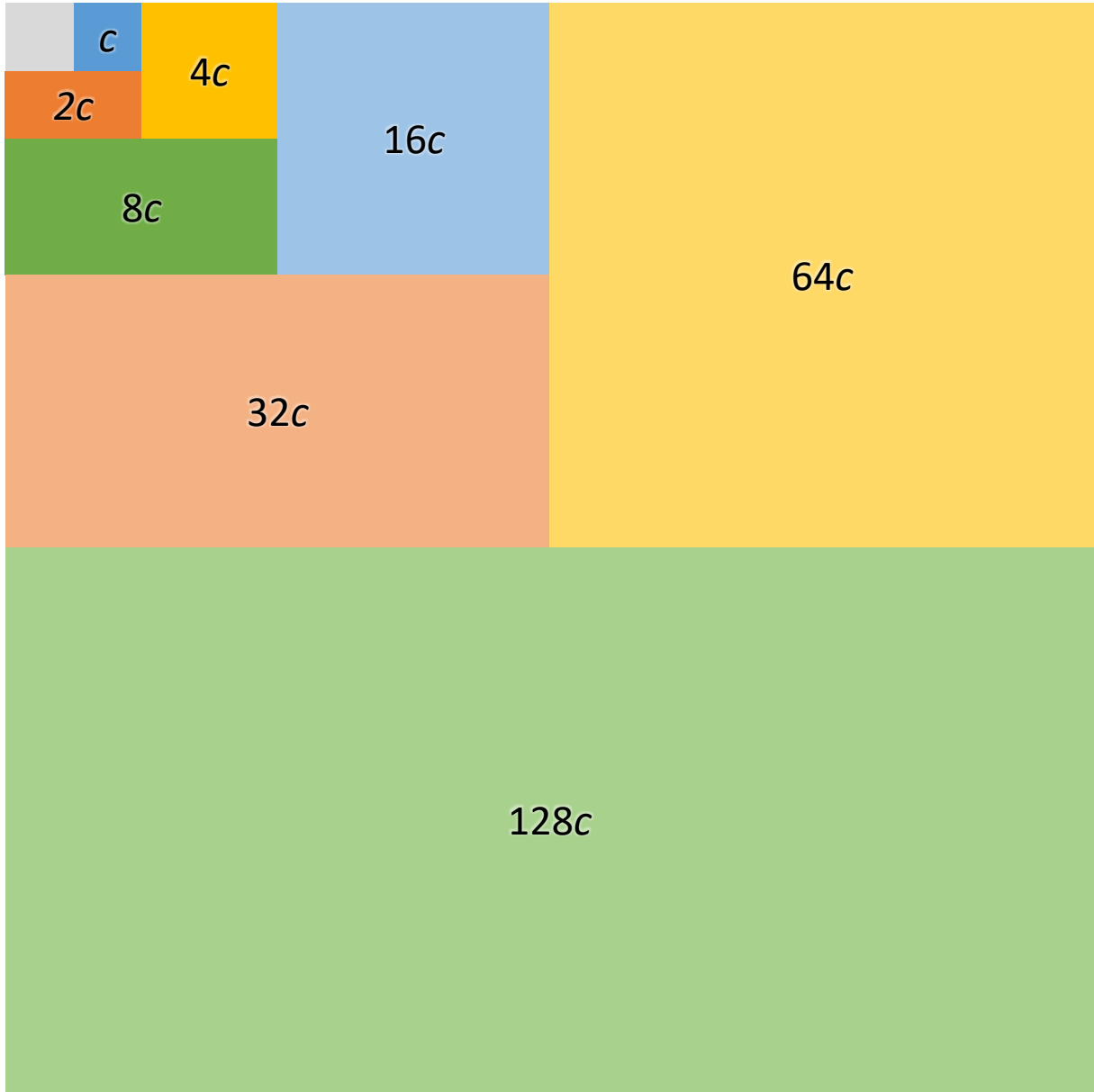
m — reallocation factor

To push back N elements we need $\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$



std::vector

$$N_{elements} = 64c + 1$$

c — initial capacity

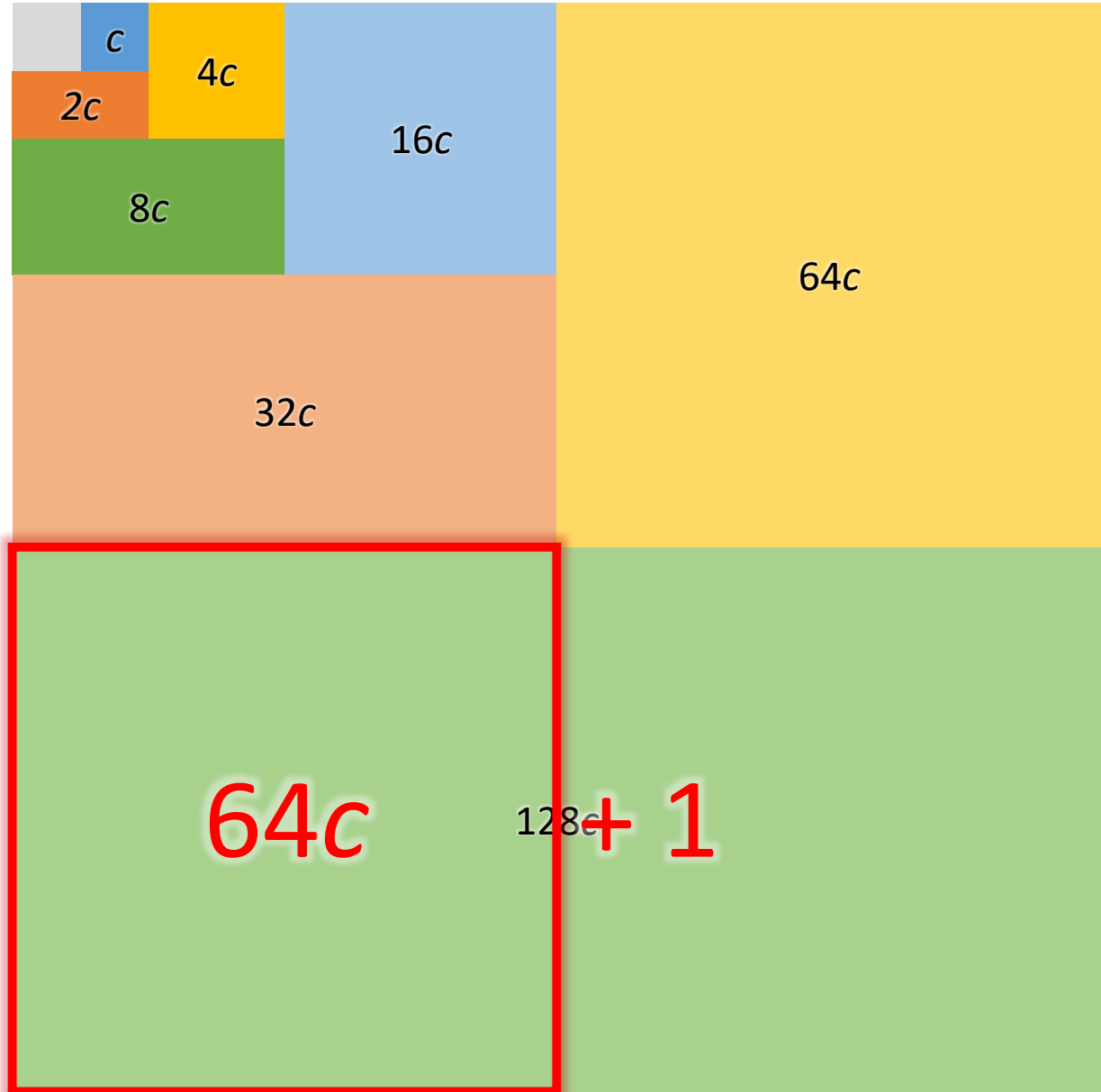
m — reallocation factor

To push back N elements we need $\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

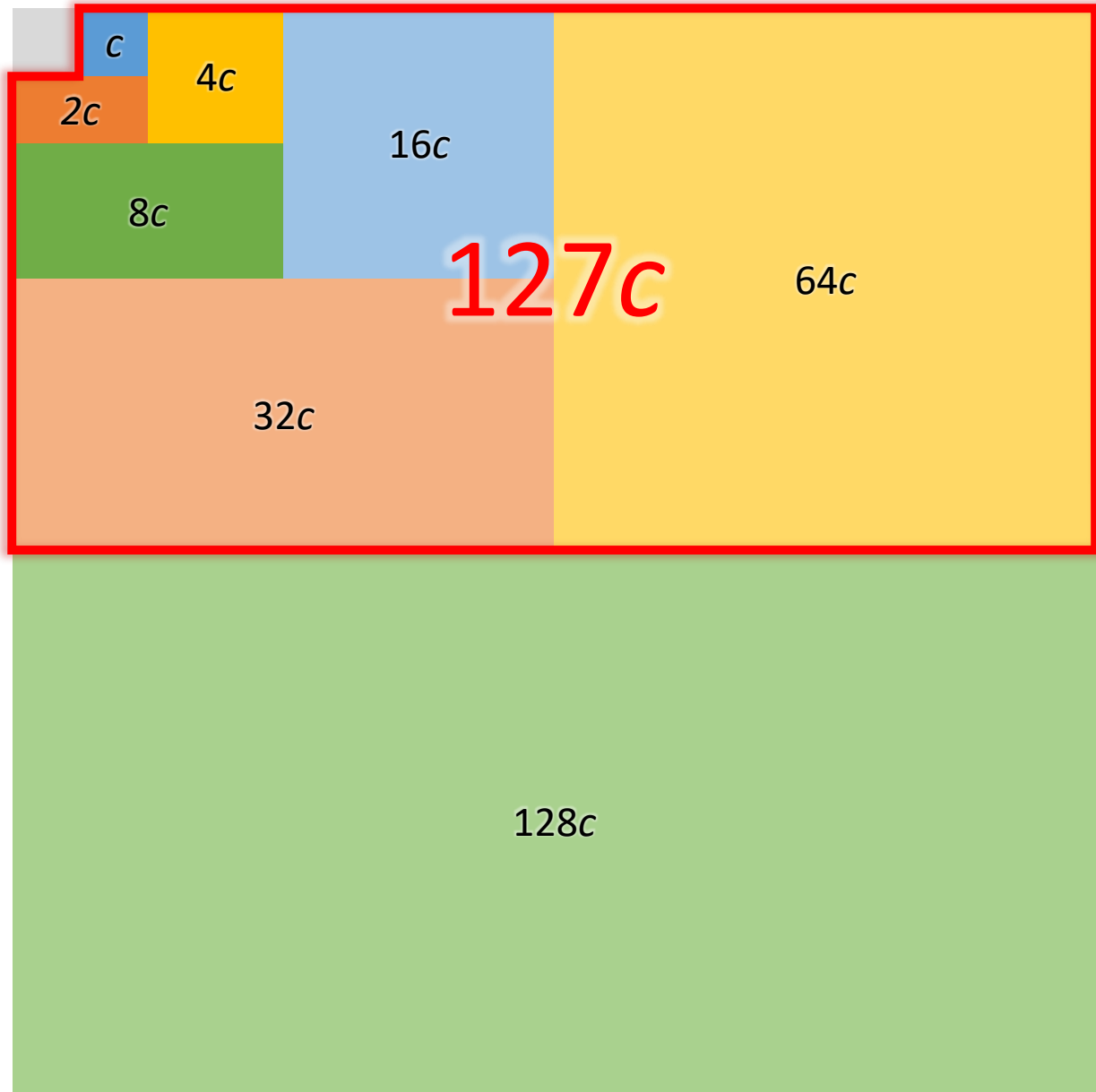
$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$



std::vector

$$N_{elements} = 64c + 1$$
$$N_{moves} = 127c;\ N_{moves}/N_{elements} \approx 2$$



c — initial capacity

m — reallocation factor

To push back N elements we need $\log_m N/c$ reallocations.

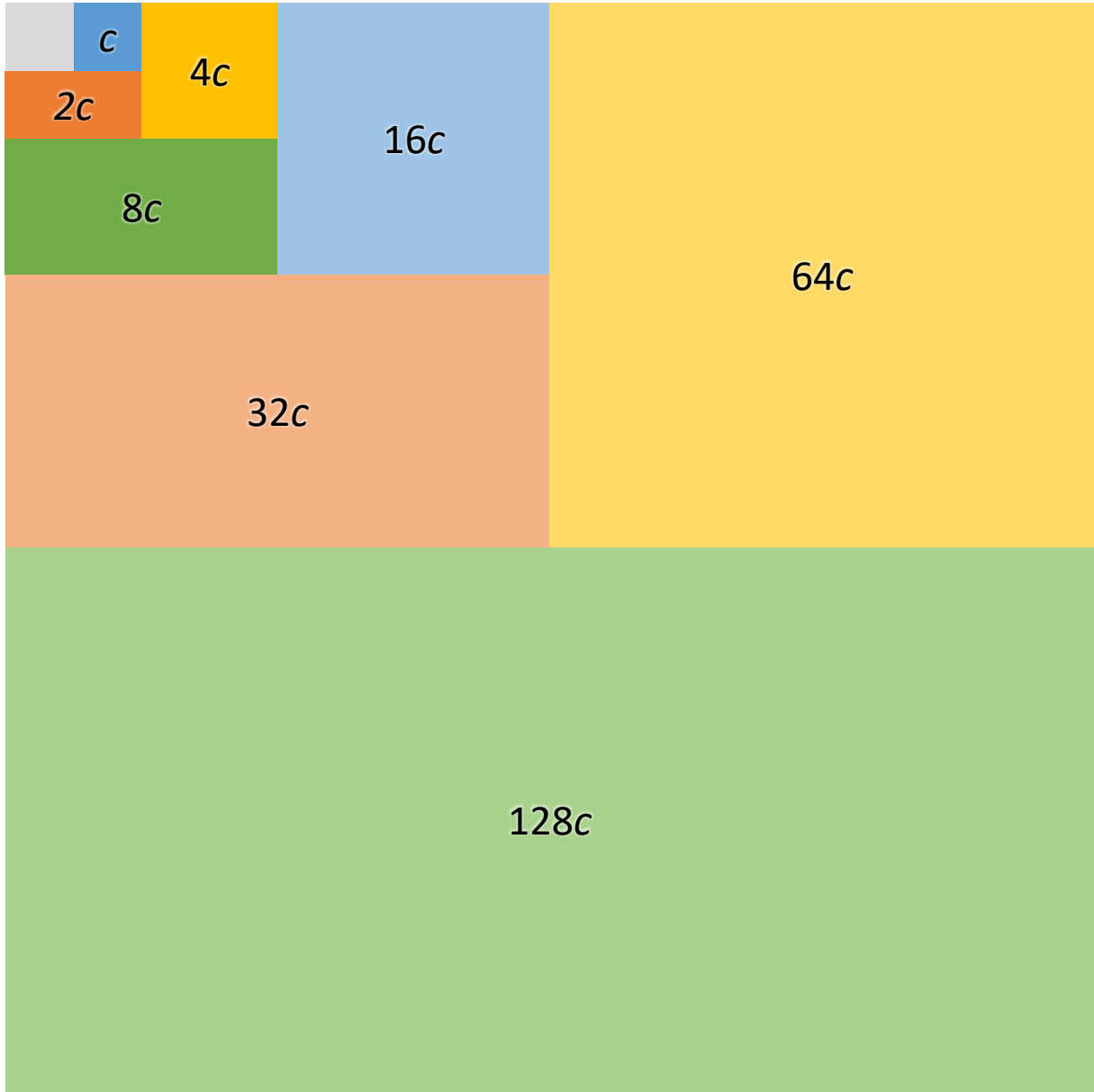
On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

std::vector

$$N_{elements} = 64c + 1$$
$$N_{moves} = 127c; N_{moves}/N_{elements} \approx 2$$
$$N_{allocs} = 8$$



c — initial capacity

m — reallocation factor

To push back N elements we need

$\log_m N/c$ reallocations.

On each reallocation we move $c \cdot m^i$ elements (i — # of reallocation)

$$N_{moves} = \sum_{i=0}^{\log_m N/c} c \cdot m^i = \frac{mN - c}{m - 1}$$

$$\frac{N_{moves}}{N} \approx \frac{m}{m - 1}$$

sum

Browse Examples Surprise Me

Assuming "sum" refers to a computation | Use as referring to a mathematical definition or a character or a general topic or a word instead

Computational Inputs:

» function to sum: c*m^i

» index: i

» lower limit: 0

» upper limit: log(m,N/c)

Compute

Sum:

$$\sum_{i=0}^{\frac{\log\left(\frac{N}{c}\right)}{\log(m)}} c\, m^i = \frac{m\, N - c}{m - 1}$$

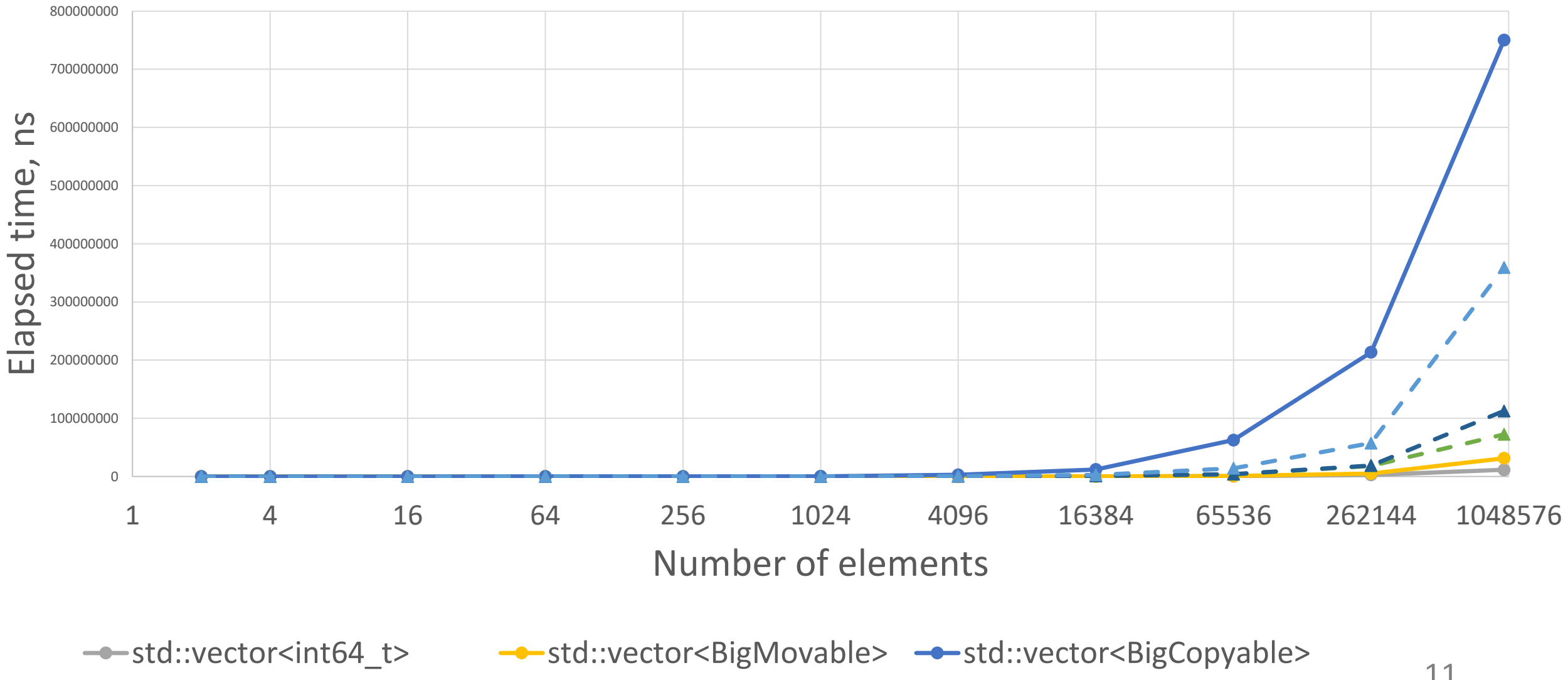
Open code

push_back()

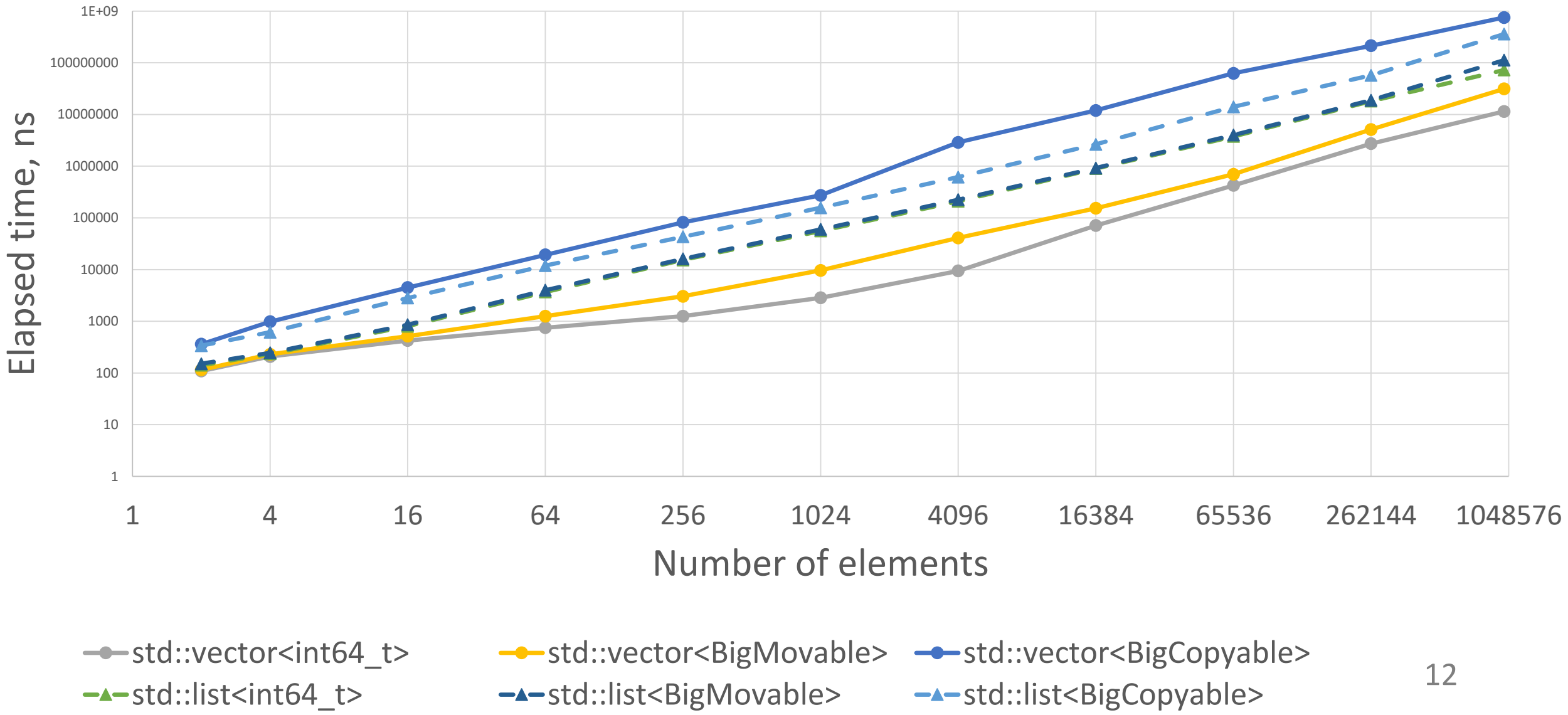
```
template<typename C>
void push_back(benchmark::State &state)
{
  for (auto _ : state) {
    C container;
    for (auto counter = N; counter--;)
      container.push_back(typename C::value_type{});
    benchmark::DoNotOptimize(container);
  }
}
BENCHMARK_TEMPLATE(push_back, std::vector<T>);
BENCHMARK_TEMPLATE(push_back, std::list<T>);
```


C++
C++
std::vector
VS
std::list

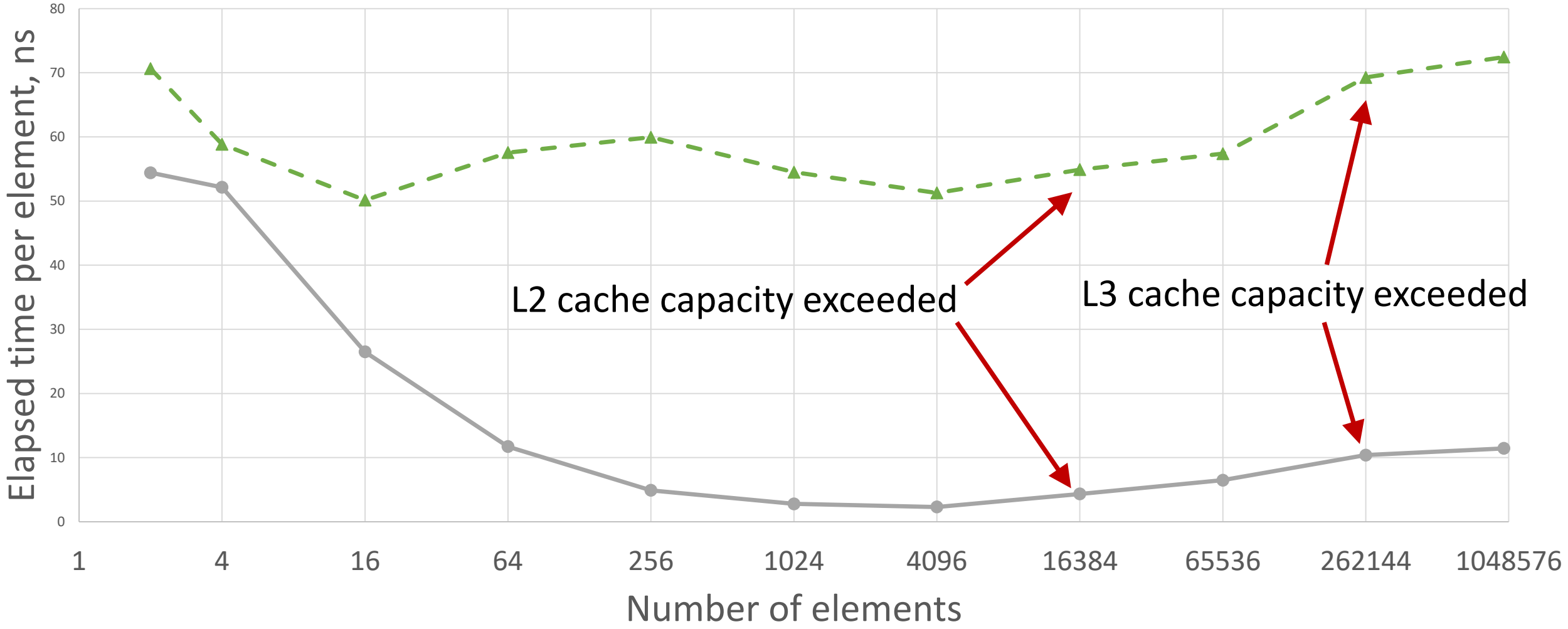
push_back()


800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
Elapsed time, ns
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
Number of elements
std::vector<int64_t>
std::vector<BigMovable>
std::vector<BigCopyable>
std::list<int64_t>
std::list<BigMovable>
std::list<BigCopyable>

push_back()

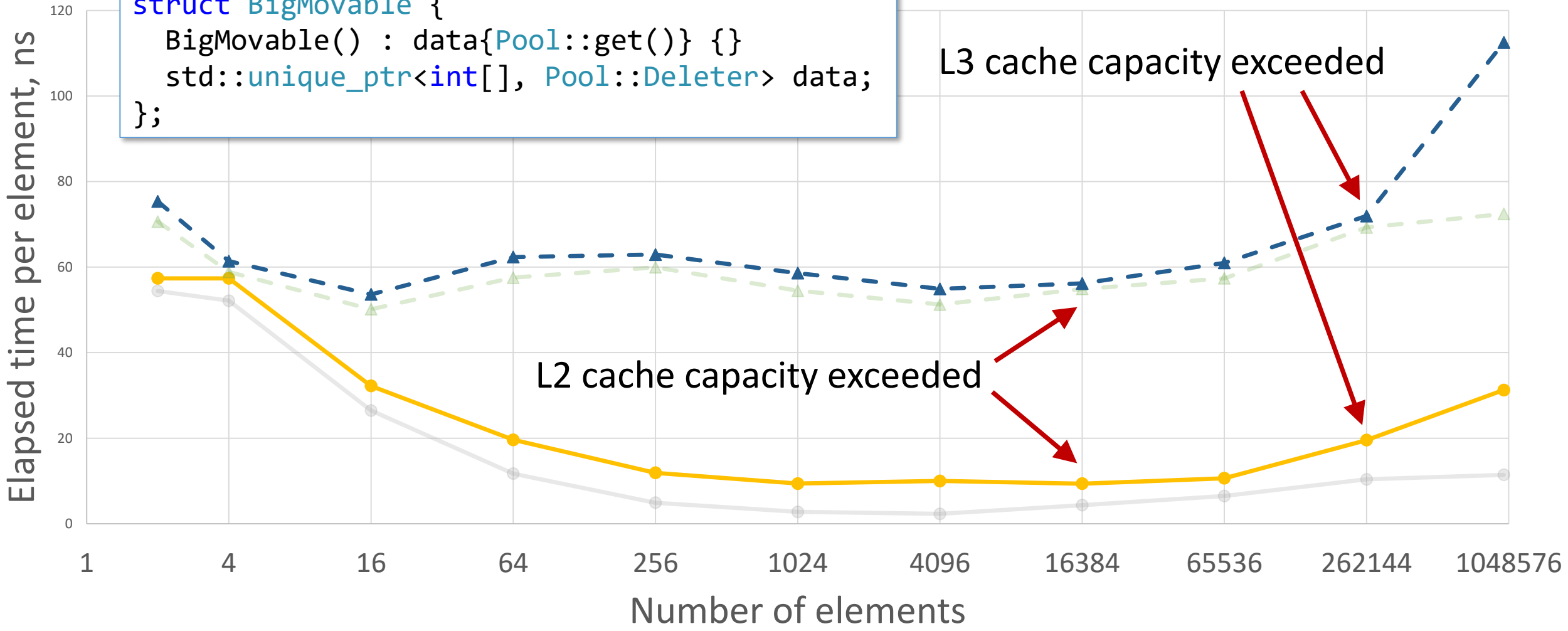

1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Elapsed time, ns
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
Number of elements
std::vector<int64_t>
std::vector<BigMovable>
std::vector<BigCopyable>
std::list<int64_t>
std::list<BigMovable>
std::list<BigCopyable>

push_back()

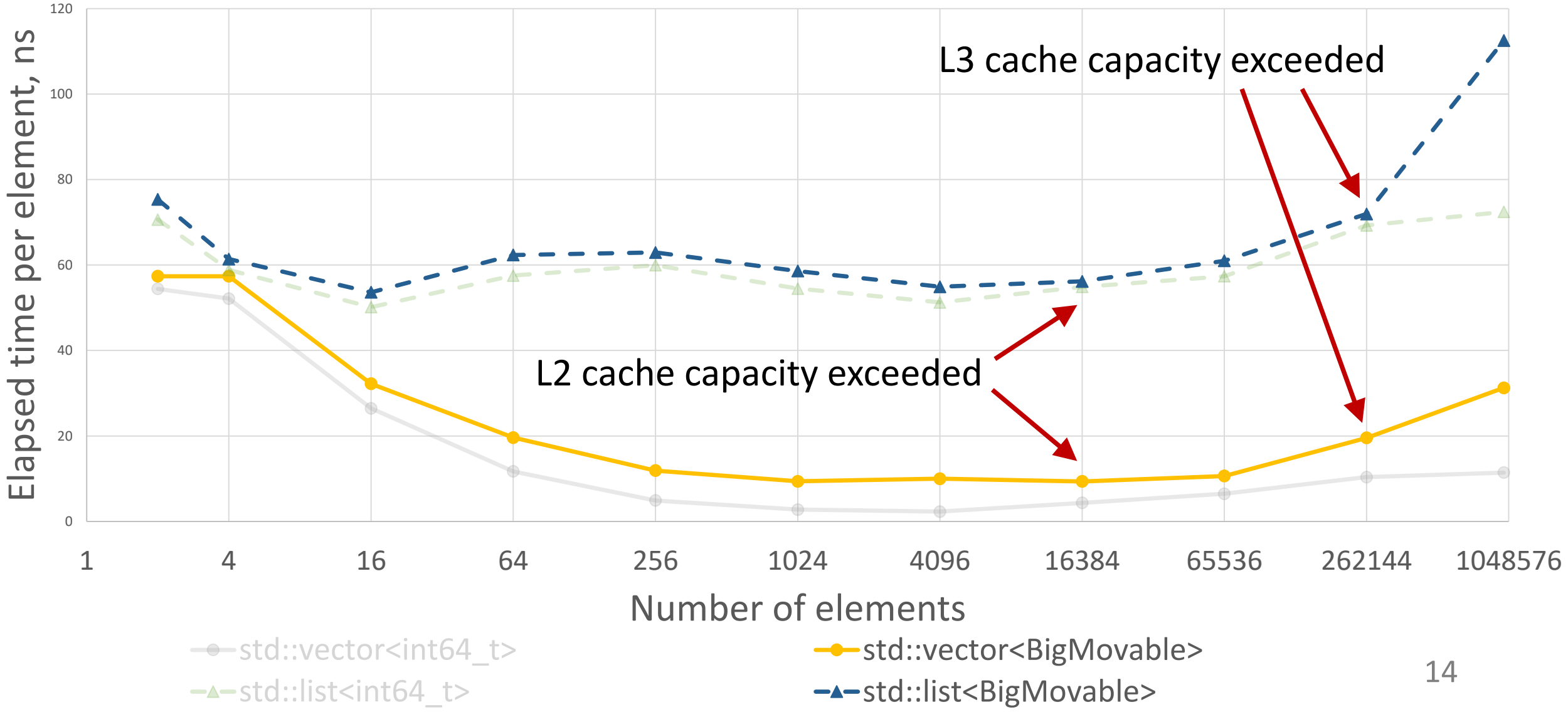

Elapsed time per element, ns
80
70
60
50
40
30
20
10
0
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
Number of elements
L2 cache capacity exceeded
L3 cache capacity exceeded
std::vector<int64_t>
std::list<int64_t>

push_back()

```
struct BigMovable {
  BigMovable() : data{Pool::get()} {}
  std::unique_ptr<int[], Pool::Deleter> data;
};
```

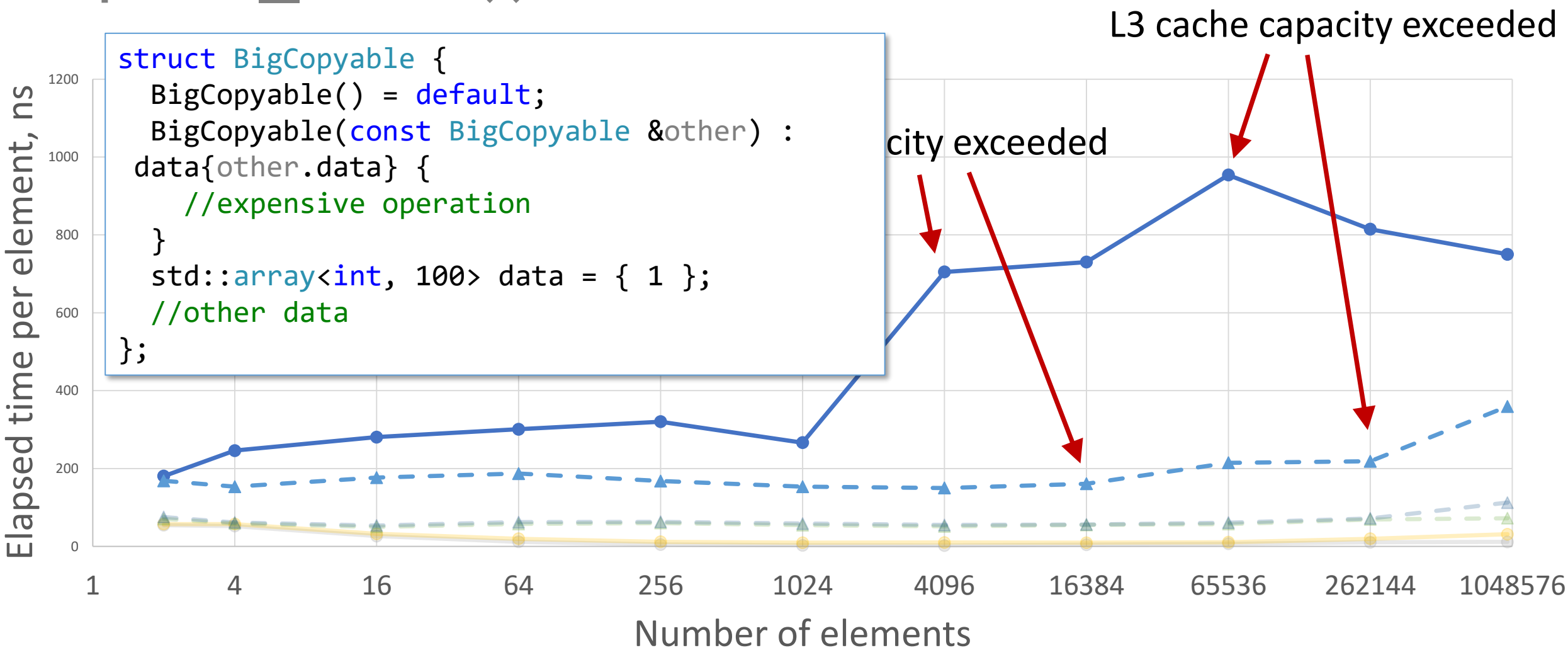


push_back()

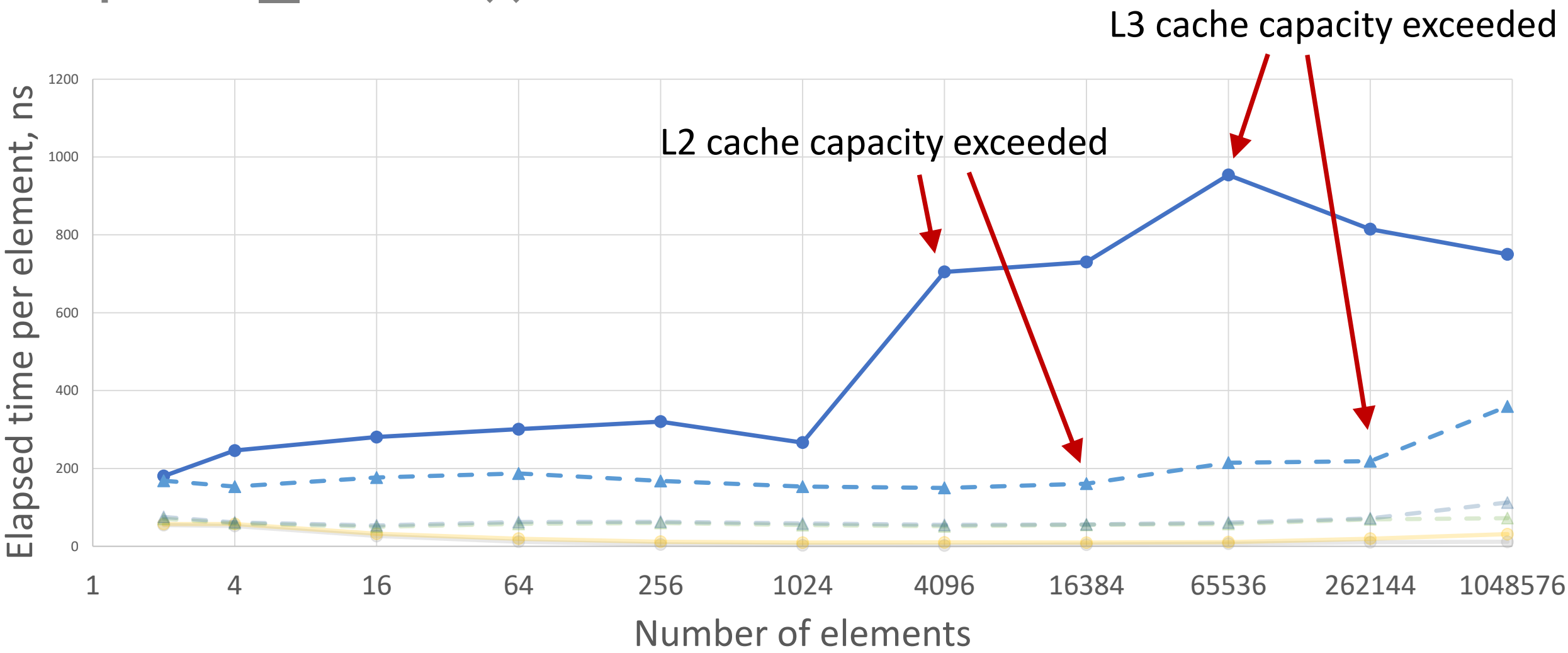

Elapsed time per element, ns
120
100
80
60
40
20
0
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
Number of elements
L3 cache capacity exceeded
L2 cache capacity exceeded
std::vector<int64_t>
std::list<int64_t>
std::vector<BigMovable>
std::list<BigMovable>

push_back()

```
struct BigCopyable {
  BigCopyable() = default;
  BigCopyable(const BigCopyable &other) :
 data{other.data} {
    //expensive operation
  }
  std::array<int, 100> data = { 1 };
  //other data
};
```



push_back()


L3 cache capacity exceeded
L2 cache capacity exceeded
Elapsed time per element, ns
1200
1000
800
600
400
200
0
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
Number of elements
std::vector<int64_t>
std::vector<BigMovable>
std::vector<BigCopyable>
std::list<int64_t>
std::list<BigMovable>
std::list<BigCopyable>

Use `std::vector` by default

Unless

- copy and move are very expensive or unavailable;
- predominantly inserting and removing elements at random positions, splicing.

emplace_back and push_back

```
std::vector<T> v;
const auto t = T{42, true};

v.push_back(t); // copy
v.push_back(T{42, true}); // move

v.emplace_back(42, true); // construct in-place
v.emplace_back(42, true).setAwesome(true); // call method
v.emplace_back(t); // equivalent to push_back(t)
v.emplace_back(T{42, true}); // ~ push_back(T{42, true})
```

```
std::vector<T>:
  void push_back(const T&)
  void push_back(T&&)

  template<typename... Args>
  T &emplace_back(Args&&...)
```

`emplace_back` and `push_back`

`push_back()` calls only *copy* or *move* constructors.

`emplace_back()` calls *any* constructor, including `explicit` constructors.

```
//was std::vector<double> values;
std::vector<std::vector<double>> values;
const double v = getValue();
values.push_back(v); // does not compile
values.emplace_back(v); // ???
```

`emplace_back` and `push_back`

`push_back()` calls only *copy* or *move* constructors.

`emplace_back()` calls *any* constructor, including `explicit` constructors.

```
std::vector<std::unique_ptr<double>> pointers;
pointers.push_back(&v); // does not compile
pointers.emplace_back(&v); // ???
```

`emplace_back` and `push_back`

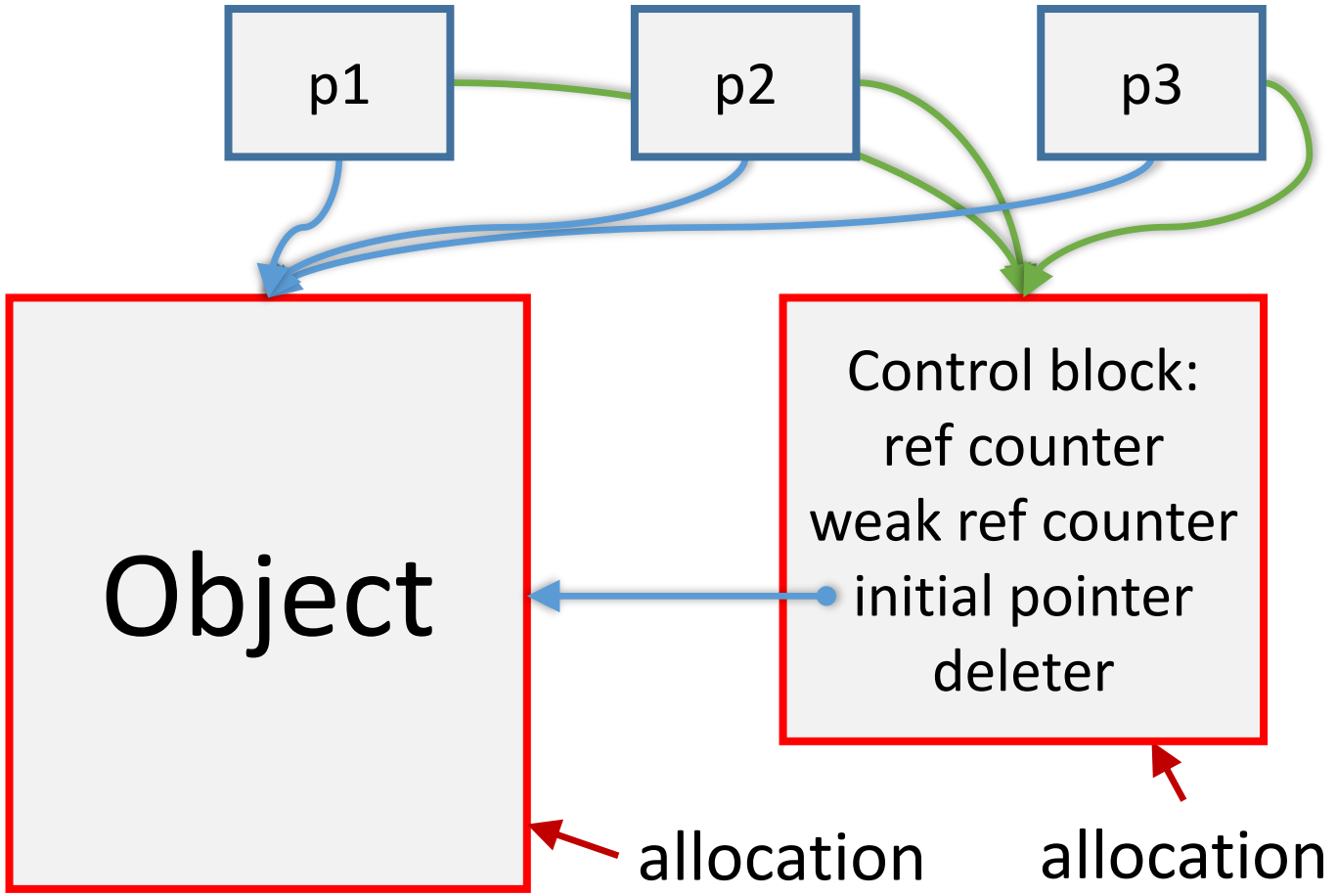
Use emplace methods with multiple arguments.

```
v.emplace_back(42, true); // construct in-place
```

Prefer `push_back()` and `insert()` to ensure correctness when both that and emplace methods will do.
Be cautious that **emplace methods call `explicit` constructors**.

std::shared_ptr

- controls lifetime
- shares ownership

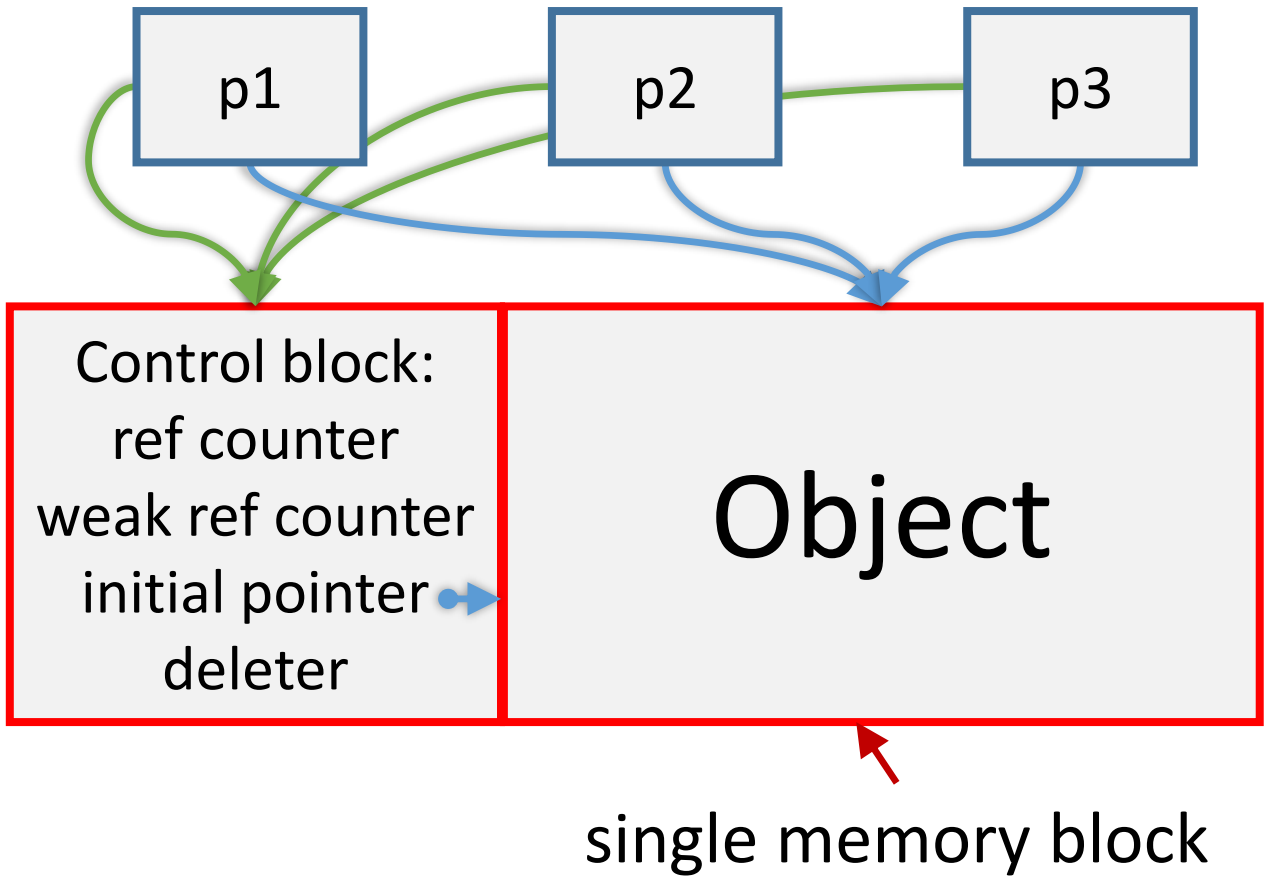



```
auto p = std::shared_ptr<Widget>(new Widget{ par });
```

std::make_shared

Only one memory allocation. 👍

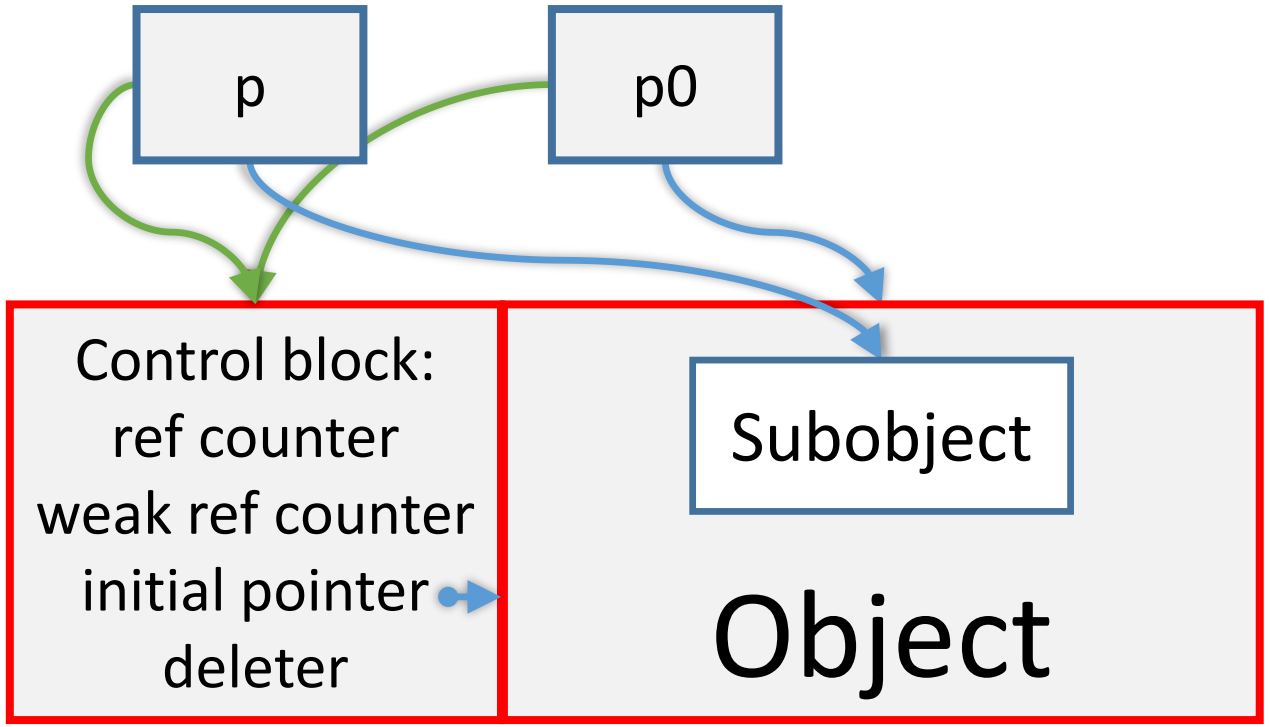
Not possible to use with a custom deleter. 🤔




```
auto p = std::make_shared<Widget>(par);
```

Aliasing constructor

- Allows to share a part of an object
  (or anything, really)




```
auto p0 = std::make_shared<Widget>(par);
auto p = std::shared_ptr<Gadget>(p0, &p0->gadget);
```

aliasing constructor

Aliasing constructor

```
struct Wrapper {
  Widget widget;

  template<typename... Args>
  Wrapper(Args&&... args) : widget{ std::forward<Args>(args)... }
  {}
  ~Wrapper() {
    widget.finalize();
  }
};

auto p0 = std::make_shared<Wrapper>();
auto p = std::shared_ptr<Widget>(p0, &p0->widget);
```

Avoid static objects with non-trivial ctors/dtors

```
const std::string MsgE2BIG = "Argument list too long";
const std::string MsgEACCES = "Permission denied";
const std::string MsgEADDRINUSE = "Address in use";
...
const std::string &getMsgString(int err)
{
  switch (err)
  {
  case E2BIG: return MsgE2BIG;
  case EACCES: return MsgEACCES;
  case EADDRINUSE: return MsgEADDRINUSE;
  ...
```

It’s

Static init in MSVC

```
const std::string MsgE2BIG = "Argument list too long";
```

std::string construction

```
; void `dynamic initializer for 'MsgE2BIG''(void) PROC
sub rsp, 40 ; 00000028H
mov r8d, 22
lea rdx, OFFSET FLAT:`string'
lea rcx, OFFSET FLAT:MsgE2BIG
call std::string::assign(char const * const,size_t)
lea rcx, OFFSET FLAT:void `dynamic atexit destructor for 'MsgE2BIG''(void)
add rsp, 40 ; 00000028H
jmp atexit
```

`strlen()` elision

atexit handler registration

Static init in MSVC: atexit handler

```
; void `dynamic atexit destructor for 'MsgE2BIG''(void) PROC
sub rsp, 40 ; 00000028H
mov rdx, QWORD PTR MsgE2BIG+24
cmp rdx, 16
jb SHORT $LN27@dynamic
mov rcx, QWORD PTR MsgE2BIG
inc rdx
mov rax, rcx
cmp rdx, 4096 ; 00001000H
jb SHORT $LN37@dynamic
mov rcx, QWORD PTR [rcx-8]
add rdx, 39 ; 00000027H
sub rax, rcx
add rax, -8
cmp rax, 31
ja SHORT $LN55@dynamic
$LN37@dynamic:
call void operator delete(void *, size_t)
$LN27@dynamic:
movdqa xmm0, XMMWORD PTR __xmm@0000000000000000f0000000000000000
movdqu XMMWORD PTR MsgE2BIG+16, xmm0
mov BYTE PTR MsgE2BIG, 0
add rsp, 40 ; 00000028H
ret 0
$LN55@dynamic:
call _invalid_parameter_noinfo_noreturn
int 3
$LN53@dynamic:
```

inlined `std::string` destructor

Static init in Clang

```
const std::string MsgE2BIG = "Argument list too long";
```

inlined `std::string` constructor

```
push rax
mov qword ptr [rip + MsgE2BIG[abi:cxx11]], offset MsgE2BIG[abi:cxx11]+16
mov qword ptr [rsp], 22
mov rsi, rsp
mov edi, offset MsgE2BIG[abi:cxx11]
xor edx, edx
call std::string::_M_create(unsigned long&, unsigned long)
mov qword ptr [rip + MsgE2BIG[abi:cxx11]], rax
mov rcx, qword ptr [rsp]
mov qword ptr [rip + MsgE2BIG[abi:cxx11]+16], rcx
movups xmm0, xmmword ptr [rip + .L.str]
movups xmmword ptr [rax], xmm0
movabs rdx, 7453016943536074612
mov qword ptr [rax + 14], rdx
mov qword ptr [rip + MsgE2BIG[abi:cxx11]+8], rcx
mov rax, qword ptr [rip + MsgE2BIG[abi:cxx11]]
mov byte ptr [rax + rcx], 0
mov edi, offset std::string::~string() [base object destructor]
mov esi, offset MsgE2BIG[abi:cxx11]
mov edx, offset __dso_handle
call __cxa_atexit
pop rax
ret
```

`strlen()` elision

copying "Argument list " using SSE instructions

copying "too long" as 64 bit integer

destructor registered as atexit handler

Avoid static objects with non-trivial ctors/dtors

```
const std::string MsgE2BIG = "Argument list too long";
const std::string MsgEACCES = "Permission denied";
const std::string MsgEADDRINUSE = "Address in use";
...
const std::string &getMsgString(int err)
{
  switch (err)
  {
  case E2BIG: return MsgE2BIG;
  case EACCES: return MsgEACCES;
  case EADDRINUSE: return MsgEADDRINUSE;
  ...
```

Avoid static objects with non-trivial ctors/dtors

```
const std::string_view MsgE2BIG = "Argument list too long";
const std::string_view MsgEACCES = "Permission denied";
const std::string_view MsgEADDRINUSE = "Address in use";
...
std::string_view   getMsgString(int err)
{
  switch (err)
  {
  case E2BIG: return MsgE2BIG;
  case EACCES: return MsgEACCES;
  case EADDRINUSE: return MsgEADDRINUSE;
  ...
```

Avoid static objects with non-trivial ctors/dtors

```
const std::string_view MsgE2BIG = "Argument list too long";
const std::string_view MsgEACCES = "Permission denied";
const std::string_view MsgEADDRINUSE = "Address in use";
...
std::string_view   getMsgString(int err)
{
  switch (err)
  {
  case E2BIG: return MsgE2BIG;
  case EACCES: return MsgEACCES;
  case EADDRINUSE: return MsgEADDRINUSE;
  ...
```

Avoid static objects with non-trivial ctors/dtors

For every static `std::string` object you waste

- MSVC: ≈150 bytes of x64 instructions (≈210 bytes in the binary)
- Clang: ≈100 bytes of x64 instructions (≈160 bytes in the binary)

To put in perspective, for ≈80 standard `errno` values
some 12..17 KB are wasted in the binary executable

plus 8..12 KB of code in total run on *every* startup and shutdown
of the process.

Homework: replace all static `std::string`s with
`std::string_view`s or plain arrays.
(`strlen()` elision to the resque!)

Small string optimization

Naïve implementation:

```
struct NaiveString {
  char *data = nullString;
  size_t size = 0;
  size_t capacity = 0;
  static char nullString[];
};
char NaiveString::nullString[] = "";
```

Small string optimization

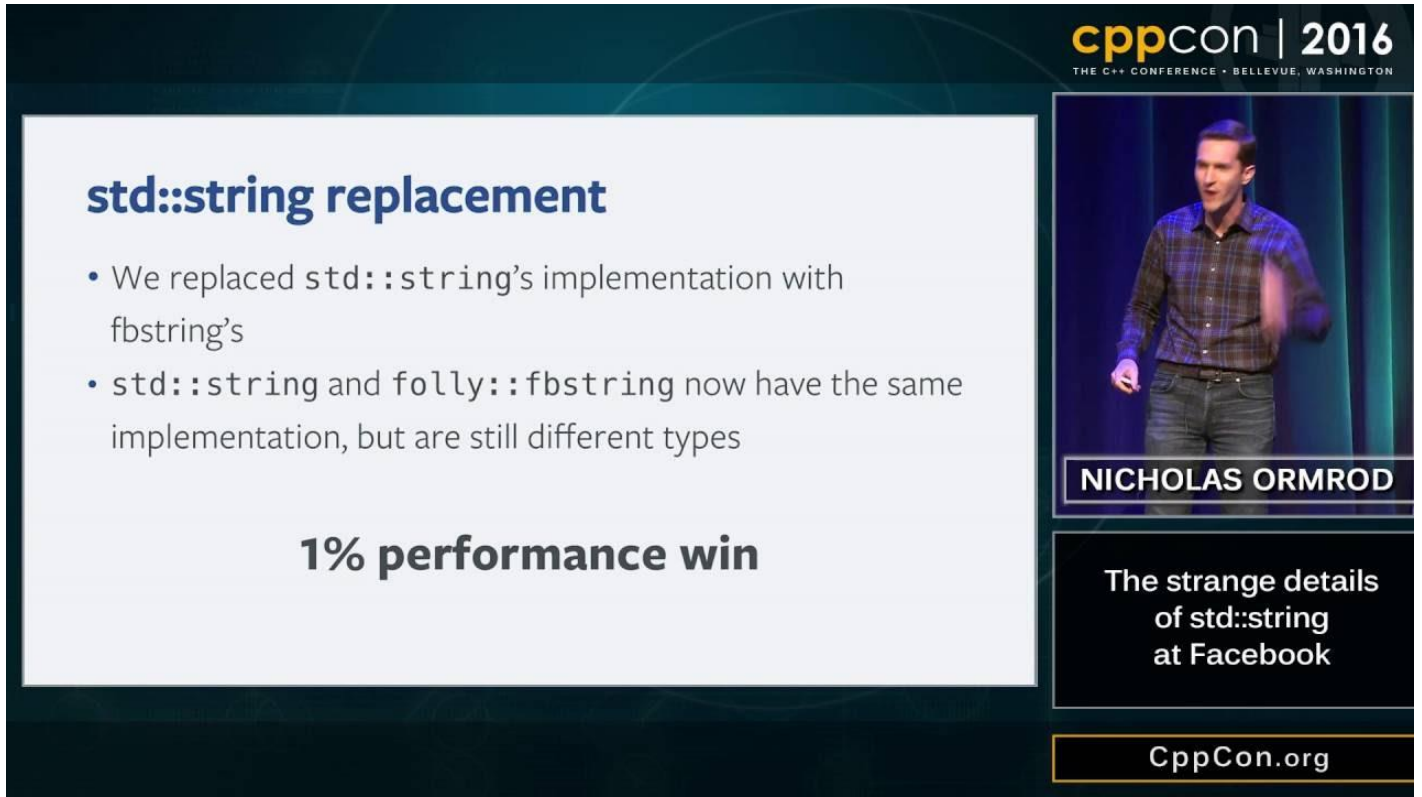
SSO implementation:

```
struct SSOString {
  static const size_t SSO_capacity = 16;
  union {
    char *data;
    char buffer[SSO_capacity];
  };
  size_t size;
  size_t capacity;
  bool is_sso_string() const {
    return capacity <= SSO_capacity;
  }
};
```

Small string optimization

Go watch a great overview of different SSO approaches!

https://youtu.be/kPR8h4-qZdk

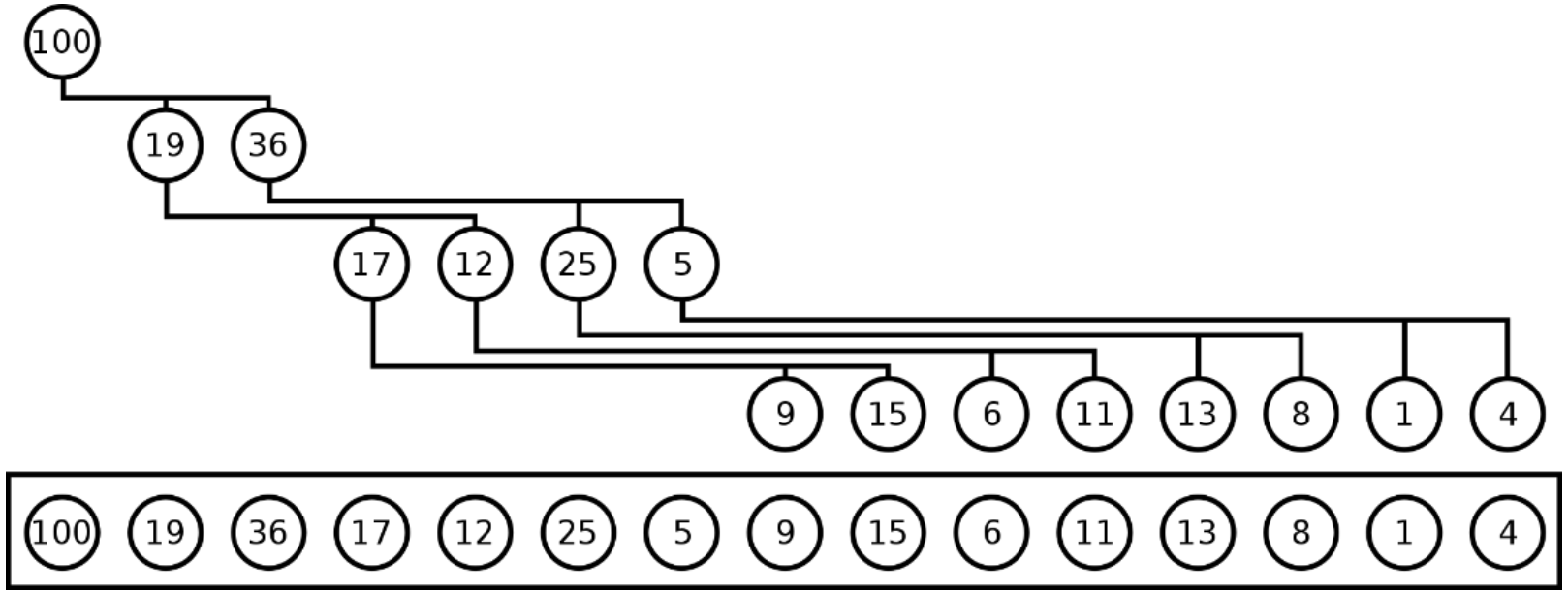


Heap algorithms

Largest element is at the top for *max heap*.

Smallest element is at the top for *min heap*.

Sorted vector *is also a heap*.

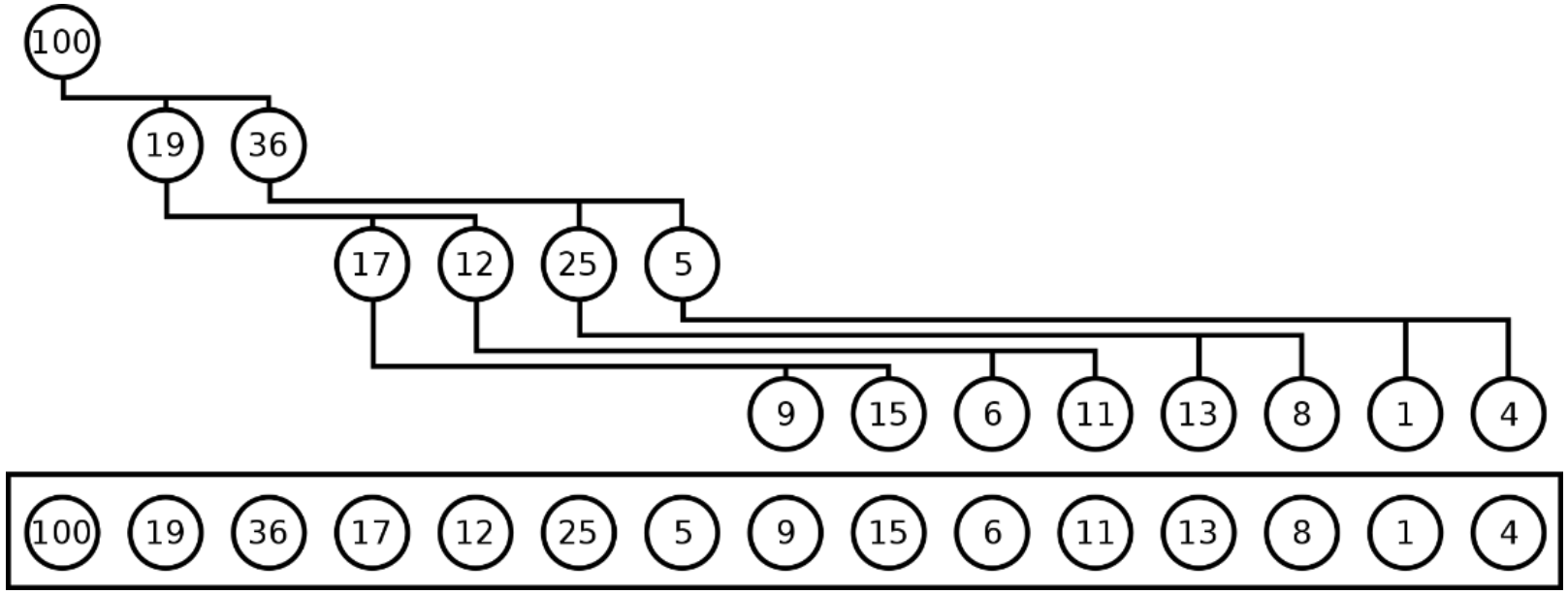


Heap algorithms

Can be built in $\mathrm{O}(N)$ time (**`std::make_heap`**).

Insertion of an element (**`std::push_heap`**) and
removal of the top element (**`std::pop_heap`**) is $\mathrm{O}(\log N)$.

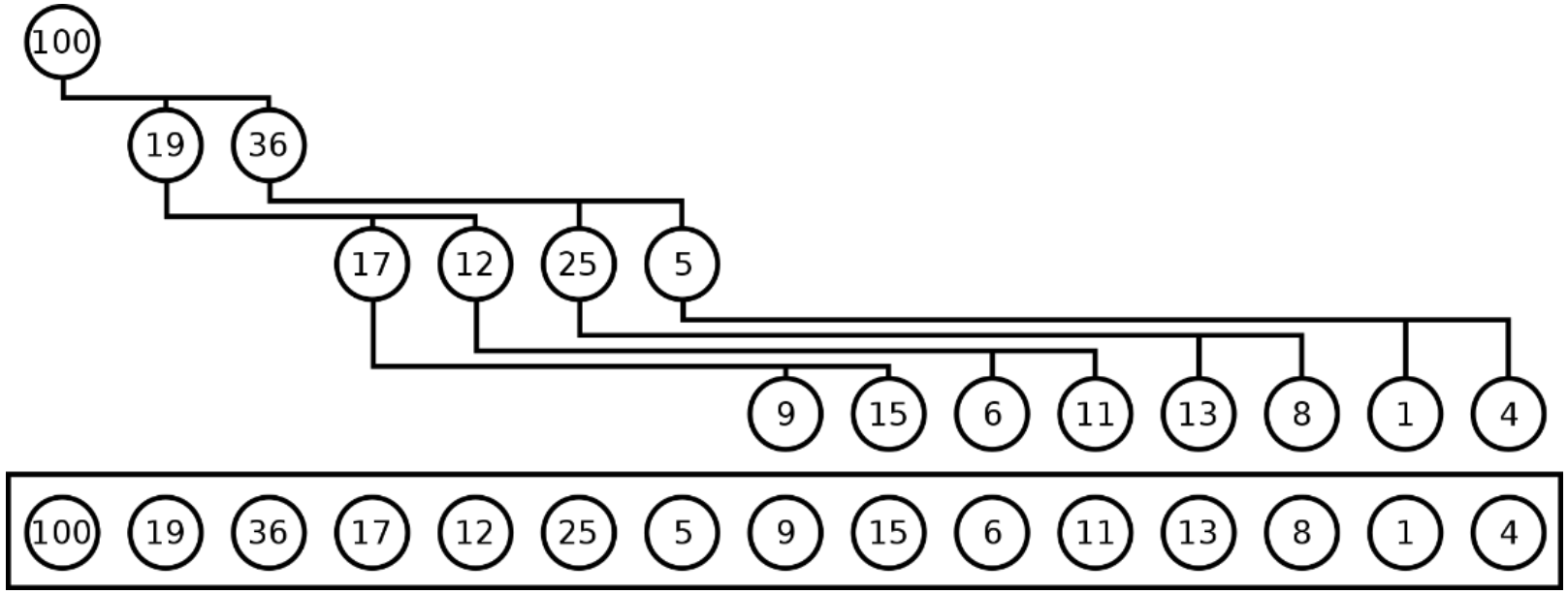
Sorting (**`std::sort_heap`**) is $\mathrm{O}(N \log N)$.



Heap algorithms

`std::priority_queue` is a container adaptor

that wraps heap algorithms in a convenient interface.



Sorting

State of the art sort algorithms:

Insertion sort: $\mathrm{O}(N^2)$ on average, **fastest on small sets** in practice.

Quicksort: $\mathrm{O}(N \log N)$ on average, $\mathrm{O}(N^2)$ in the worst case, **fastest on average** in practice.

Heapsort: $\mathrm{O}(N \log N)$ **in all cases**, slower than quicksort in practice.

Big O notation

$f(x) = \mathrm{O}\big(g(x)\big)$

if and only if for some positive m
$|f(x)| \leq m \cdot g(x)$

for all $x \geq x_0$.

Properies

$f(n) = 9\log n + 5(\log n)^4 + 3n^2 + 2n^3 = \mathrm{O}(n^3)$ — defined by the fastest growing term.

$k \cdot f(x) = \mathrm{O}\big(k \cdot g(x)\big) = \mathrm{O}\big(g(x)\big)$, where k is constant

Sorting

State of the art sort algorithms:

Insertion sort: $\mathrm{O}(N^2)$ on average, **fastest on small sets** in practice.

Quicksort: $\mathrm{O}(N \log N)$ on average, $\mathrm{O}(N^2)$ in the worst case, **fastest on average** in practice.

Heapsort: $\mathrm{O}(N \log N)$ **in all cases**, slower than quicksort in practice.

How to guarantee $\mathrm{O}(N \log N)$ complexity in the worst case?

How to employ speed of quicksort avoiding the worst case?

Sorting

Enter **introsort**.

Best of two worlds:

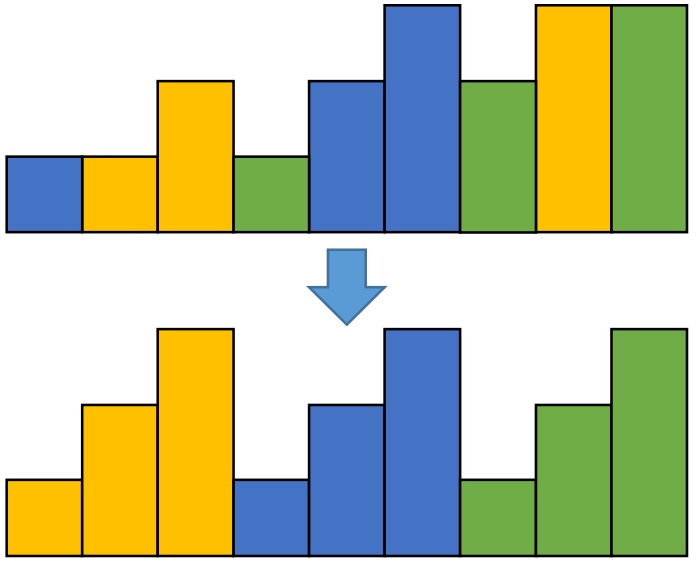
- speed of quicksort on average and
- $\mathrm{O}(N \log N)$ complexity in the worst case.

Recipe:

- do at most $m \log N$ iterations of quicksort,
- if it's not sorted yet, do heapsort on unsorted parts.

m is a small constant, say 2.

Stable sorting



- preserves order of equivalent elements,
- complexity is $\mathrm{O}(N \log^2 N)$ or $\mathrm{O}(N \log N)$ if there is enough extra space.

`stable_sort` under the hood:

insertion sort of small chunks and

merge sort in $\mathrm{O}(N \log N)$ and $\mathrm{O}(N)$ space or

inplace merge sort in $\mathrm{O}(N \log^2 N)$ time if there isn't enough space,

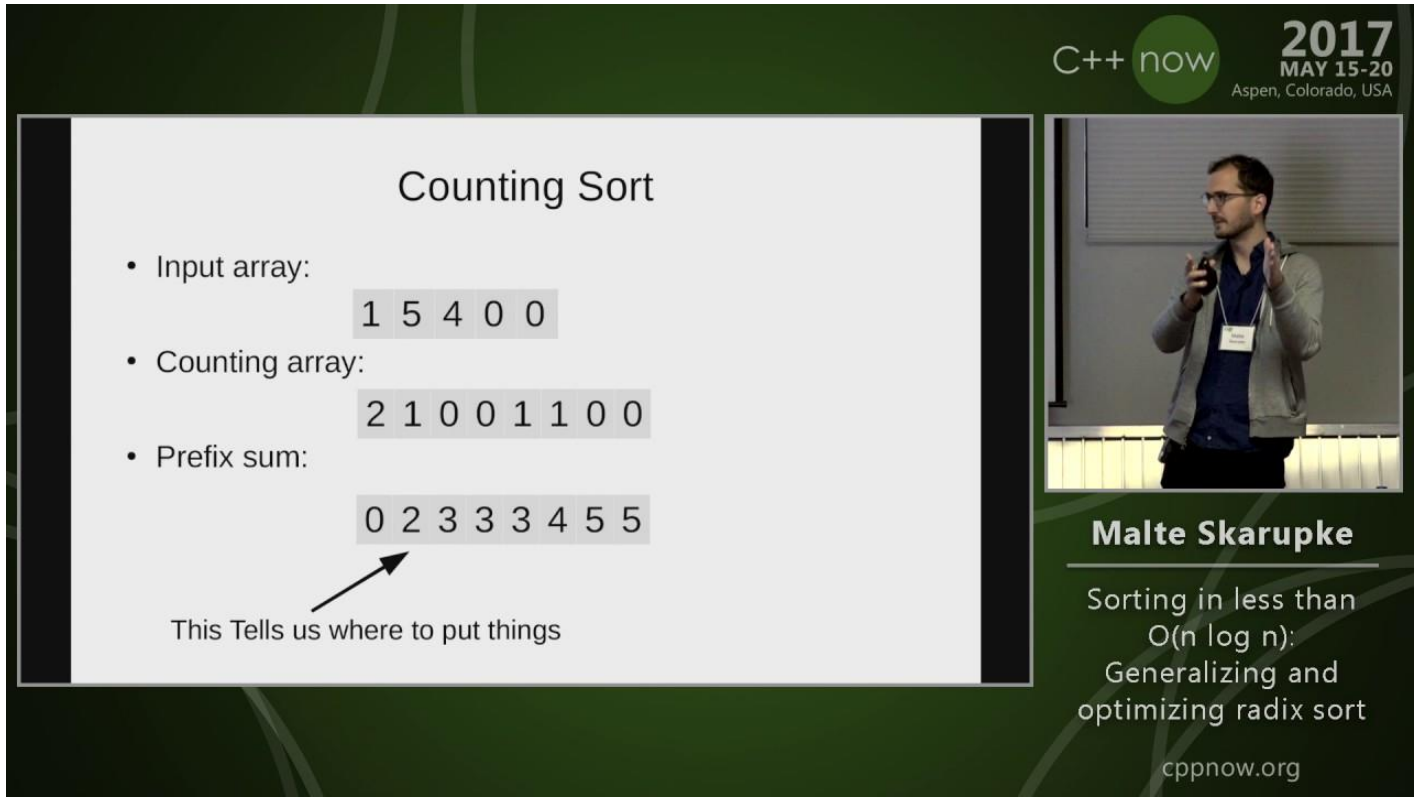
all are stable.

Can we sort faster than $\mathbf{O}(N \log N)$?

Can we sort faster than $O(N \log N)$?

Radix sort!

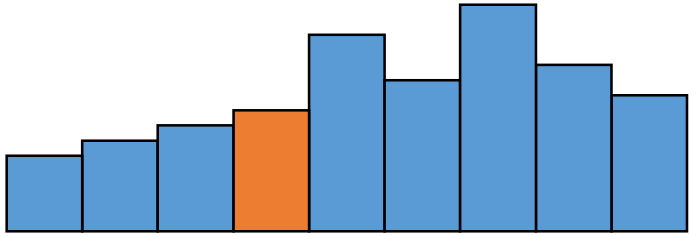
https://youtu.be/zqs87a_7zxw



Sorting and selection

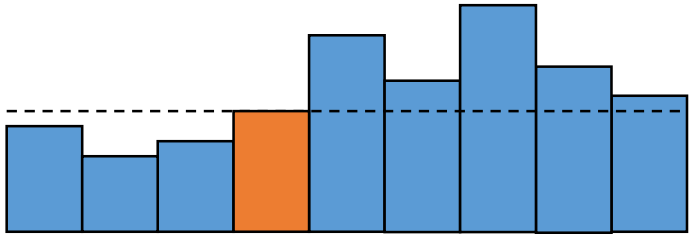
`partial_sort`: "approximately" $\mathrm{O}(N \log k)$.

k first elements will be sorted.



`nth_element`: $\mathrm{O}(N)$ "on average".

Independent of k.



Should we use `partial_sort` or `nth_element` + sort all the time?

What are the use cases?

Sorting and selection

`partial_sort`:

build a heap of the first k elements in $\mathrm{O}(k)$
+ insert the rest $(N - k)$ elements into the heap in $\mathrm{O}(N \log k)$
+ sort heap in $\mathrm{O}(k \log k)$,

which results in $\mathrm{O}(N \log k)$ complexity overall.

`nth_element`:

introselect in $\mathrm{O}(N)$ (quickselect + heapselect to avoid worst case)

`nth_element` + `sort`: $\mathrm{O}(N) + \mathrm{O}(k \log k)$.

For both approaches:

$\mathrm{O}(N)$ if k is a small constant,

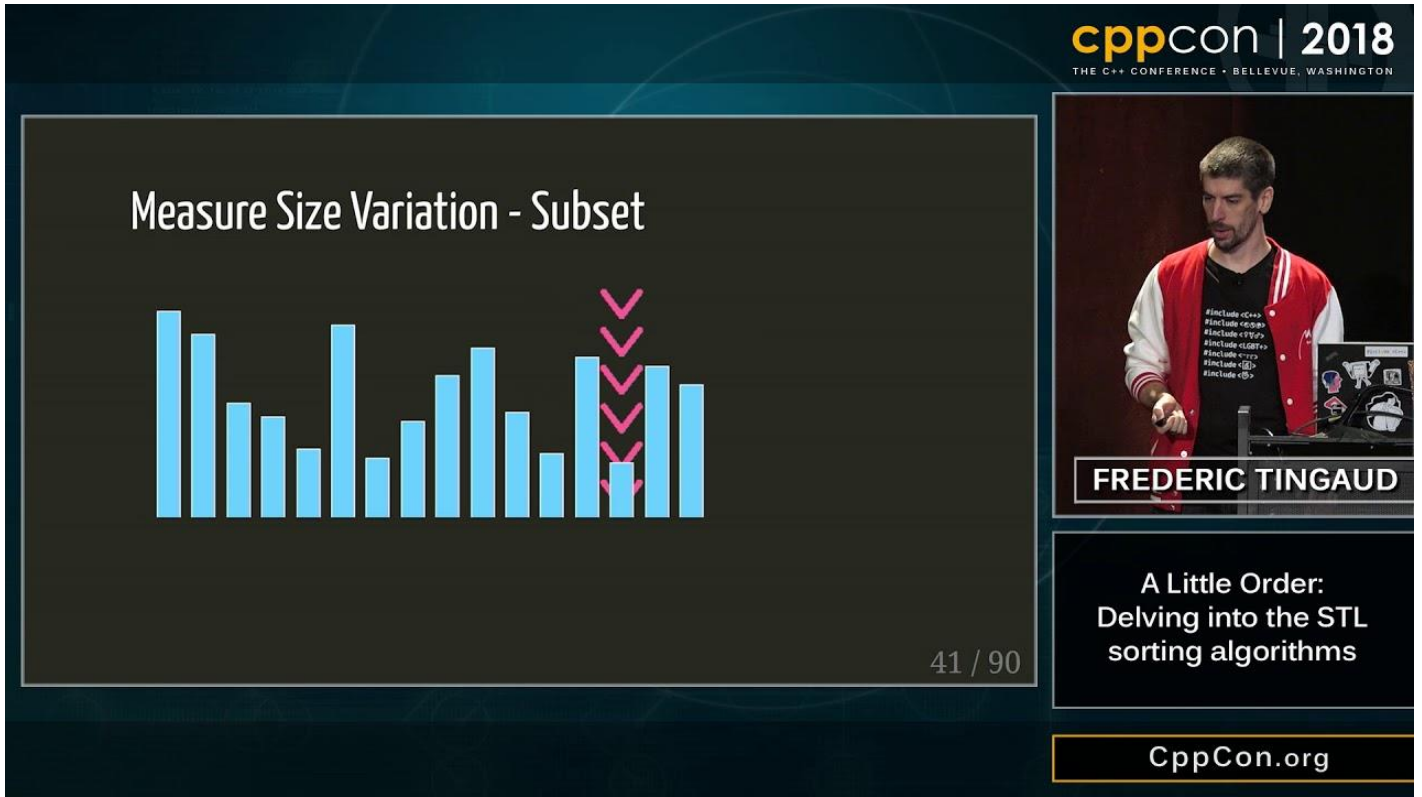
$\mathrm{O}(N \log mN) = \mathrm{O}(N) + \mathrm{O}(mN \log mN) = \mathrm{O}(N \log N)$ if k is some fraction mN.

Sorting and selection

In practice for small constant k `partial_sort` is *faster* than `nth_element` + sort.

However for $k \gg 1$ and $N \gg 1$ `partial_sort` is significantly slower.

https://youtu.be/-0tO3Eni2uo



Associative containers

	insertion/deletion	lookup	
`unordered_map` `unordered_set`	$O(1)$ on average		iteration is unordered, hash function must be provided
`map` `set`	$O(\log N)$		iteration is ordered, specified by predicate
flat map/set aka sorted vector	$O(N)$	$O(\log N)$	

flat map

```
template<typename Key, typename Value>
struct FlatMap : std::vector<std::pair<Key, Value>>
{
  Value &operator[](const Key &key);
  auto find(const Key &key) const;

private:
  struct Less;
};
```

flat map

```
Value &operator[](const Key &key) {
  const auto i = std::lower_bound(this->begin(), this->end(),
                                  key,
                                  Less{});
  if (i == this->end()) {
    this->emplace_back(key, Value{});
    return this->back().second;
  }
  if (key < i->first)
    return this->emplace(i, key, Value{})->second;
  return i->second;
}
```

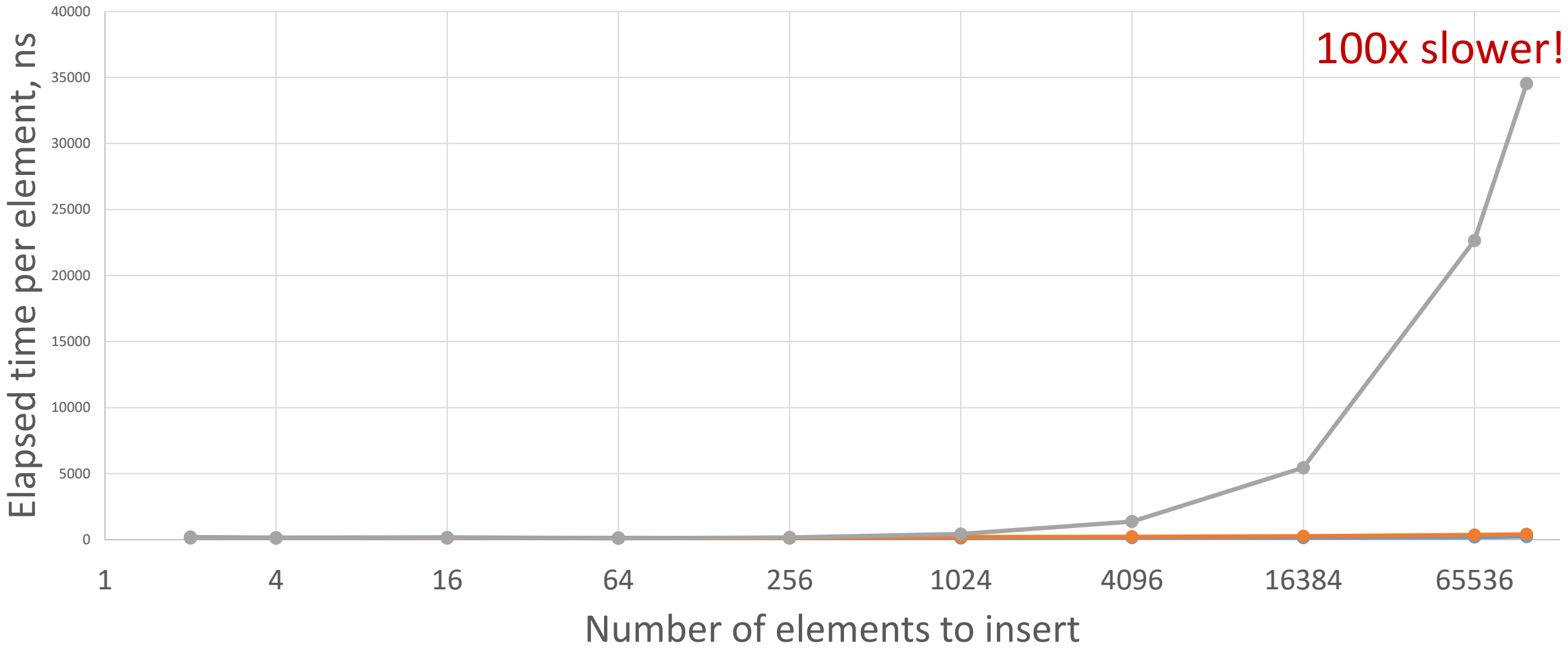
flat map

```
auto find(const Key &key) const {
  const auto i = std::lower_bound(this->begin(), this->end(),
                                  key,
                                  Less{});
  return key < i->first ? this->end() : i;
}
```

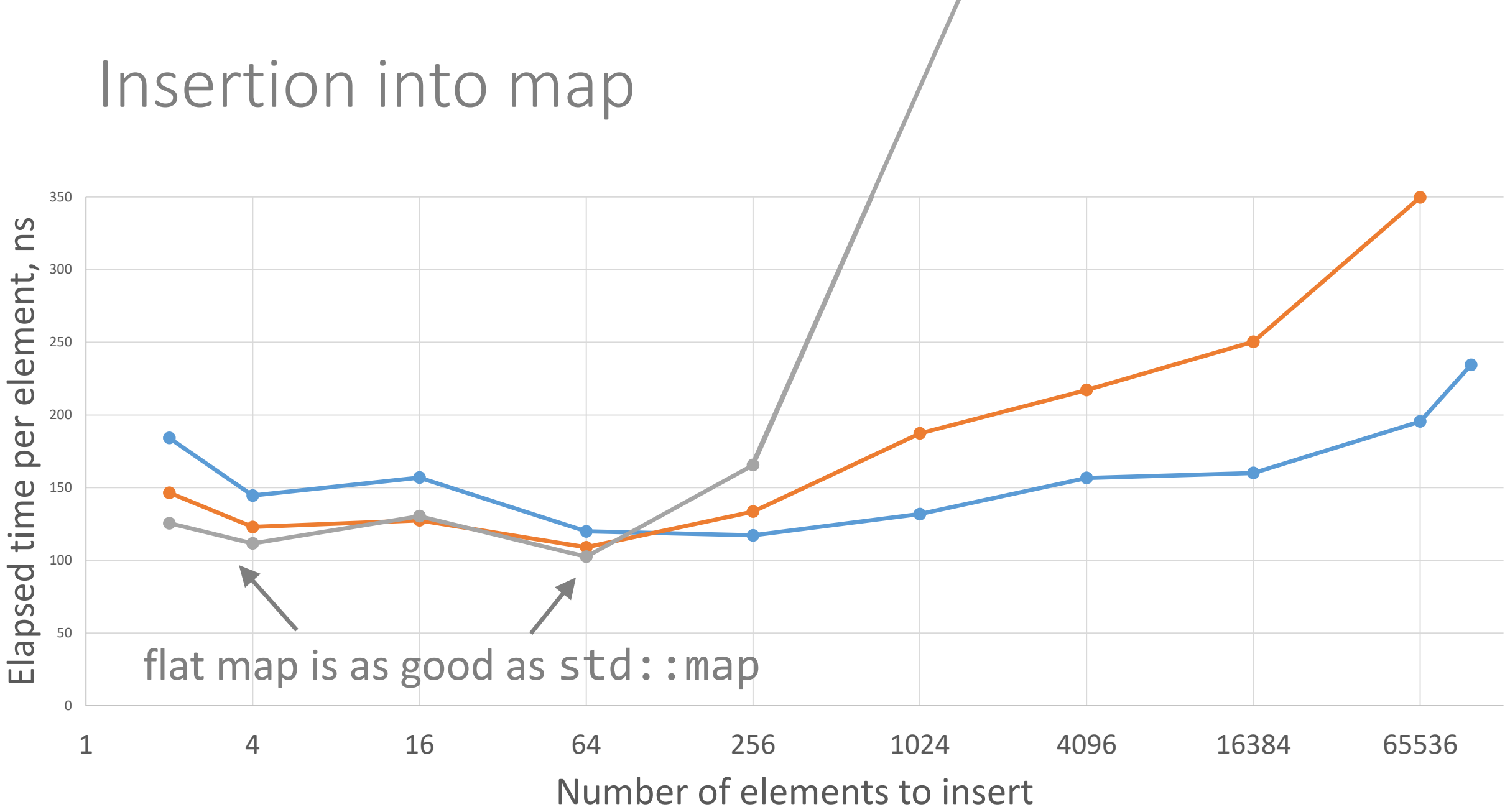
flat map

```
struct Less
{
  template<typename A, typename B>
  bool operator()(const A &a, const B &b) {
    if constexpr (std::is_same_v<A, Key>)
      return a < b.first;
    else if constexpr (std::is_same_v<B, Key>)
      return a.first < b;
    else
      return a.first < b.first;
  }
};
```

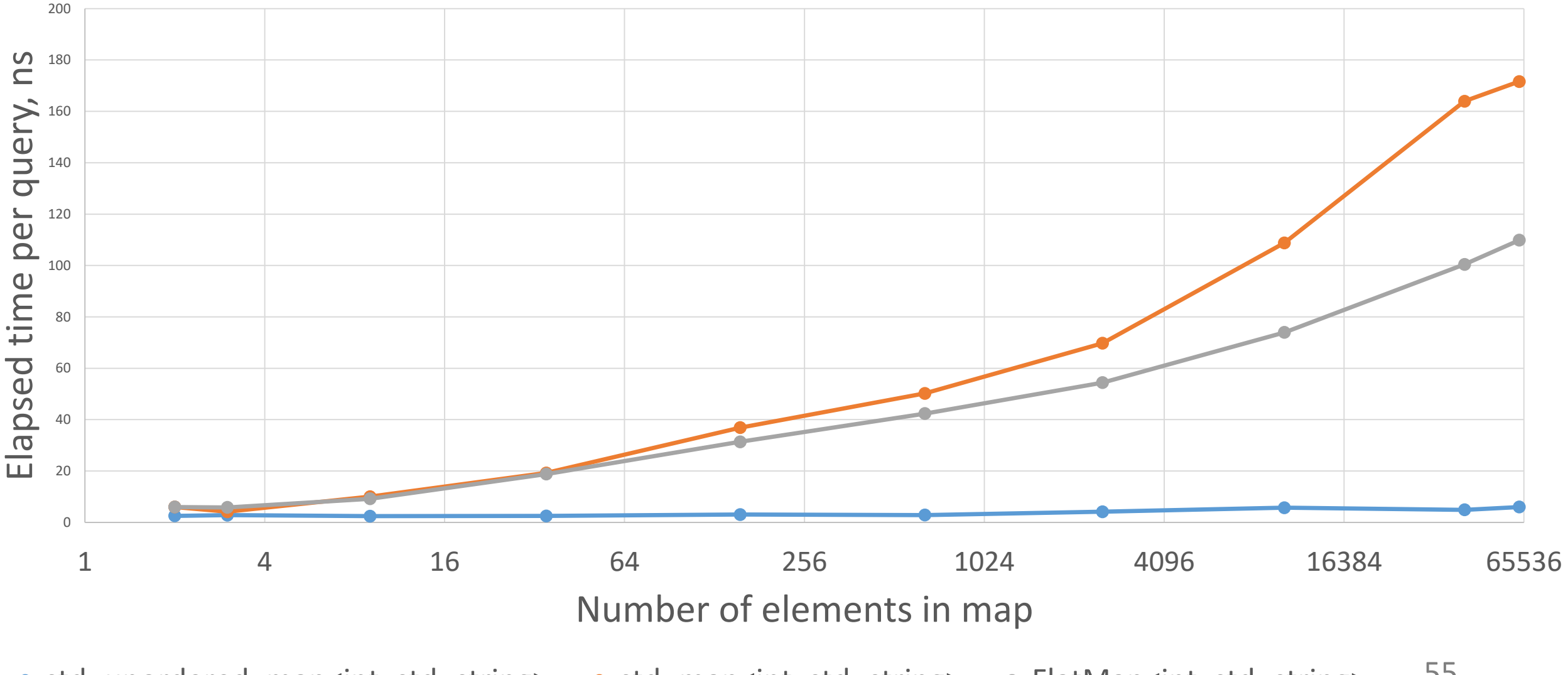
Insertion into map



Insertion into map



Querying map (`Map::find`)



- Use **unordered** associative containers by default

  - Use ordered associative containers if you frequently need to iterate over elements in order. Duh!
  - Use flat map only for small **constant** sizes or as constant one-time initialized map.

- Use **unordered** associative containers by default

  - Use ordered associative containers if you frequently need to iterate over elements in order. Duh!
  - Use flat map only for small **constant** sizes or as constant one-time initialized map.

What if we want to iterate over elements in order only once in a while?

Adding elements to a map & iterating in order

Adding N elements to `std::map`:

$$\mathrm{O}\left(\sum_{i=1}^{N} \log i\right) = \mathrm{O}\big(N \log N - N + \mathrm{O}(\log N)\big) = \mathrm{O}(N \log N)$$

$$\lfloor u \rfloor = u + \mathrm{O}(1)$$

Abel's summation formula

$$\sum_{i=1}^{N} 1 \cdot \log i = N \log N - \int_{1}^{N} \lfloor u \rfloor \cdot (\log u)' du = N \log N - \int_{1}^{N} \frac{\lfloor u \rfloor}{u} du =$$

$$= N \log N - N + 1 + \mathrm{O}\left(\int_{1}^{N} \frac{1}{u} du\right) = N \log N - N + \mathrm{O}(\log N)$$

Adding elements to a map & iterating in order

Adding N elements to `std::map`: $\mathrm{O}(N \log N)$

$+\mathrm{O}(N)$ iteration $= \mathrm{O}(N \log N)$ overall

Adding N elements to `std::unordered_map`: $\mathrm{O}(N)$ ($\mathrm{O}(1)$ per element on average)

$+\mathrm{O}(N)$ copying to a vector

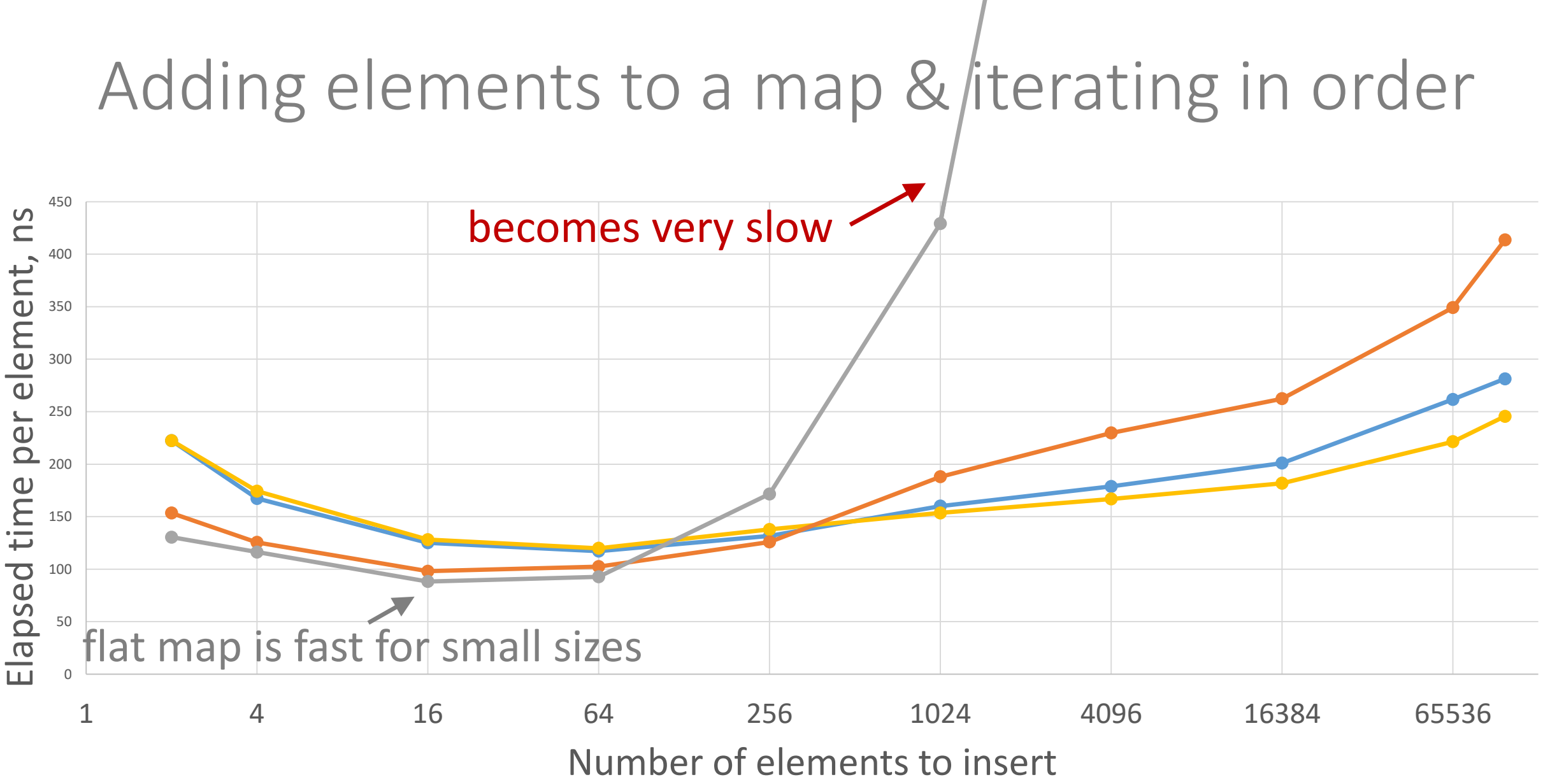
$+\mathrm{O}(N \log N)$ sorting

$+\mathrm{O}(N)$ iteration

$= \mathrm{O}(N \log N)$ overall

Is `std::unordered_map` approach faster than `std::map`?

Adding elements to a map & iterating in order



Big O notation

$f(x) = \mathrm{O}\big(g(x)\big)$

if and only if for some positive m
$|f(x)| \leq m \cdot g(x)$

for all $x \geq x_0$.

Properies

$f(n) = 9\log n + 5(\log n)^4 + 3n^2 + 2n^3 = \mathrm{O}(n^3)$ — defined by the fastest growing term.

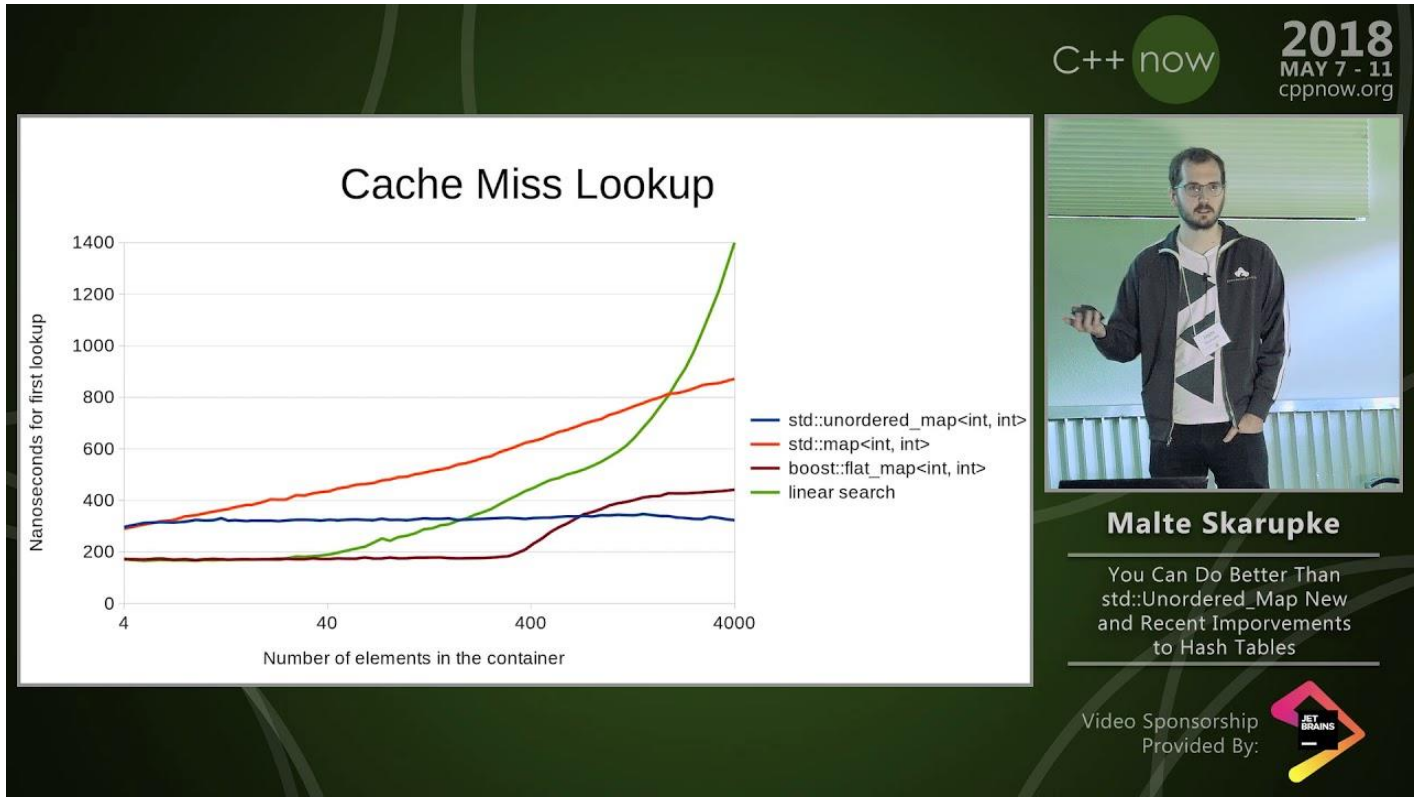
$k \cdot f(x) = \mathrm{O}\big(k \cdot g(x)\big) = \mathrm{O}\big(g(x)\big)$, where k is constant

Can hash map be faster?

Yes!

If we drop some requirements of standard library.

https://youtu.be/M2fKMP47slQ



Pop quiz!

Does it compile?

```
std::string(someString);
```

Pop quiz!

Does it compile?

```
std::string(someString);
```

Yes.

Does it compile?

```
std::string someString;
std::string(someString);
```

Pop quiz!

Does it compile?

```
std::string(someString);
```

Yes.

Does it compile?

```
std::string someString; // a variable definition
std::string(someString); // the same
```

Redefinition of `someString` variable.

Pop quiz!

```
std::unique_lock<std::mutex>(mutex);
```

It compiles. See the error?

Pop quiz!

```
std::unique_lock<std::mutex>(mutex);
```

It compiles. See the error?

https://youtu.be/lkgszkPnV8g

Thanks for coming!

Insights into the C++ standard library

Pavel Novikov

@cpp_ape

R&D Align Technology

align

Slides: https://git.io/Je44v

references

- `emplace_back` and `push_back`

https://stackoverflow.com/a/36919571/11068024: Why would I ever use push_back instead of emplace_back?

https://abseil.io/tips/112: Abseil Tip of the Week #112: emplace vs. push_back

- Small string optimization

https://youtu.be/kPR8h4-qZdk: Nicholas Ormrod “The strange details of std::string at Facebook”

- Sorting and selection

https://youtu.be/-0tO3Eni2uo: Fred Tingaud “A Little Order: Delving into the STL sorting algorithms”

https://youtu.be/zqs87a_7zxw: Malte Skarupke “Sorting in less than O(n log n):
Generalizing and optimizing radix sort”

- Associative containers

https://math.stackexchange.com/a/1560721/702319: showing that the partial sums
of $\log(j) = n\log(n) - n + O(\log(n))$

https://youtu.be/M2fKMP47slQ: Malte Skarupke “You Can Do Better than std::unordered_map:
New Improvements to Hash Table Performance”

references

- Beware of missing the variable name

https://youtu.be/lkgszkPnV8g: Louis Brandy “Curiously Recurring C++ Bugs at Facebook”