pinned_vector

A Contiguous Container without Pointer Invalidation

Meeting C++ 2018

`std::vector`

contiguous layout	cache locality
fastest iteration	O(1) lookup
random access	amortized O(1) growth

`std::vector`

contiguous layout

cache locality

POINTER INVALIDATION

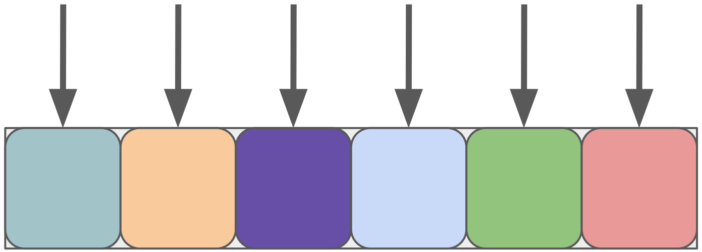
fastest iteration

O(1) lookup

random access

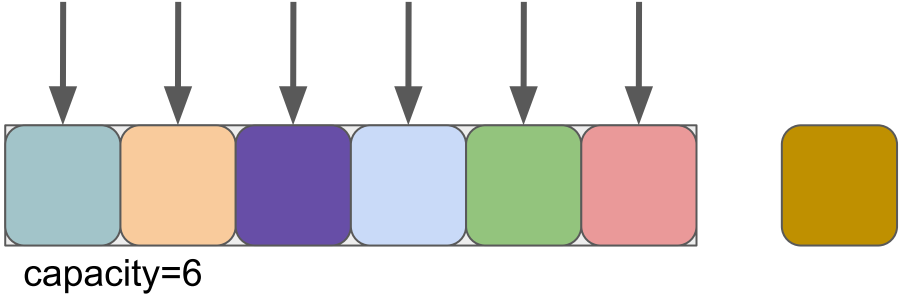
amortized O(1) growth

`std::vector` Invalidation

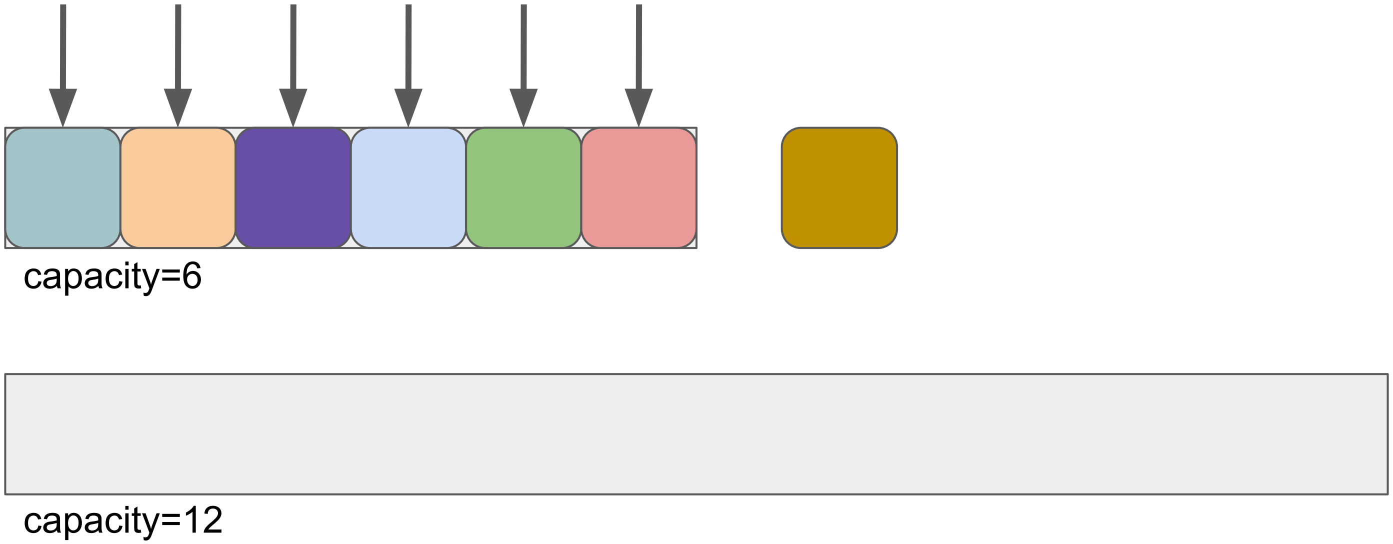


capacity=6

`std::vector` Invalidation



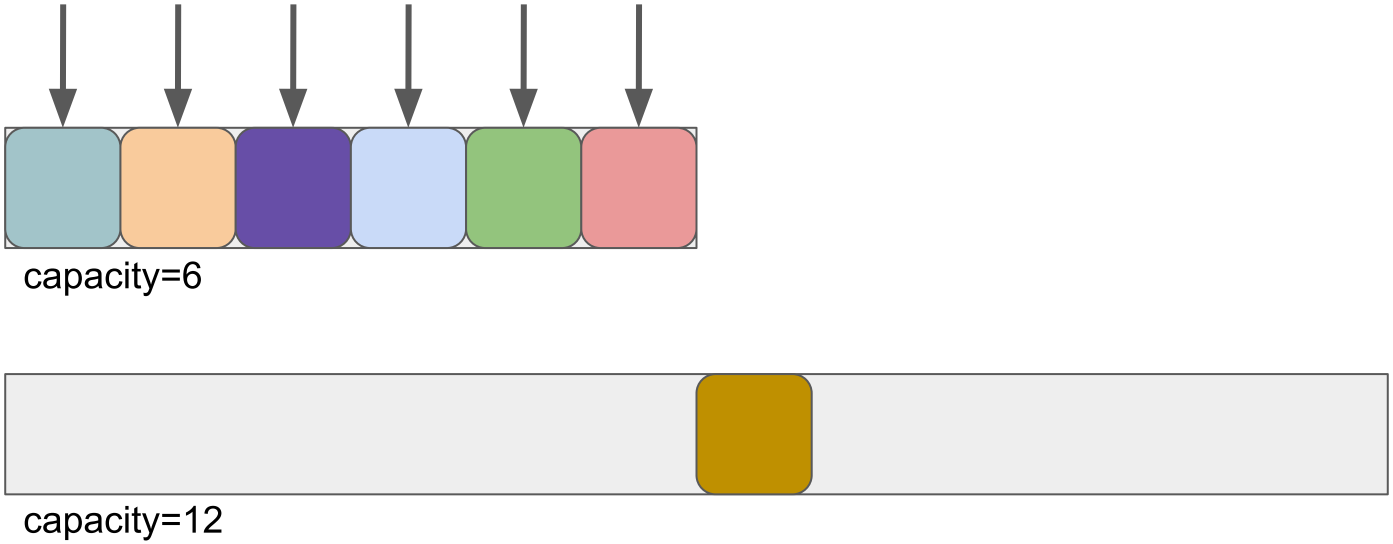
`std::vector` Invalidation



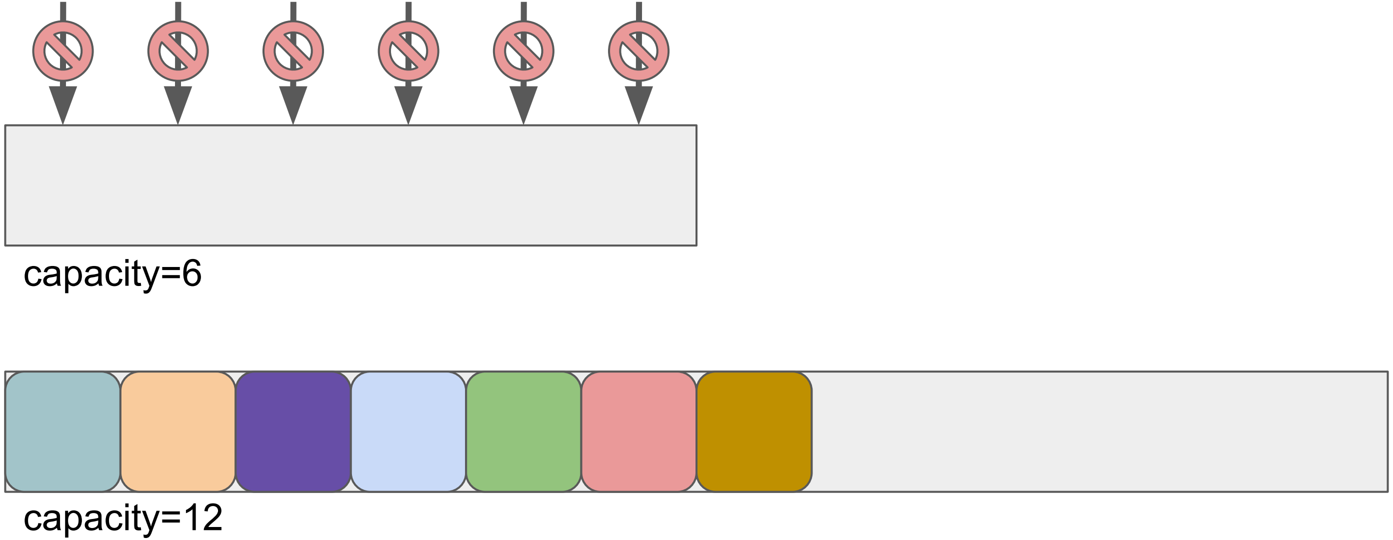
capacity=6

capacity=12

`std::vector` Invalidation



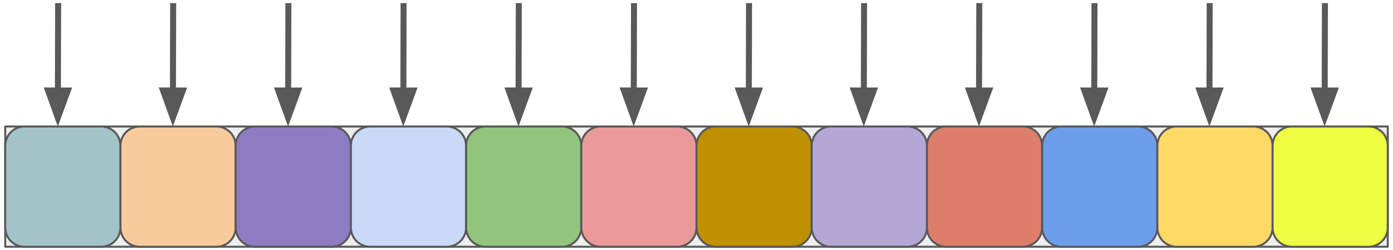
`std::vector` Invalidation



`std::vector` Invalidation

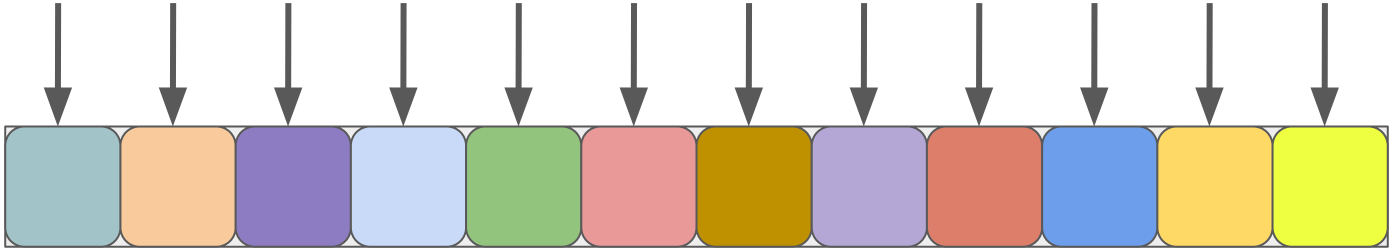
may invalidate all	always invalidates all	may invalidate other
push_back	clear	insert
emplace_back	assign	erase
insert		
emplace		
reserve		
resize		
shrink_to_fit		

Contiguous Storage Invariant



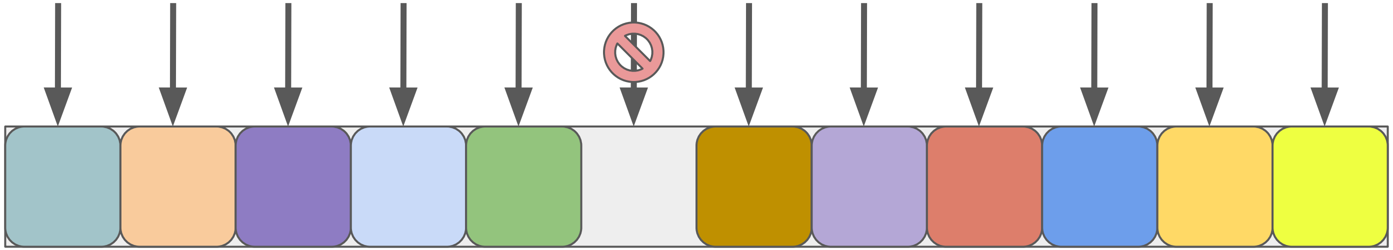
Contiguous Storage Invariant

erase(🟥)



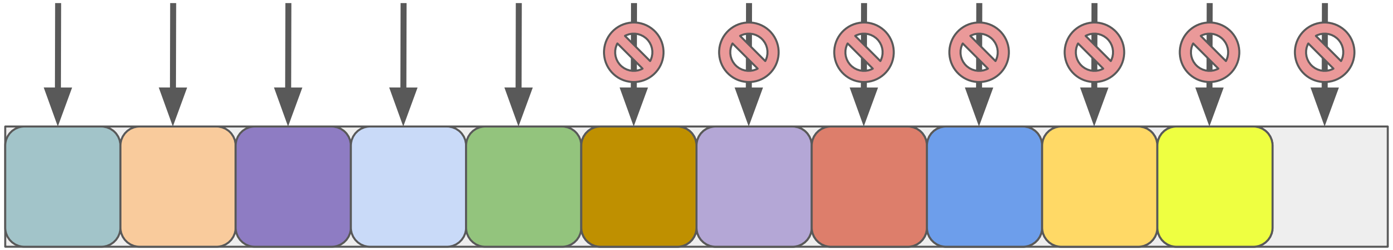
Contiguous Storage Invariant

erase(🟥)

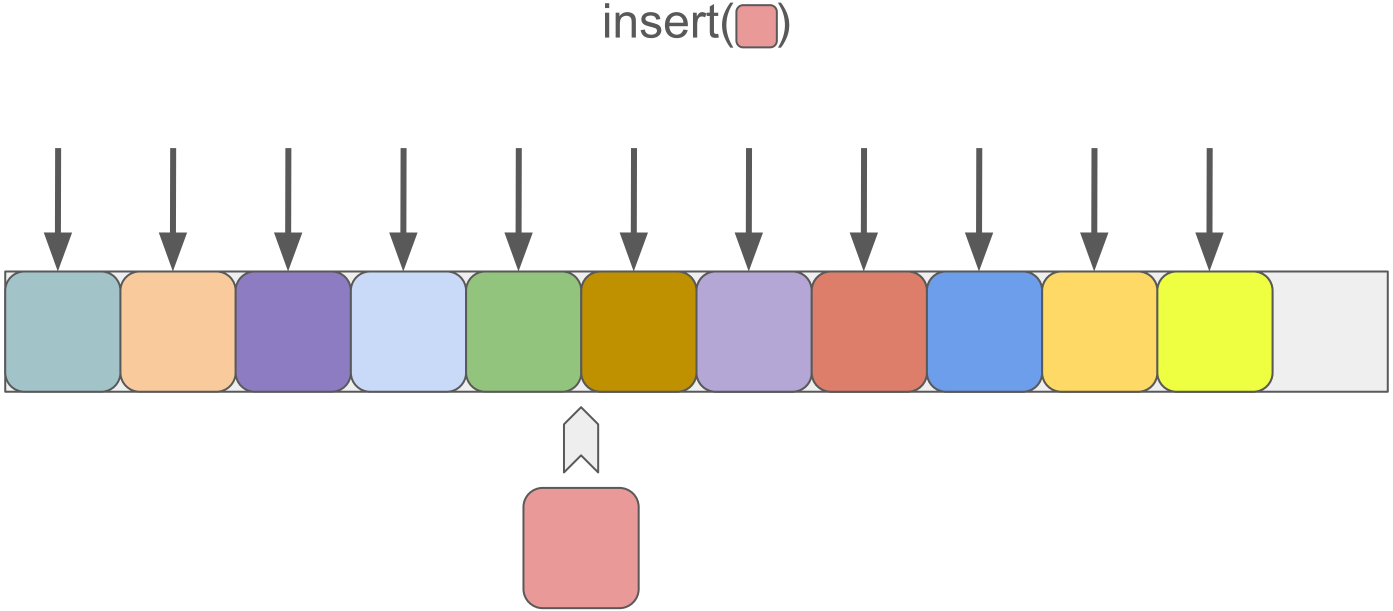


Contiguous Storage Invariant

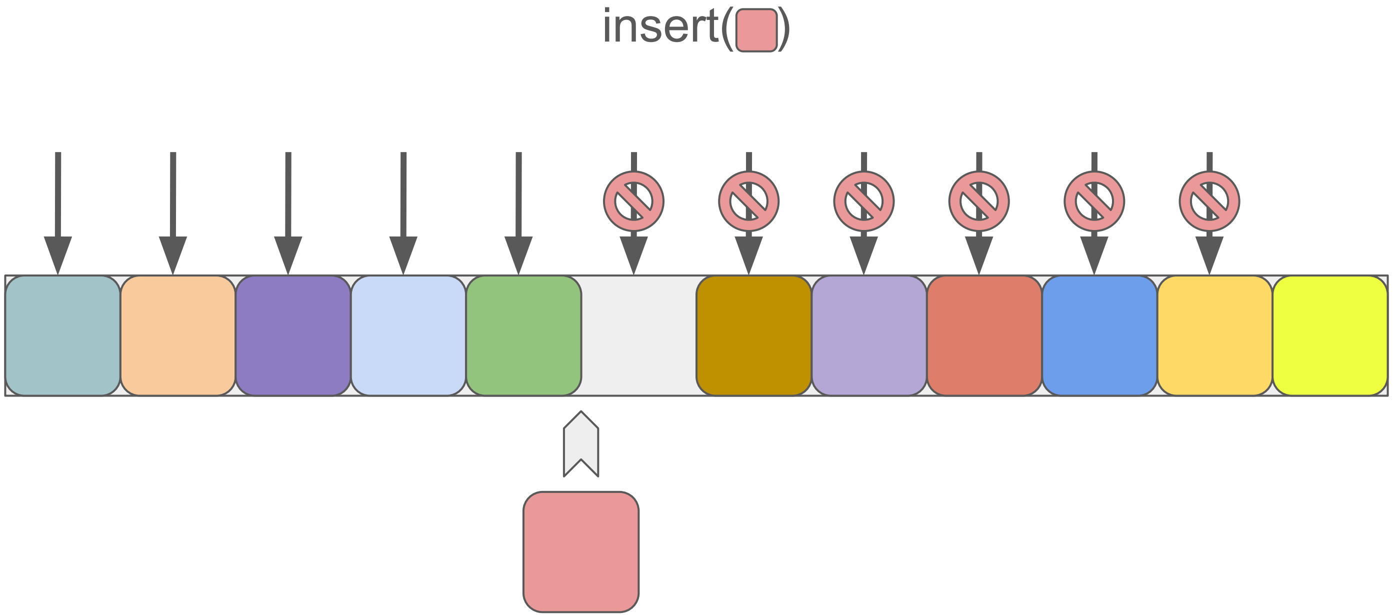
erase(🟥)



Contiguous Storage Invariant

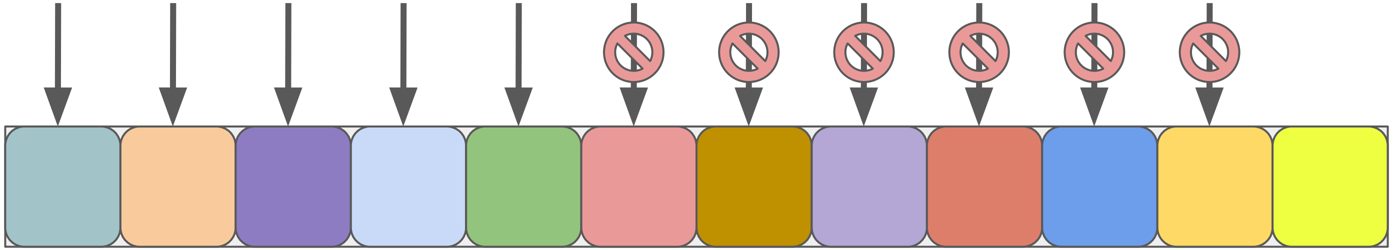


Contiguous Storage Invariant

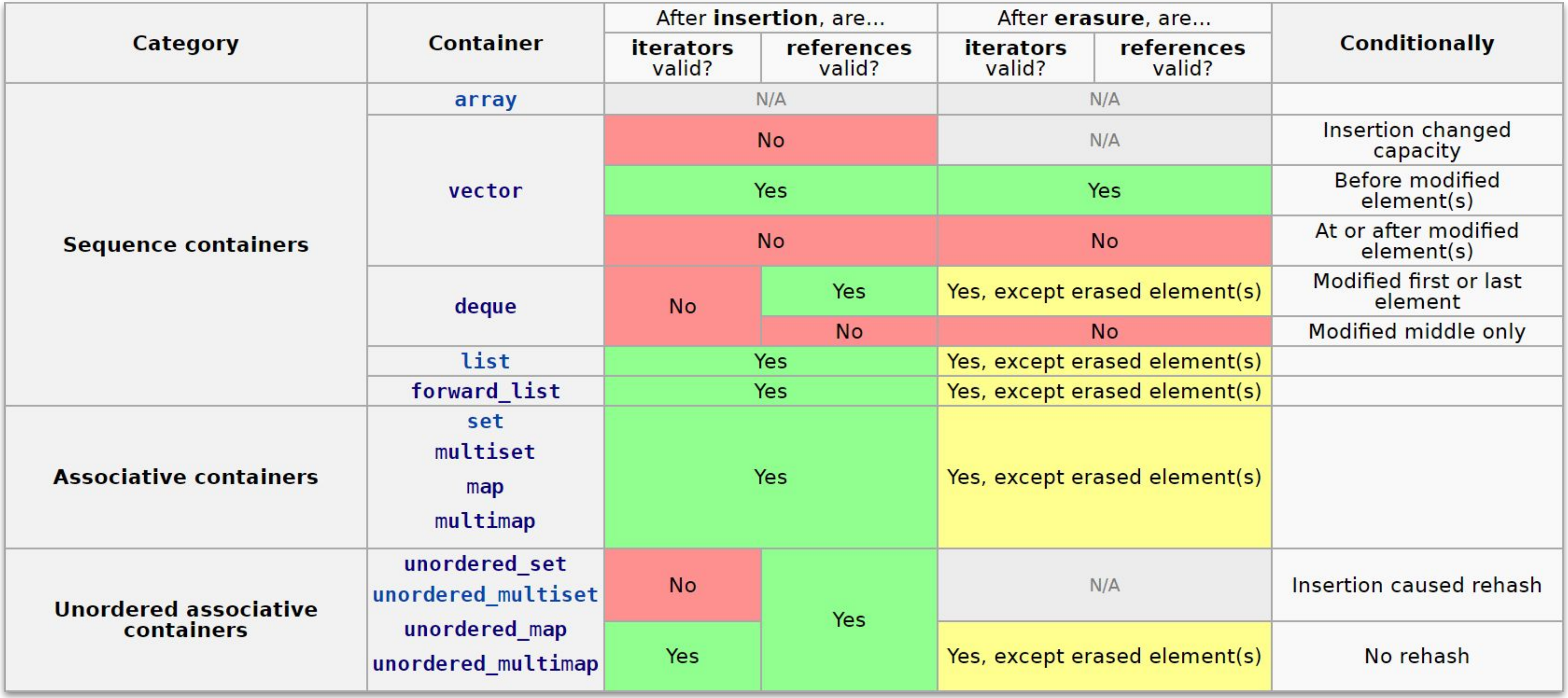


Contiguous Storage Invariant

insert(🟥)



Alternatives with Truly Stable Pointers



Category	**Container**	After **insertion**, are...		After **erasure**, are...		**Conditionally**
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		

Associative containers	`set`
	`multiset`
	`map`
	`multimap`
		Yes		Yes, except erased element(s)		

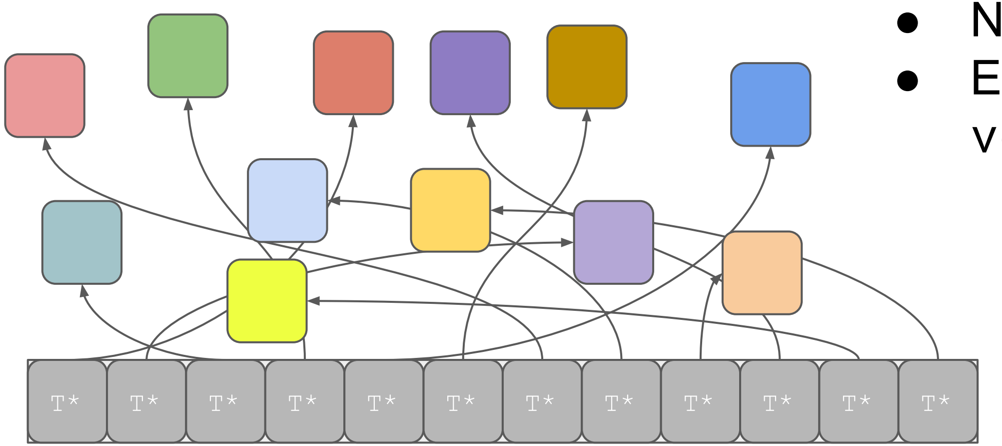
Unordered associative containers	`unordered_set`
	`unordered_multiset`
	`unordered_map`
	`unordered_multimap`
		No	Yes	N/A		Insertion caused rehash

		Yes		Yes, except erased element(s)		No rehash

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

Alternatives with Truly Stable Pointers

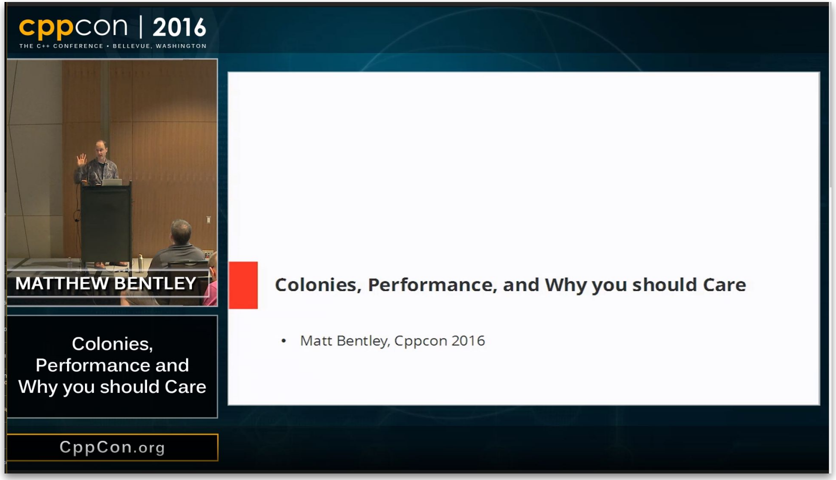
`boost::stable_vector<T>`



- Not a “vector”
- Not contiguous
- Equivalent to `vector<unique_ptr<T>>`

Alternatives with Truly Stable Pointers

`plf::colony`



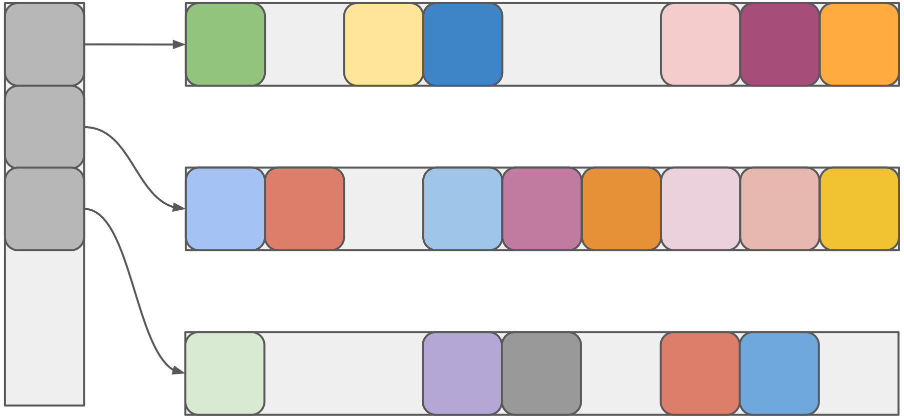
Colonies, Performance and Why you should Care

Colonies, Performance, and Why you should Care

- Matt Bentley, Cppcon 2016

Alternatives with Truly Stable Pointers

`plf::colony`



- Manages elements in disjoint memory chunks
- Contiguous layout not guaranteed
- Iteration performance comparable to std::deque
- Primary use case is storage, not iteration

“A Contiguous Container without Pointer Invalidation”

~~“A Contiguous Container without Pointer Invalidation”~~

not quite…

must maintain contiguous layout invariant

“A Contiguous Container with **Essential** Pointer Invalidation”

The minimum amount of pointer invalidation absolutely necessary to maintain the contiguous layout invariant.

*If insertion or erasure occurs **only** at the **end** of the container then pointers to all other elements shall remain valid.*

Idealized std::vector *with **infinite** capacity.*

`std::vector` Invalidation

may invalidate all	always invalidates all	may invalidate other
push_back	clear	insert
emplace_back	assign	erase
insert		
emplace		
reserve		
resize		
shrink_to_fit		

`pinned_vector` Invalidation

may invalidate all	always invalidates all	may invalidate other
~~push_back~~	clear	insert
~~emplace_back~~	assign	erase
~~insert~~		
~~emplace~~		
~~reserve~~		
~~resize~~		
~~shrink_to_fit~~		

Virtual Memory History

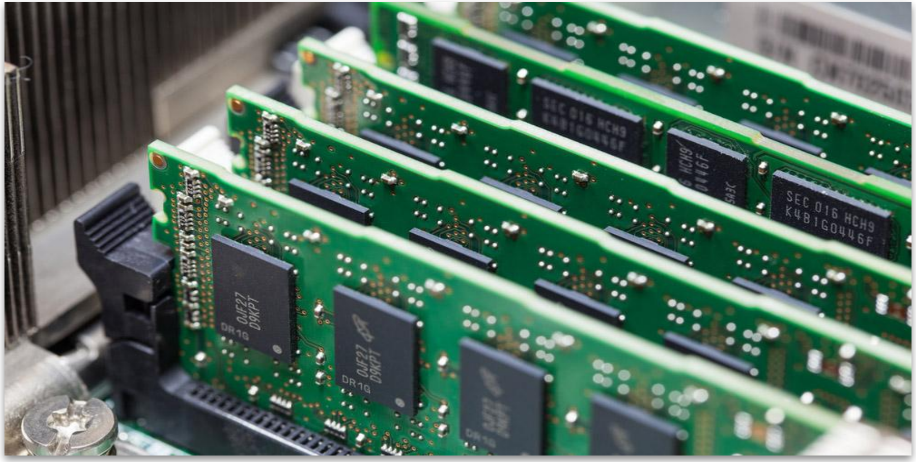
- Introduced in DEC’s VAX-11/780 (*“Virtual Address eXtension”, 1977*)
- First consumer CPU with integrated MMU Intel 80286 (1982)

Virtual Memory

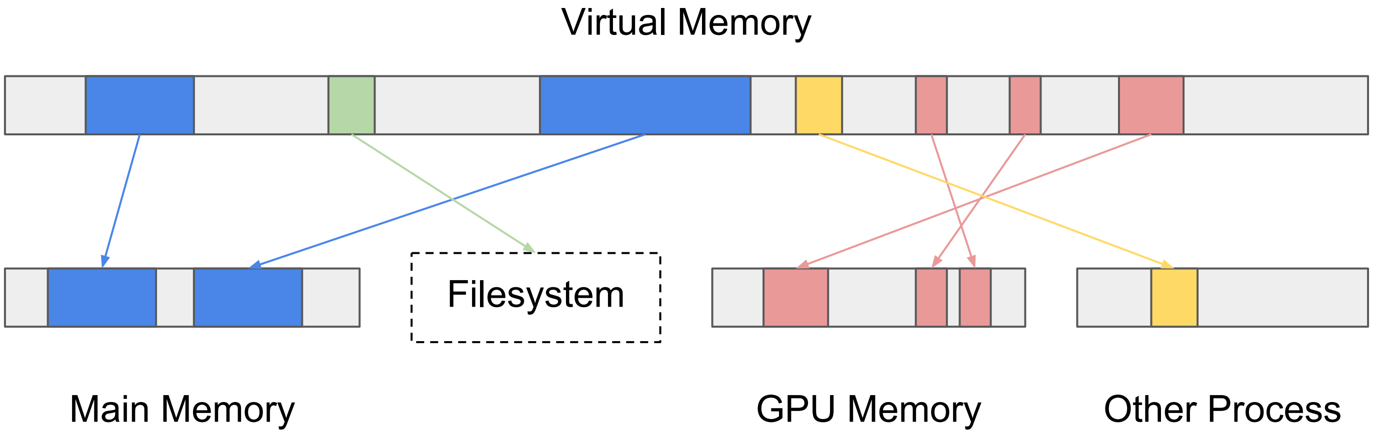
- Illusion of huge memory
- Abstraction of Hardware Storage and Resources
  - Physical Memory
  - Filesystem
  - Memory mapped I/O
  - Inter-Process Communication

Virtual Memory vs Physical Memory

```
#include <memory>
#include <iostream>

int main()
{
  auto foo = std::make_unique(42)
  std::cout << foo.get() << std::endl;
  return 0;
}
```



Virtual Memory



Virtual Memory

- Process isolation
  - Separate address space
- More space then physical available
  - x86-64 eg. 128TiB

Page

- Fixed size block of virtual memory
- Most CPUs have a minimum page size of 4 KiB
    - Memory aligned in page size
- Huge Pages
    - x86-64 has also 2 MiB and 1 GiB pages
    - Performance

Memory Management Unit

- Everyone here has seen it in action already
  - `terminated by signal SIGSEGV (Address boundary error)`
  - `Access Violation`

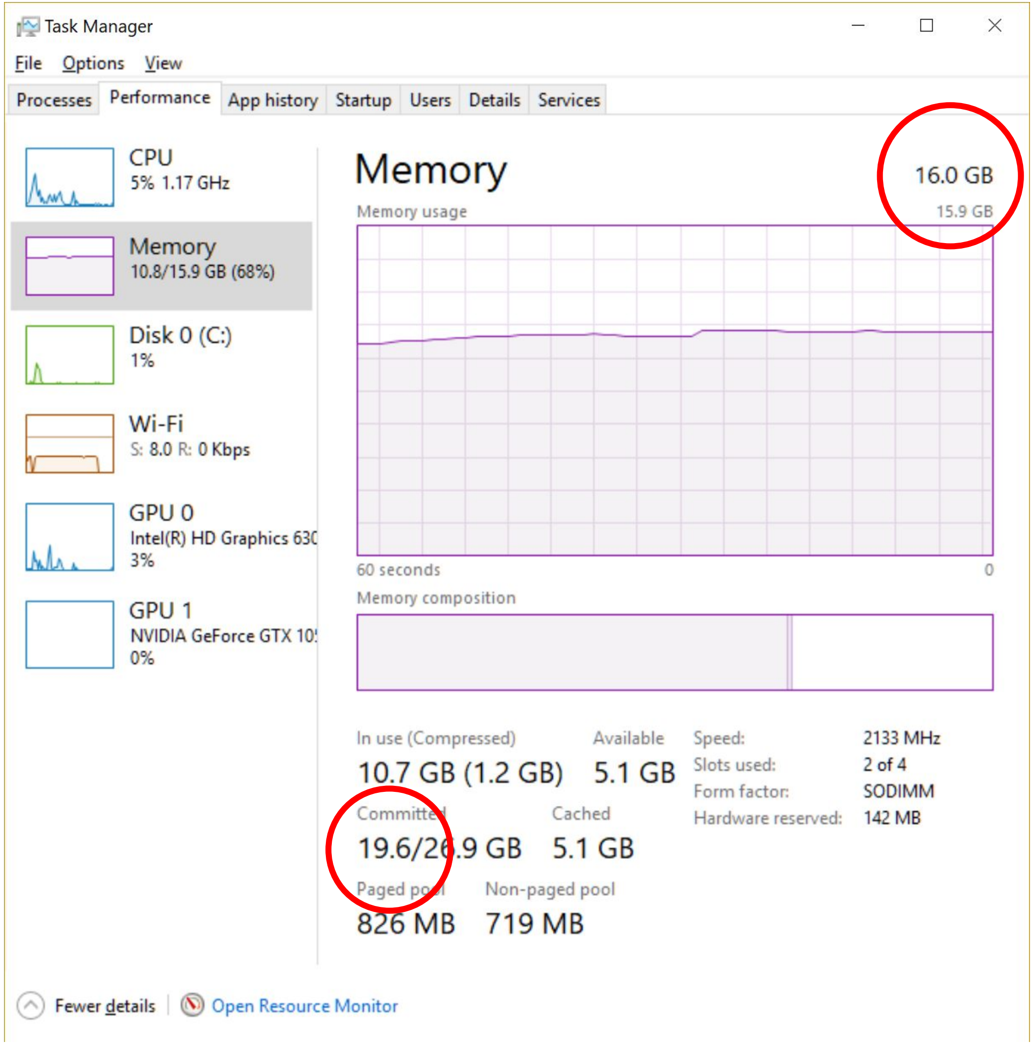
- Separate part on the CPU to map virtual memory addresses to physical memory addresses
- Page protection
  - Check Read, Write, Executable Bit

Translation Lookaside Buffer

- Part of the MMU
- Stores mapping of physical and virtual addresses
- Hardware accelerated
- Typically has 4096 entries

Page Table

- Cache for TLB
- Stored in memory
- Page walk
  - Hardware or Software



CPU
5% 1.17 GHz

Memory
10.8/15.9 GB (68%)

Disk 0 (C:)
1%

Wi-Fi
S: 8.0 R: 0 Kbps

GPU 0
Intel(R) HD Graphics 630
3%

GPU 1
NVIDIA GeForce GTX 10
0%

Memory
16.0 GB

Memory usage
15.9 GB

60 seconds
0

Memory composition

In use (Compressed)
10.7 GB (1.2 GB)

Available
5.1 GB

Committed
19.6/26.9 GB

Cached
5.1 GB

Paged pool
826 MB

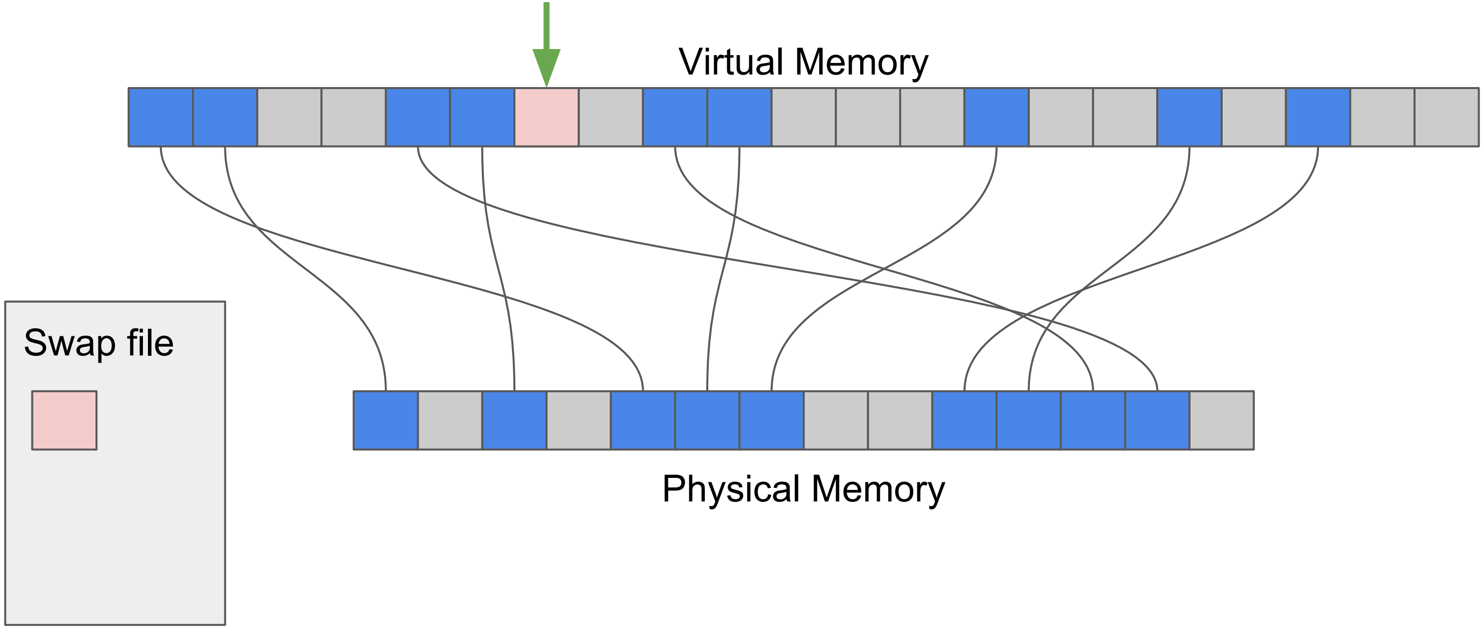
Non-paged pool
719 MB

Speed: 2133 MHz
Slots used: 2 of 4
Form factor: SODIMM
Hardware reserved: 142 MB

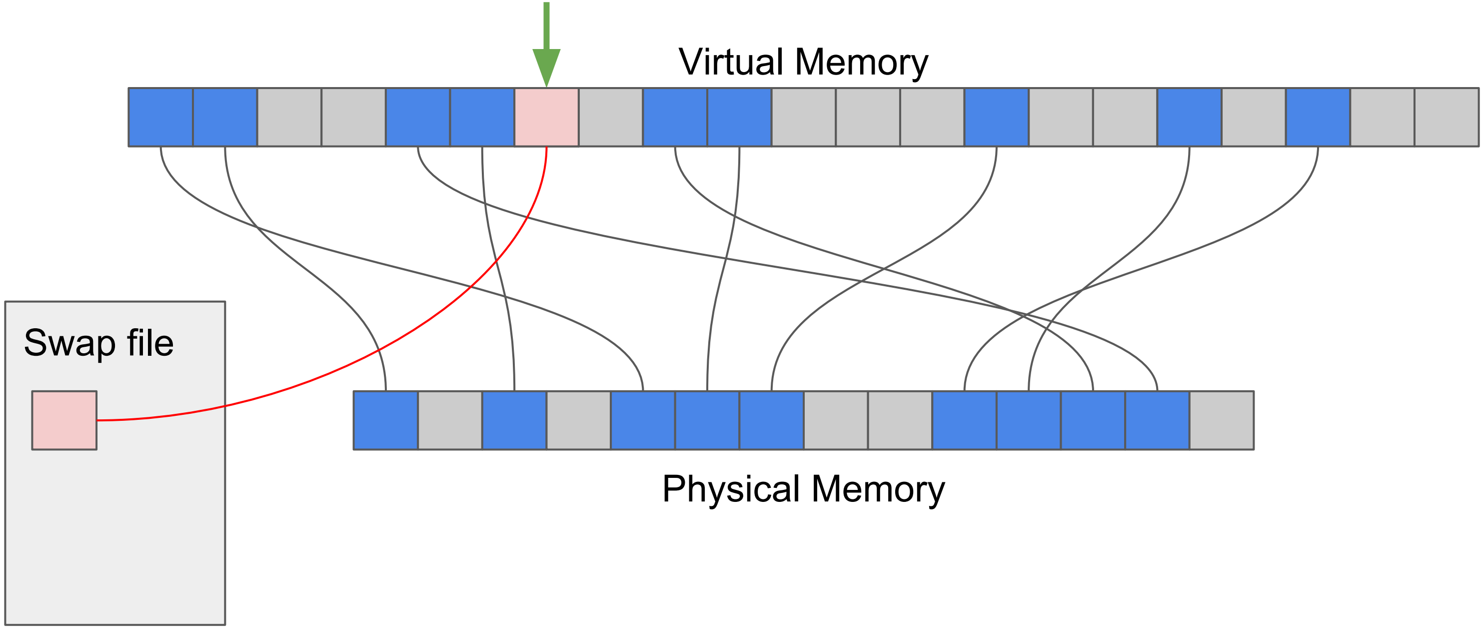
Swap Space

- File / Partition
- Unused Pages are saved on disk to free physical memory
- Controlled by the OS

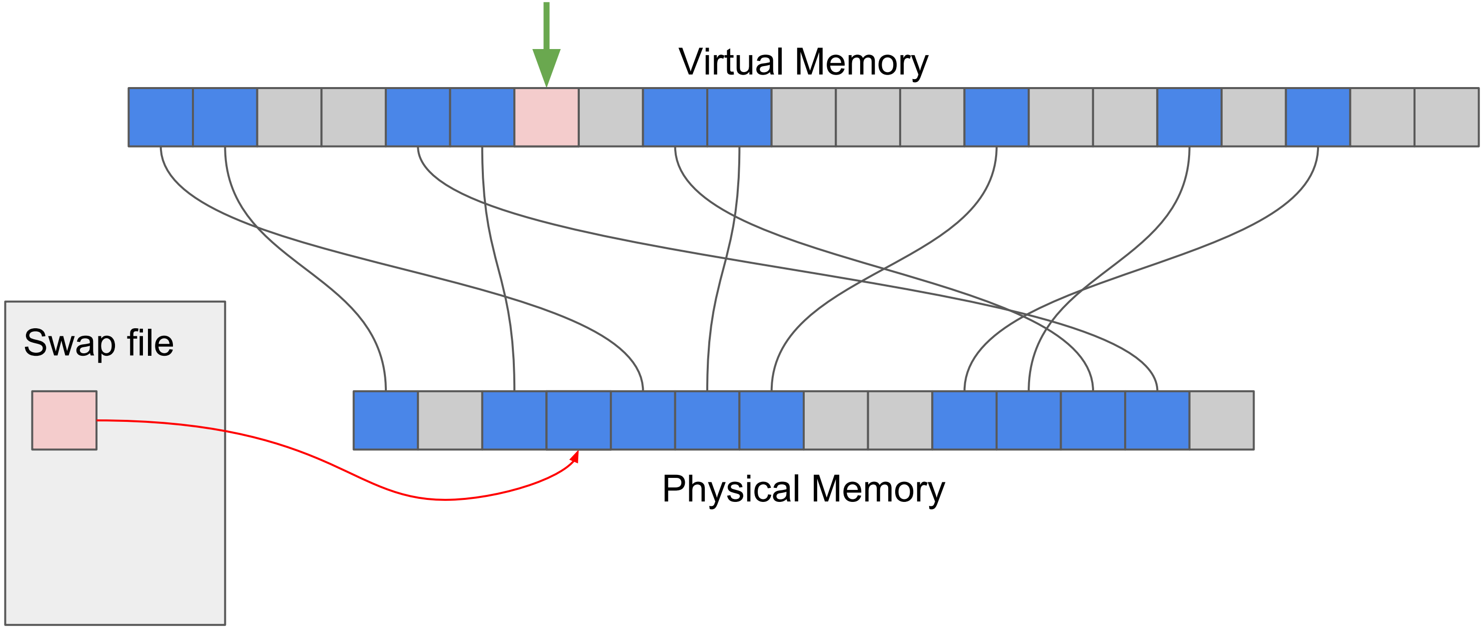
Page-Faults



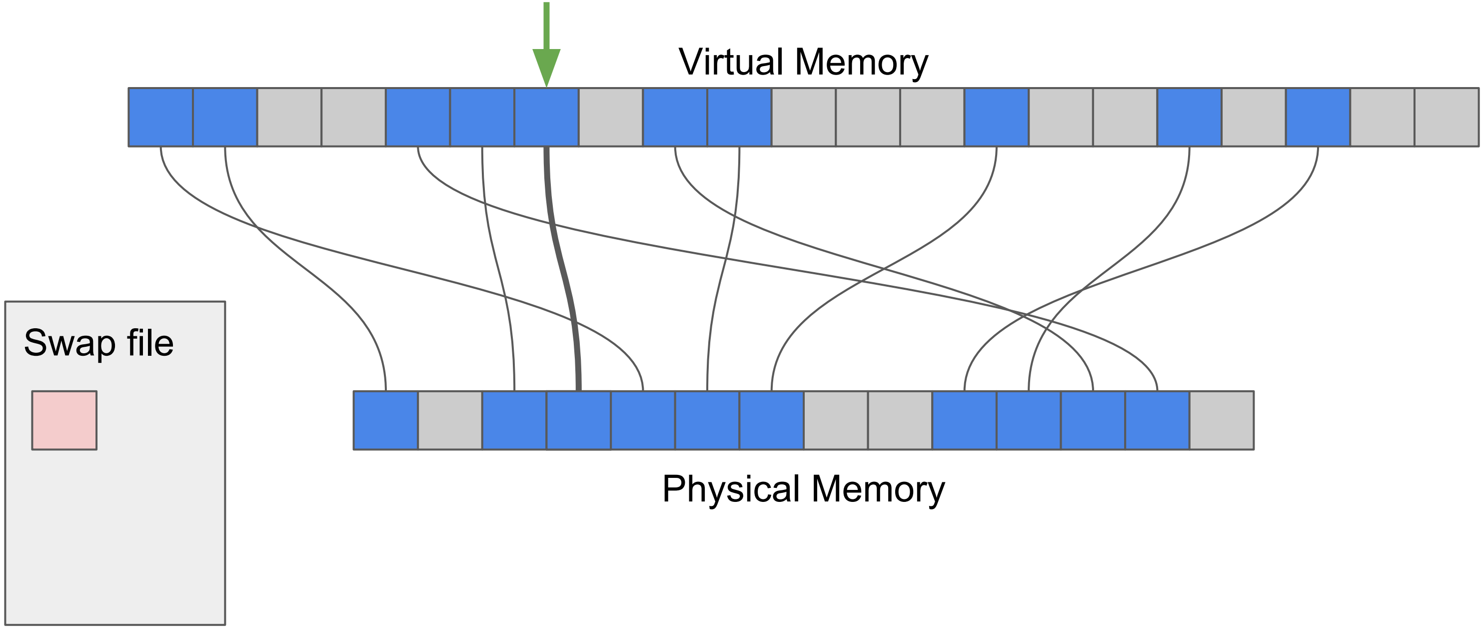
Page-Faults



Page-Faults



Page-Faults

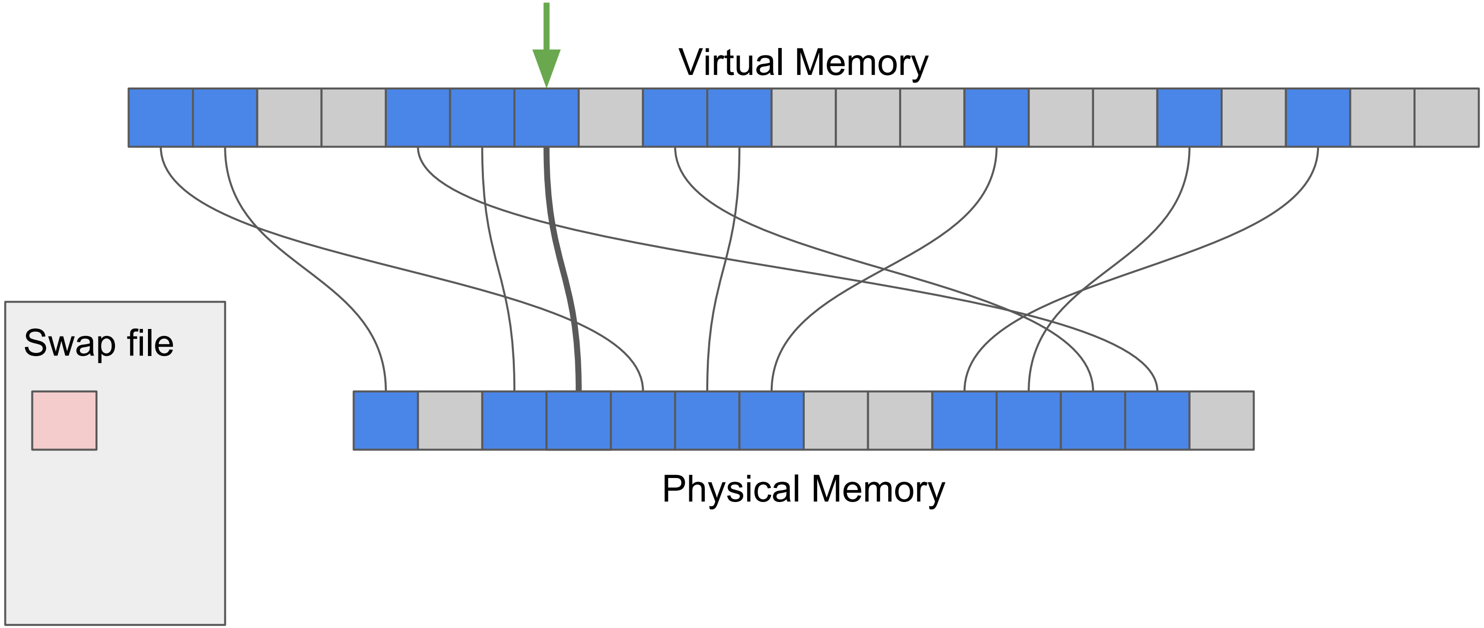


Virtual Memory

Swap file

Physical Memory

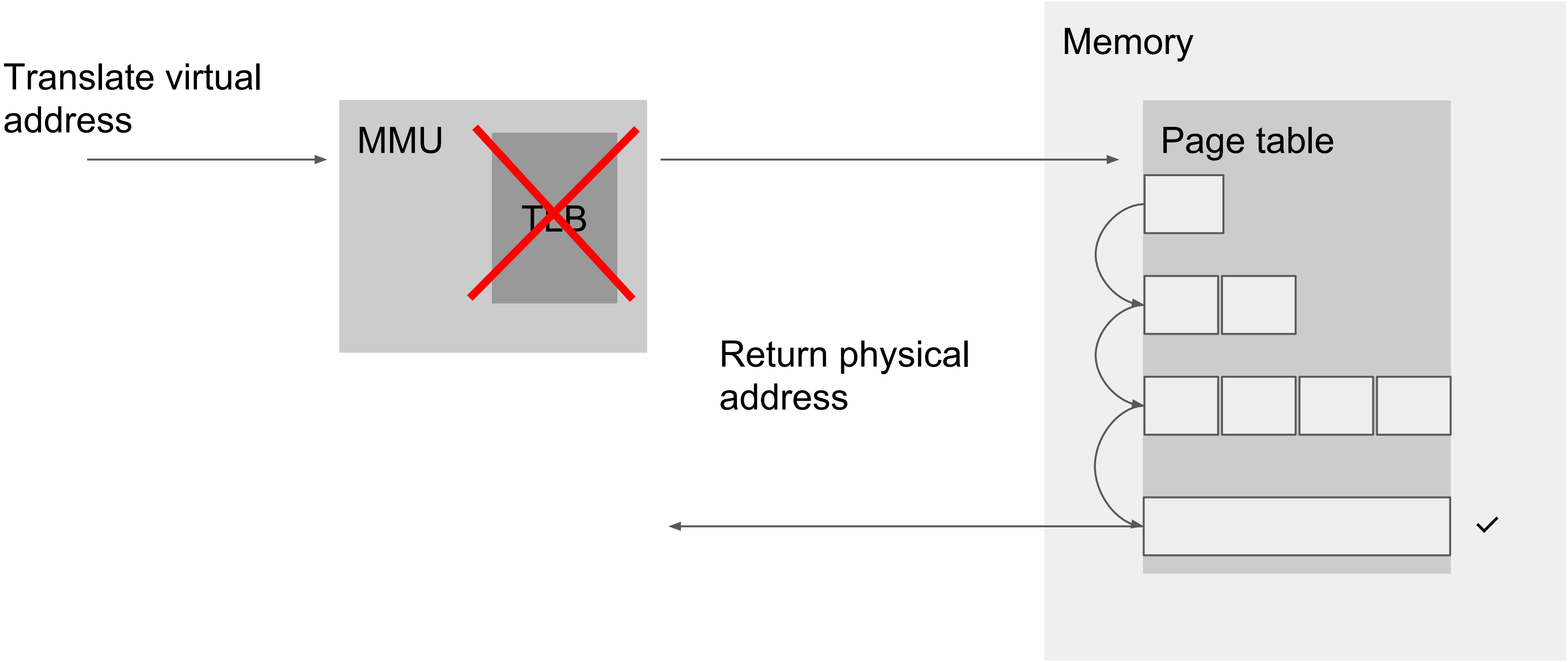
Page-Faults



Page-Faults

- Access to pages which are not loaded in physical memory
- Swap of pages into/from swap file
- Super expensive

TLB Miss



Translate virtual address

MMU

TLB

Return physical address

Memory

Page table

✓

Thrashing

- Constant swapping of pages
- Unresponsive system
  - Filesystem Access

Mapping Memory

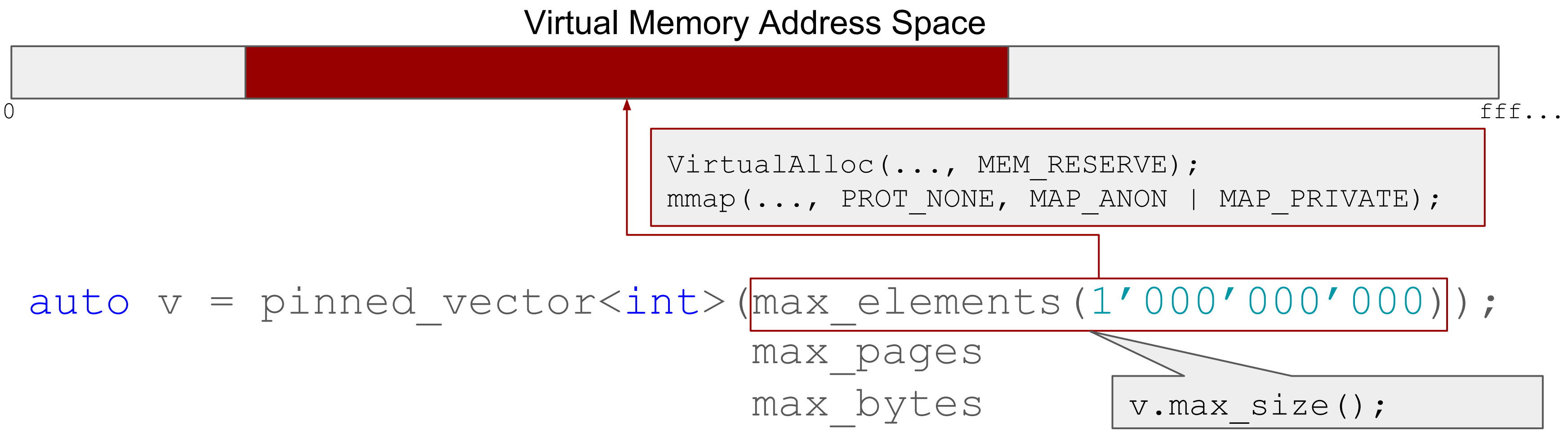
Reserve

- Prevents other allocations within reserved area
- Does not consume memory or swap space

Commit

- Get physical memory space
- Consumes memory or swap space

pinned_vector Internals



Virtual Memory Address Space

0

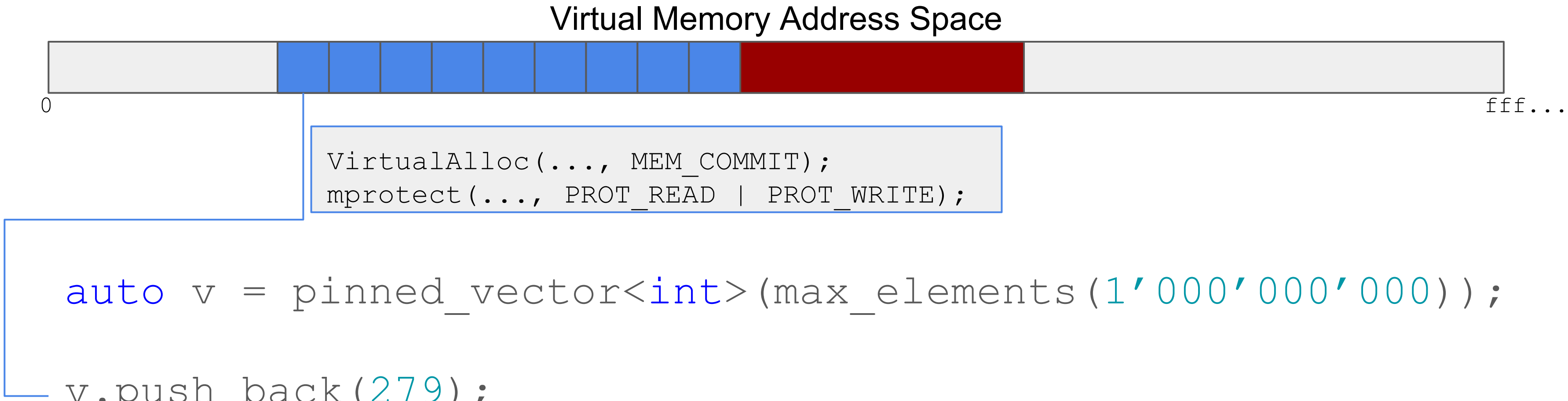
fff...

```
VirtualAlloc(..., MEM_RESERVE);
mmap(..., PROT_NONE, MAP_ANON | MAP_PRIVATE);
```

```
auto v = pinned_vector<int>(max_elements(1’000’000’000));
                            max_pages
                            max_bytes
```

```
v.max_size();
```

pinned_vector Internals



```
VirtualAlloc(..., MEM_COMMIT);
mprotect(..., PROT_READ | PROT_WRITE);
```

```
auto v = pinned_vector<int>(max_elements(1’000’000’000));

v.push_back(279);

v.push_back(188);

...
```

pinned_vector Internals

Virtual Memory Address Space

```
VirtualFree(..., MEM_DECOMMIT);
mprotect(..., PROT_NONE); madvise(..., MADV_DONTNEED);
```

```
auto v = pinned_vector<int>(max_elements(1'000'000'000));

v.pop_back();

...

v.shrink_to_fit();
```

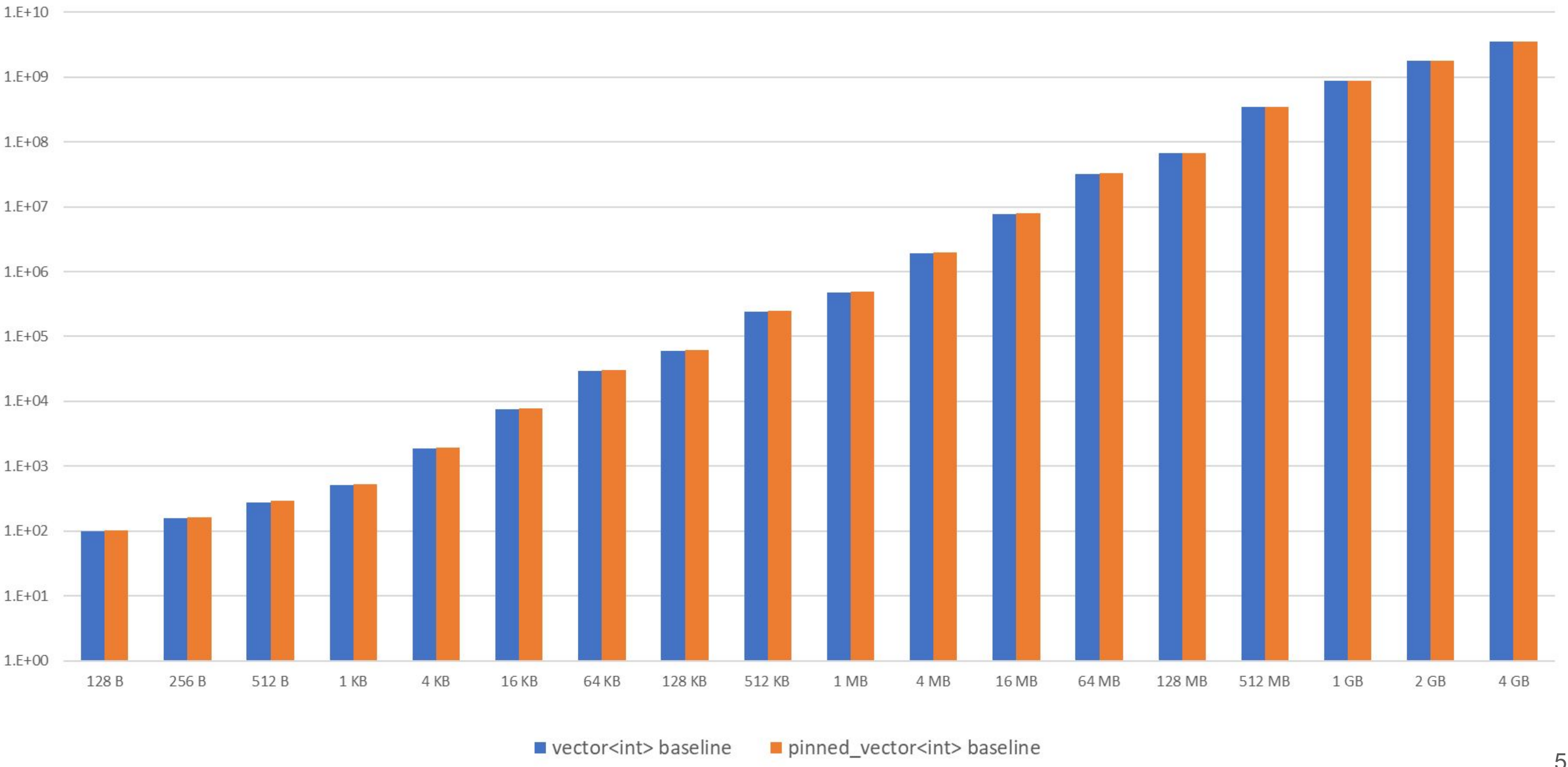
But Is It Any Good?

`std::vector` ⚔ `pinned_vector`

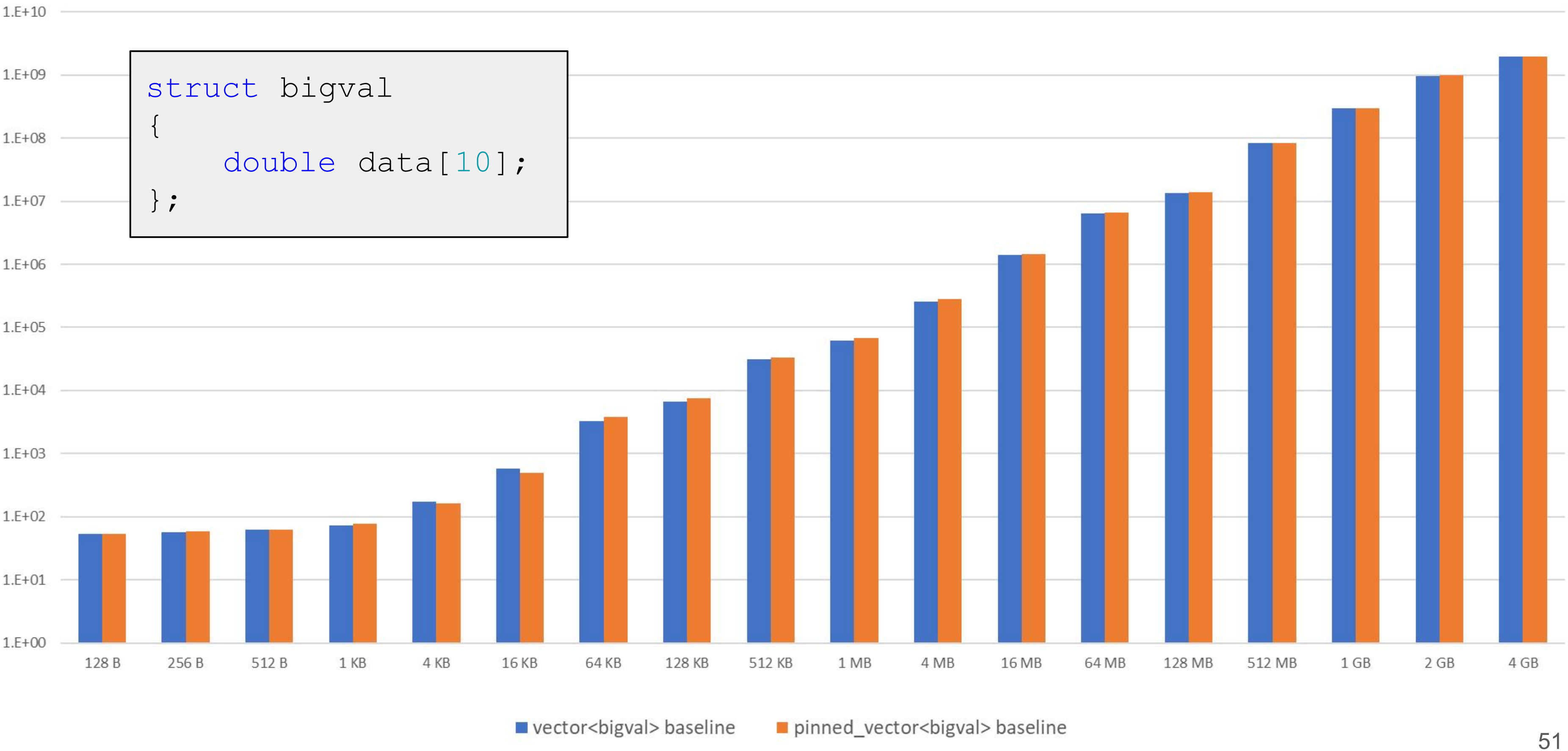
Round 1: establish a common baseline

```
auto v = Container<T>();

v.reserve(n);

⏱

fill_n(back_inserter(v), n, x);

⏱
```

Baseline for `int`

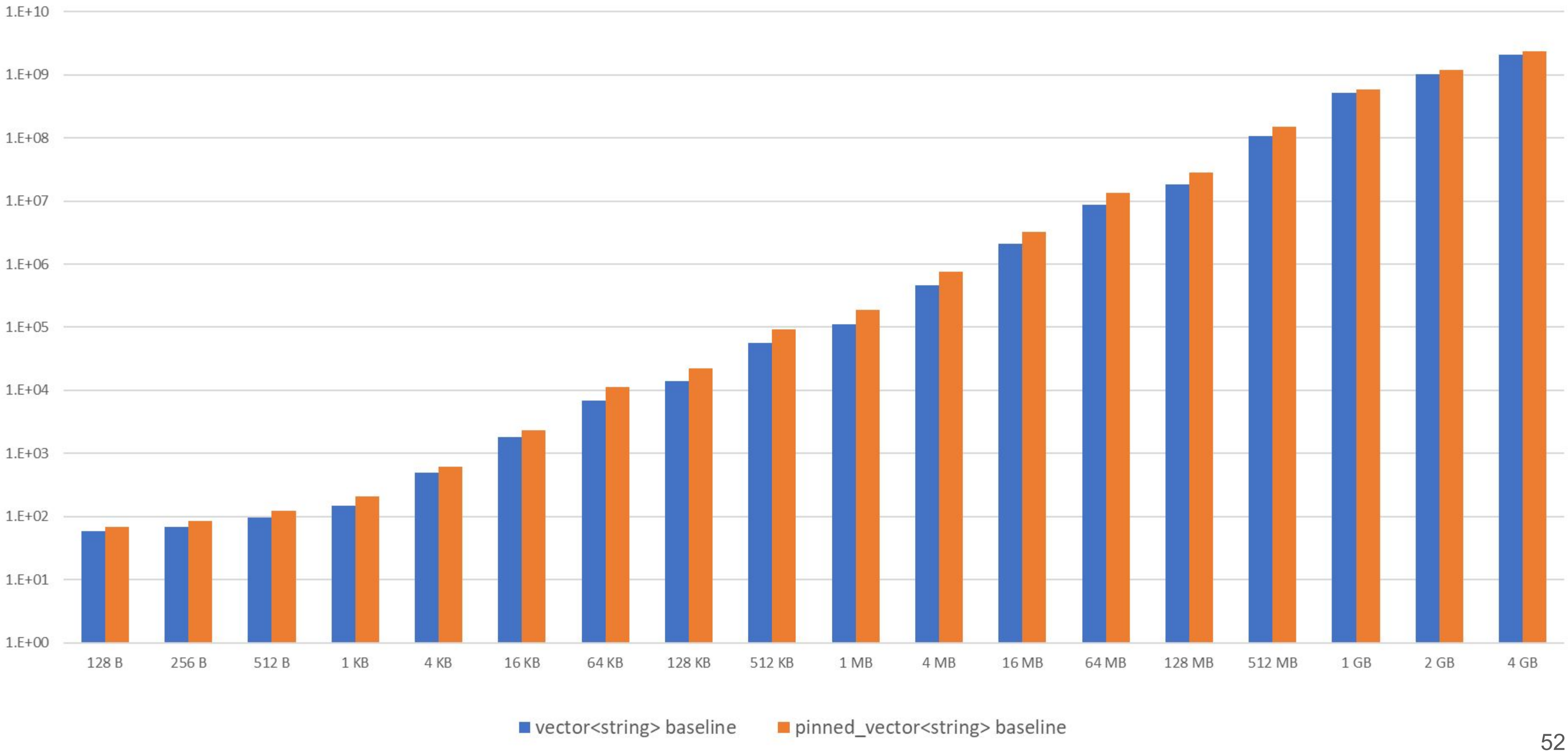


Baseline for `bigval`

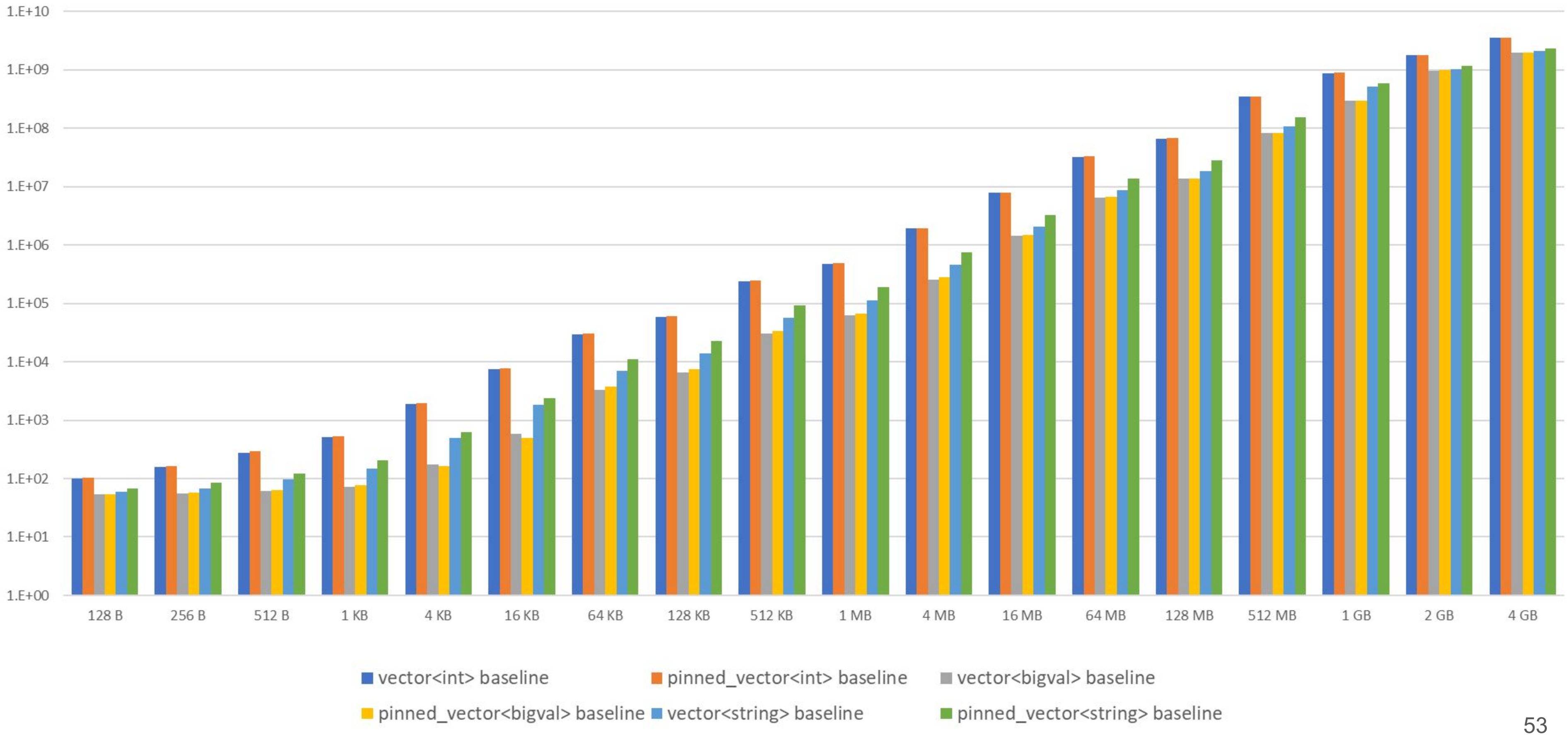


```
struct bigval
{
    double data[10];
};
```

Baseline for `std::string`



Baseline All



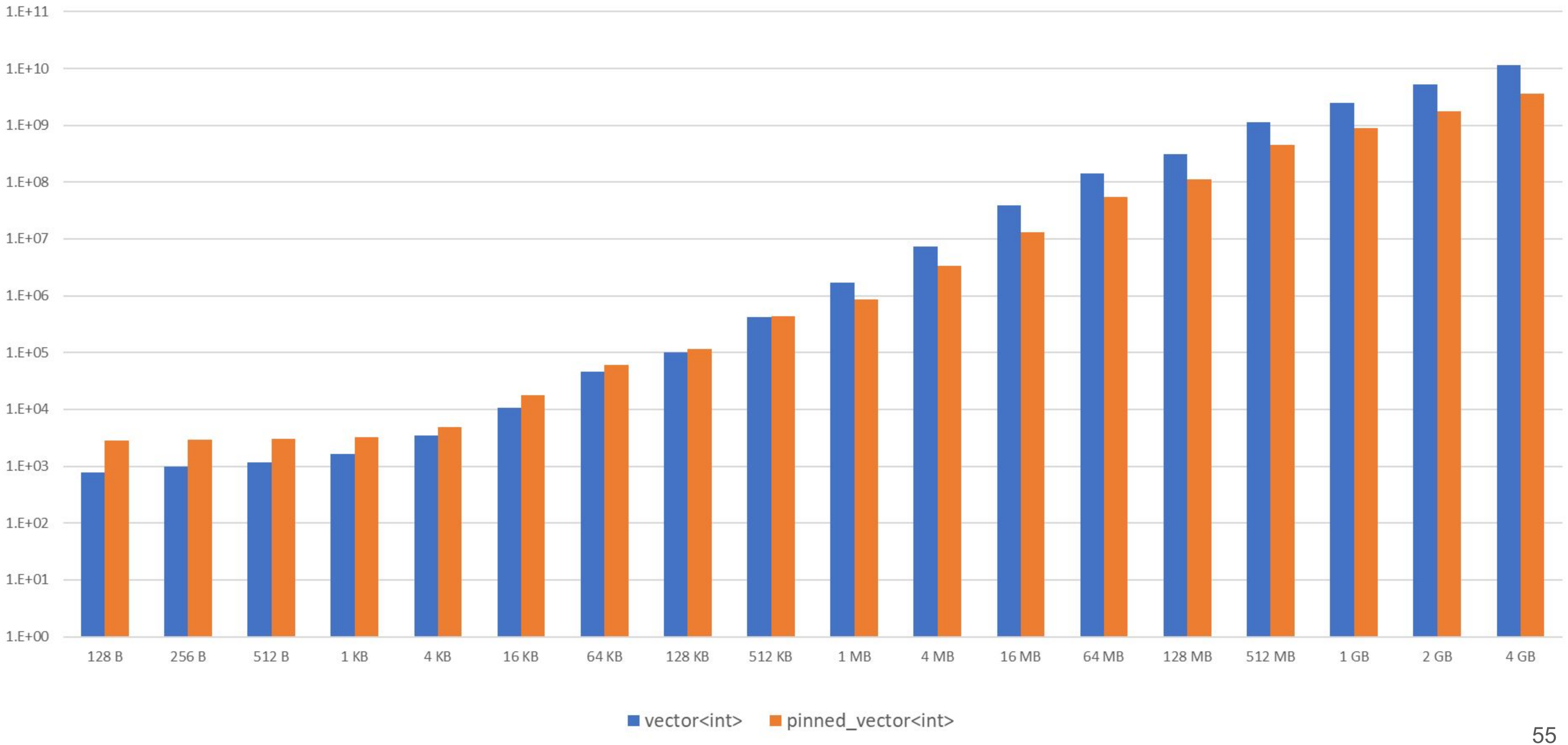
So Is It Any Good?

`std::vector` ⚔️ `pinned_vector`

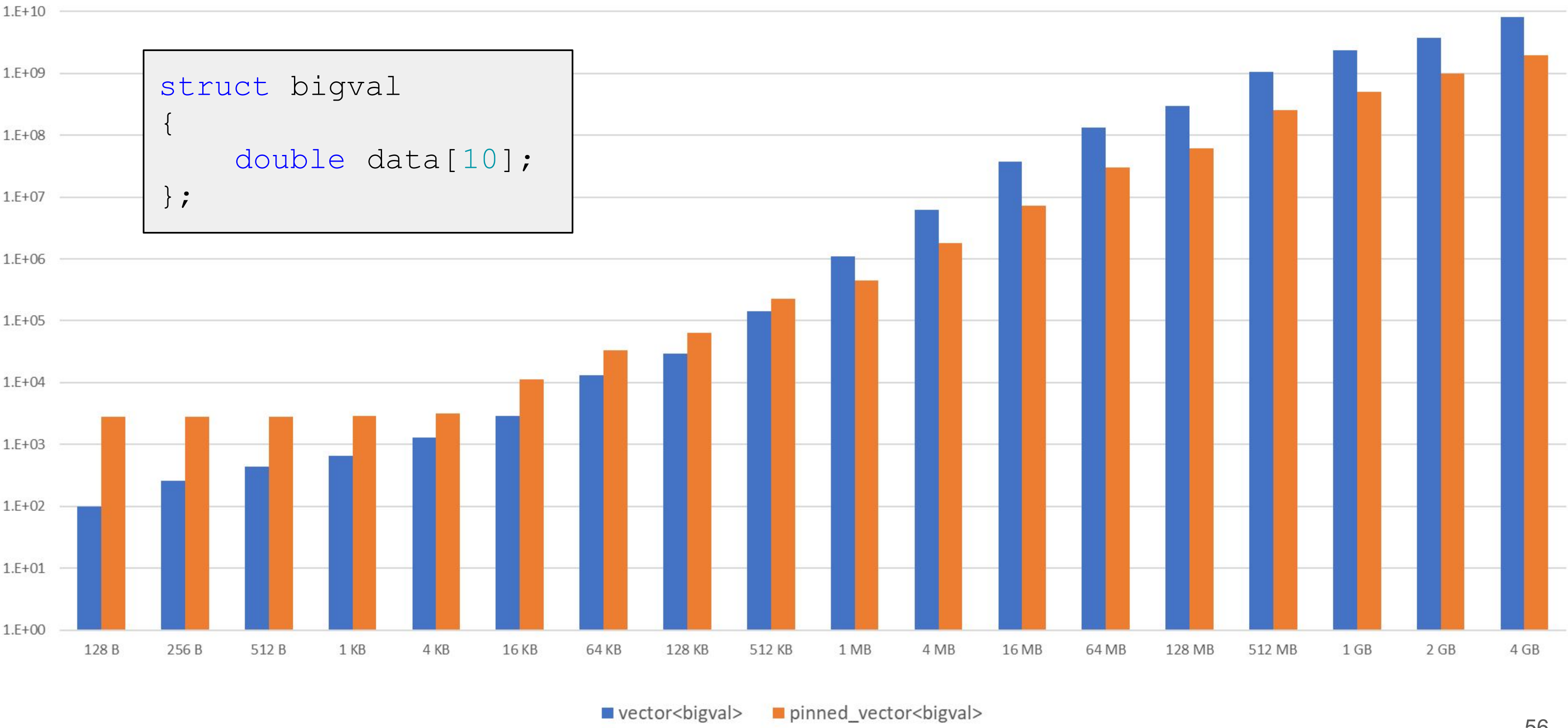
Round 2: size not known upfront

```
auto v = Container<T>();

~~v.reserve(n);~~

⏱

fill_n(back_inserter(v), n, x);

⏱
```

Total Time for `int`

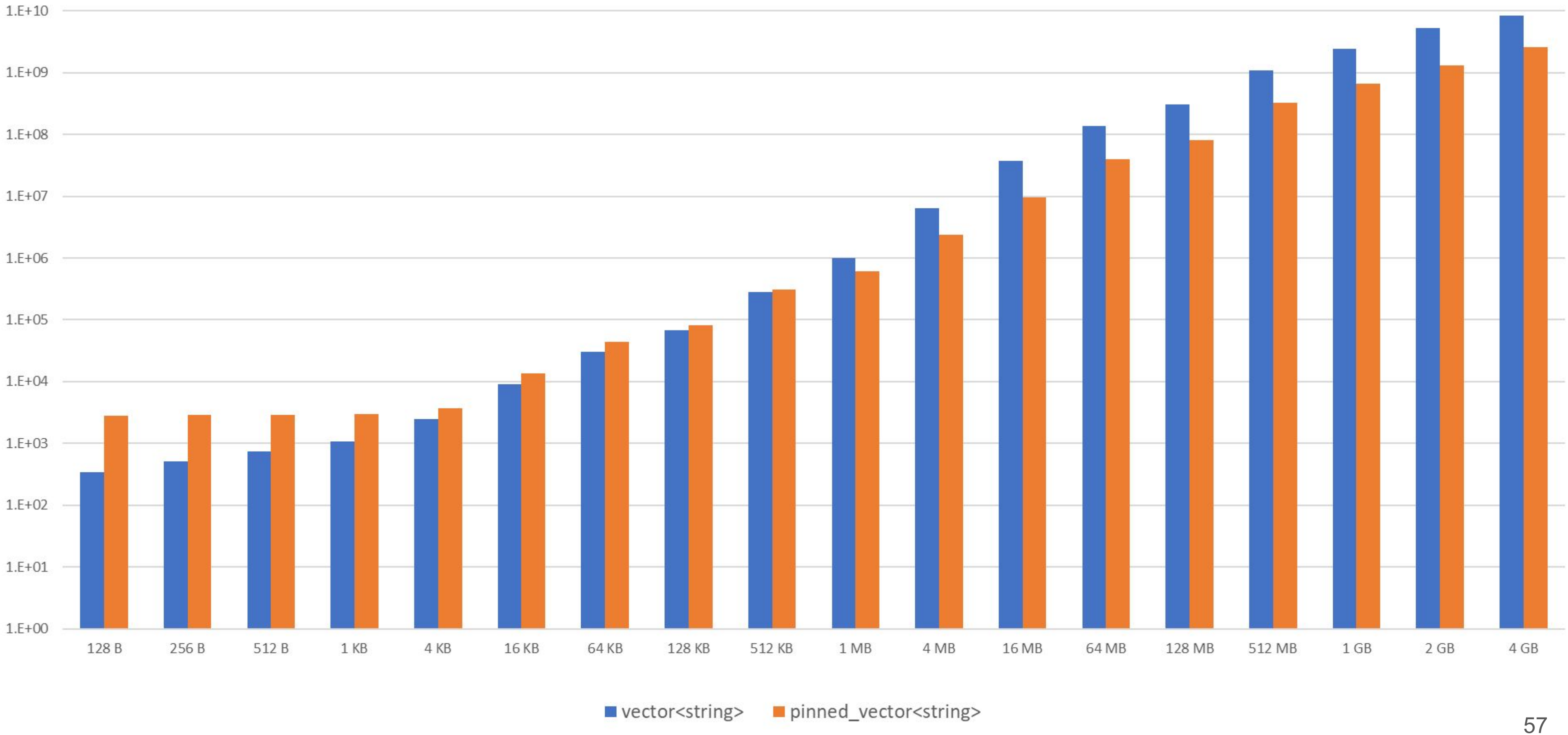


Total Time for `bigval`

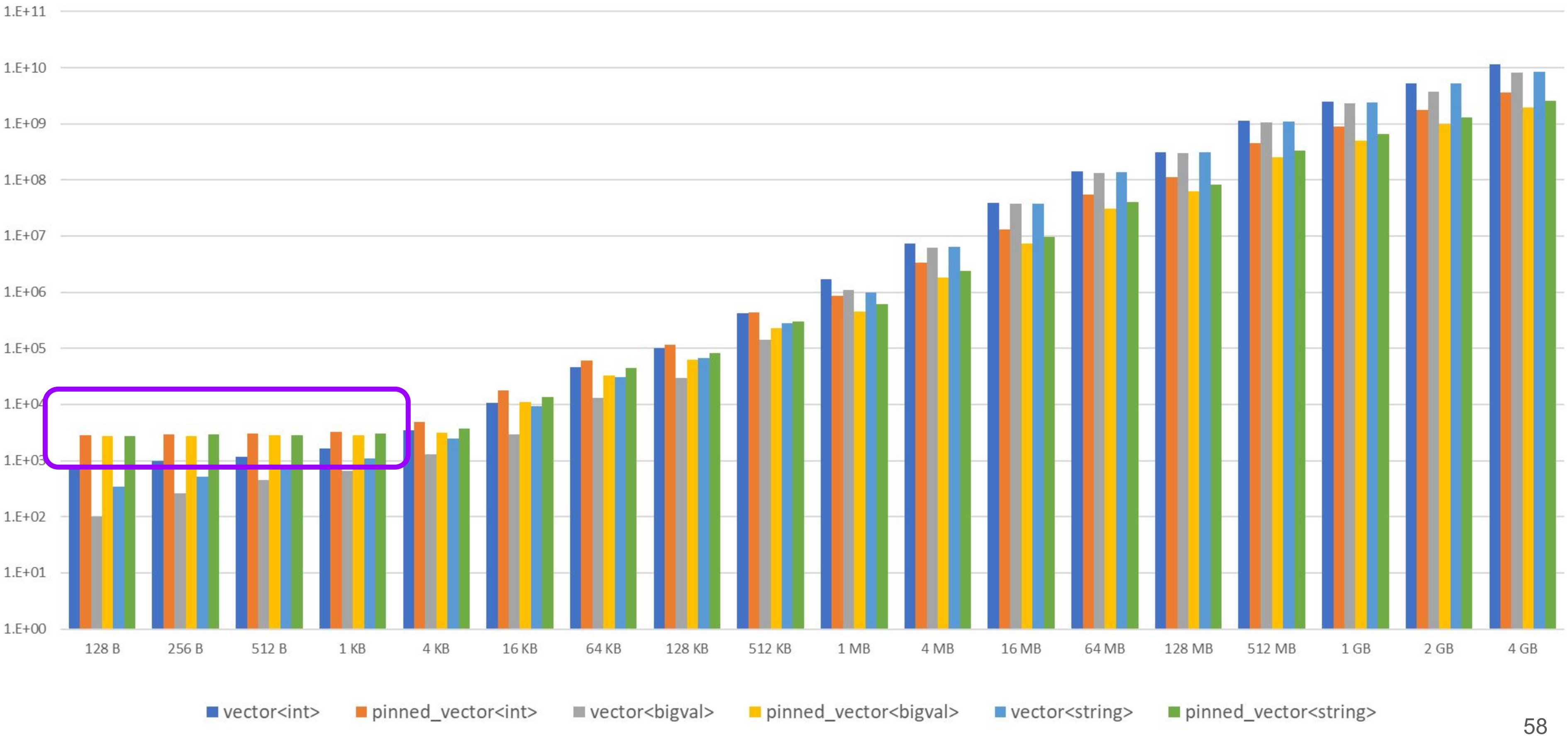


```
struct bigval
{
    double data[10];
};
```

Total Time for `std::string`



Total Time


1.E+11
1.E+10
1.E+09
1.E+08
1.E+07
1.E+06
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
128 B
256 B
512 B
1 KB
4 KB
16 KB
64 KB
128 KB
512 KB
1 MB
4 MB
16 MB
64 MB
128 MB
512 MB
1 GB
2 GB
4 GB
vector<int>
pinned_vector<int>
vector<bigval>
pinned_vector<bigval>
vector<string>
pinned_vector<string>

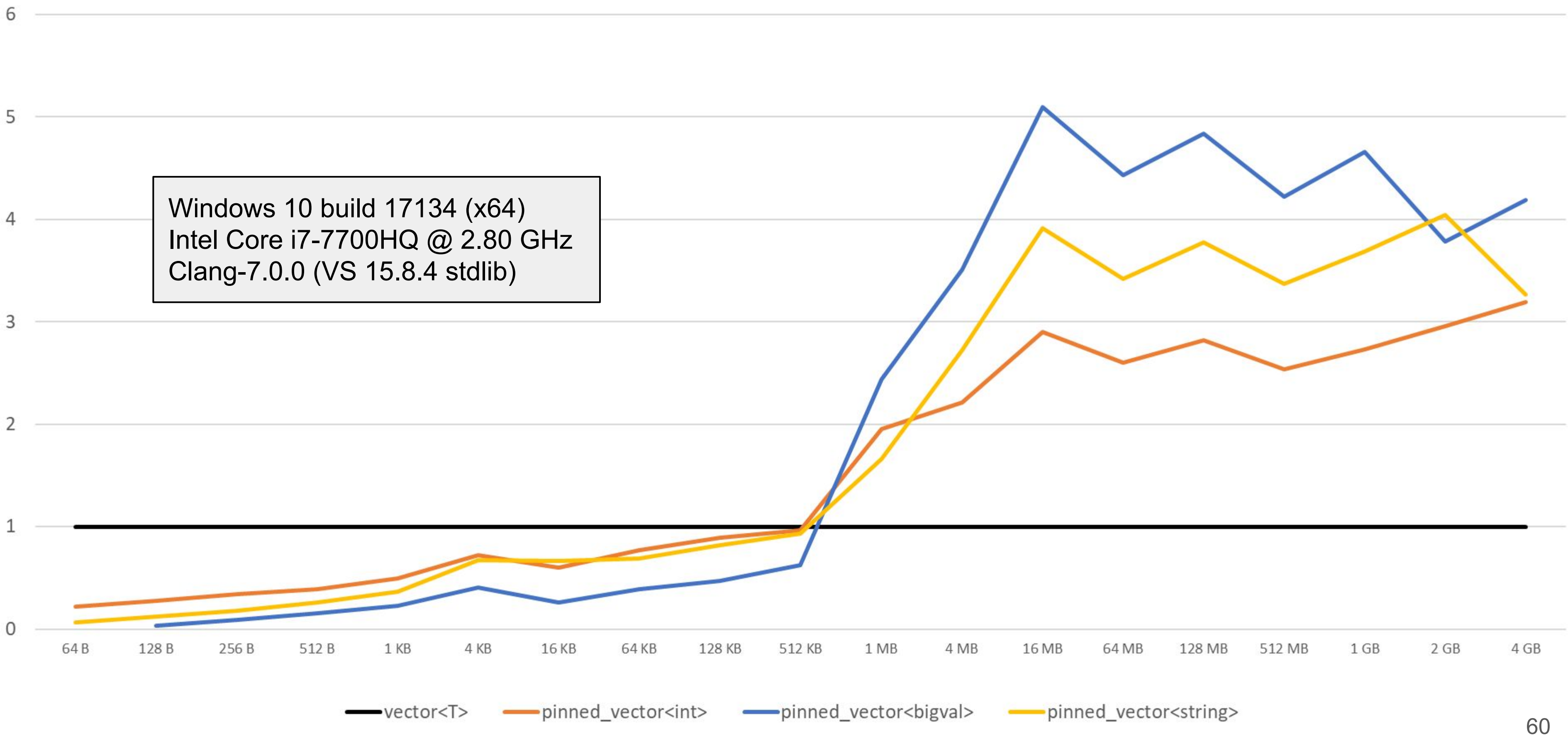
Yes It Is Good

`std::vector` ⚔ `pinned_vector`

Round 3: so how much faster is it?

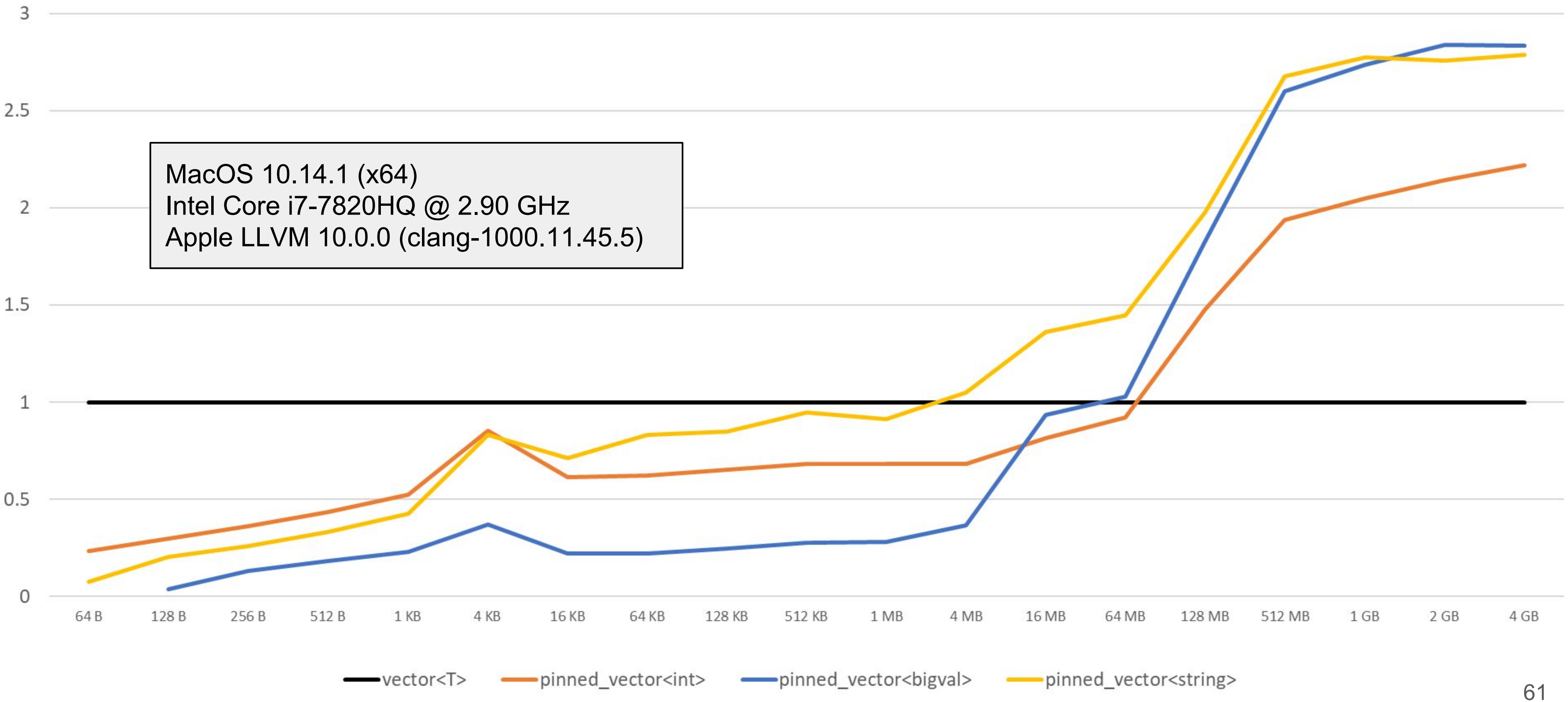
- Normalize the runtimes:
- Treat `vector<T>` time as 1.0
- Rescale `pinned_vector<T>` time based on that

Total Speedup



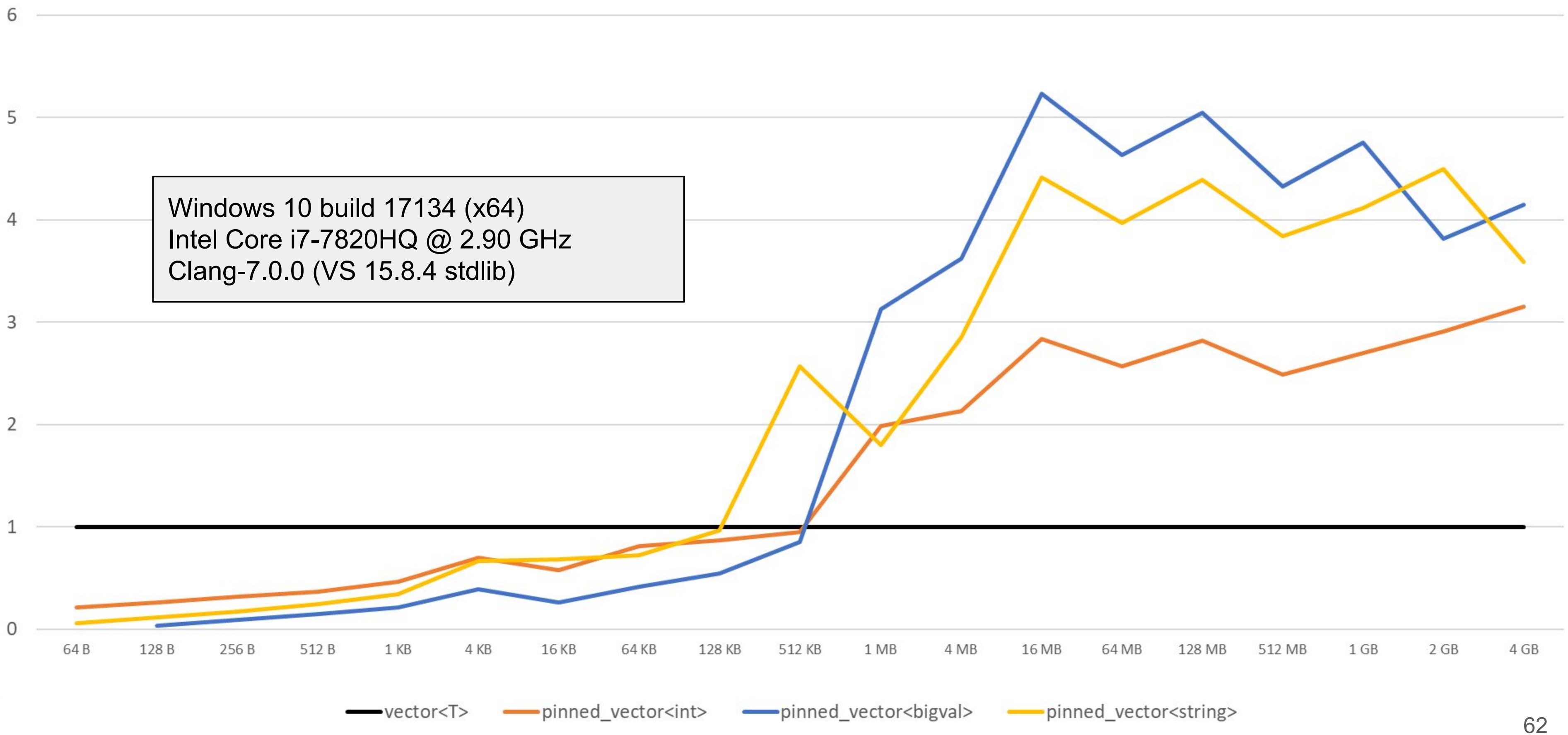
Windows 10 build 17134 (x64)
Intel Core i7-7700HQ @ 2.80 GHz
Clang-7.0.0 (VS 15.8.4 stdlib)

Total Speedup



MacOS 10.14.1 (x64)
Intel Core i7-7820HQ @ 2.90 GHz
Apple LLVM 10.0.0 (clang-1000.11.45.5)

Total Speedup


Windows 10 build 17134 (x64)
Intel Core i7-7820HQ @ 2.90 GHz
Clang-7.0.0 (VS 15.8.4 stdlib)

But Why Is It Good?

`std::vector` ⚔ `pinned_vector`

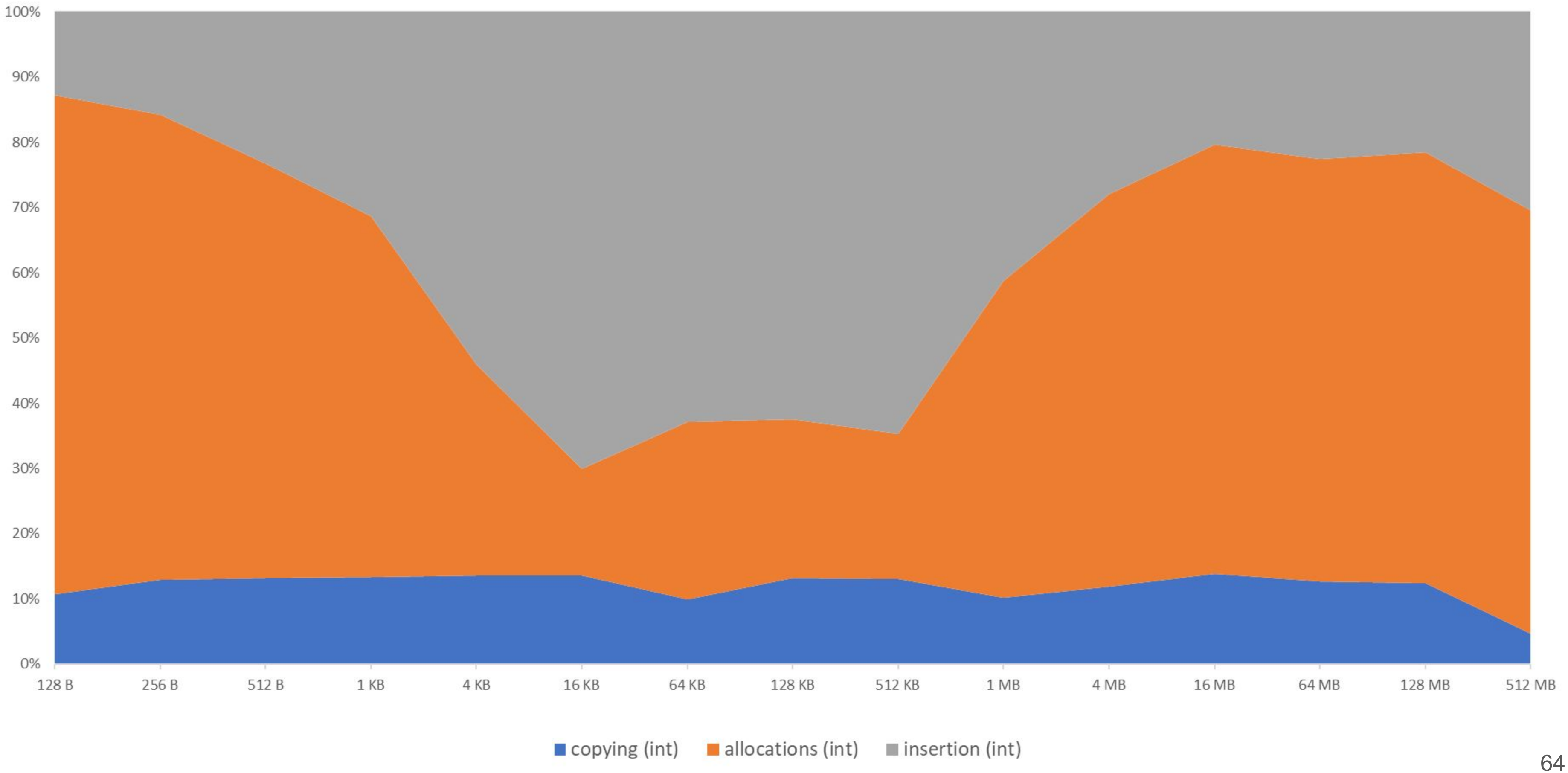
Round 4: where does a vector’s time go?

```
auto v = vector<T, bump_alloc>();

⏱

fill_n(back_inserter(v), n, x);

⏱
```

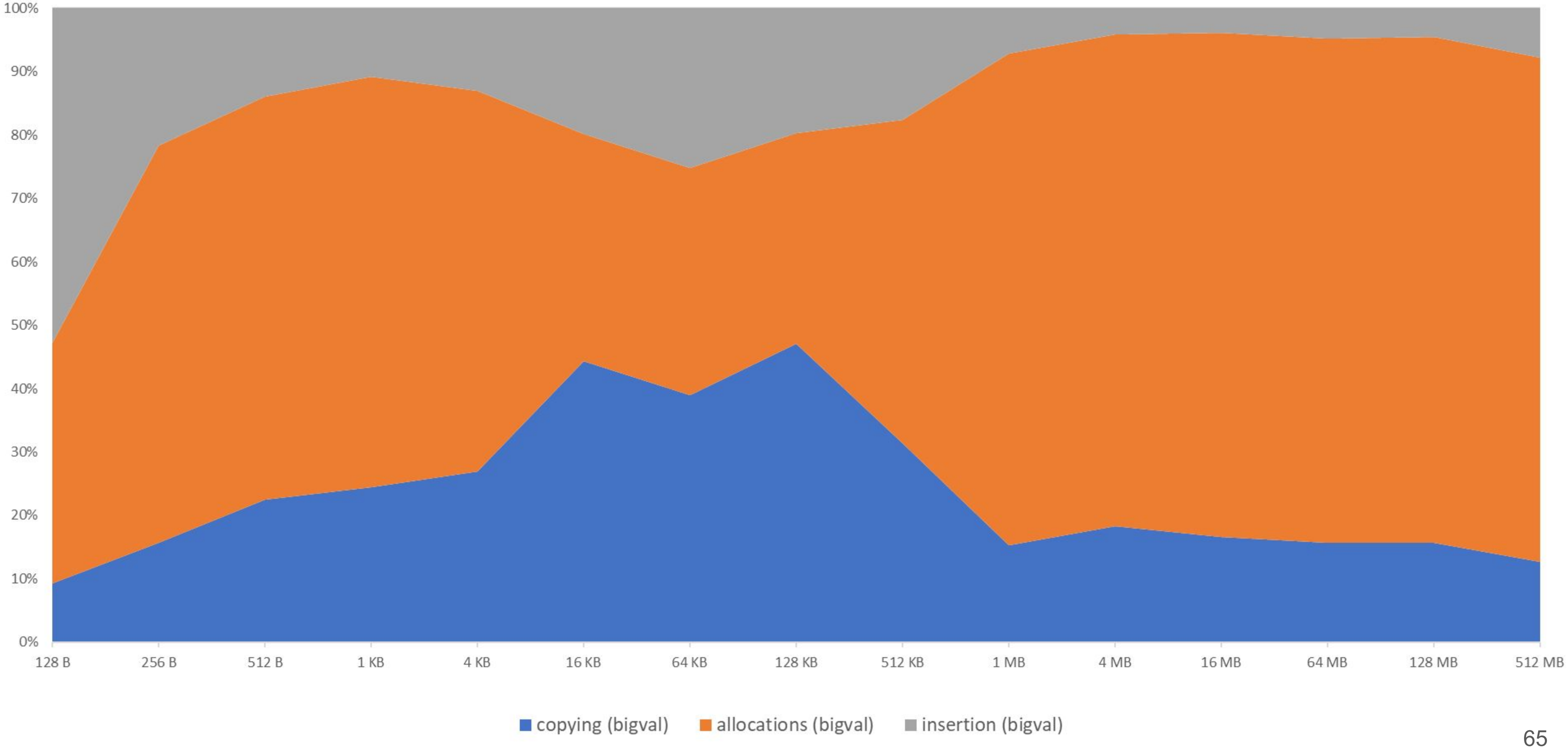
≡ total time - allocations
≡ insertion + copying
≡ baseline + copying

Times for: insertion + allocation + copying

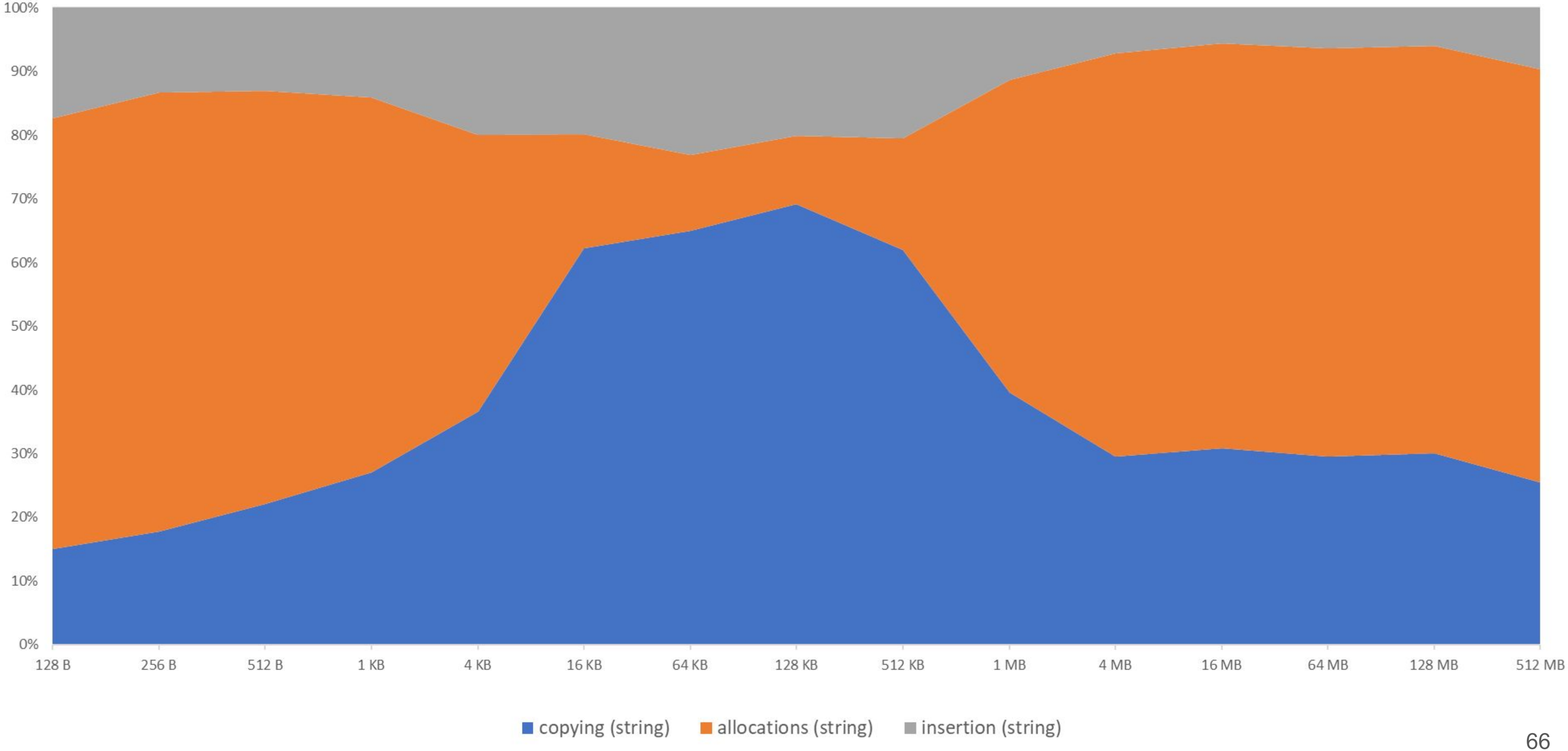
Breakdown of push_back



Breakdown of push_back



Breakdown of push_back



Benchmark Conclusions

`push_back` with preceding `reserve()` roughly equivalent

slower than `std::vector` for small sizes

faster than `std::vector` after a breaking point

achieved by not copying values around

exact numbers vary significantly by system and `value_type`

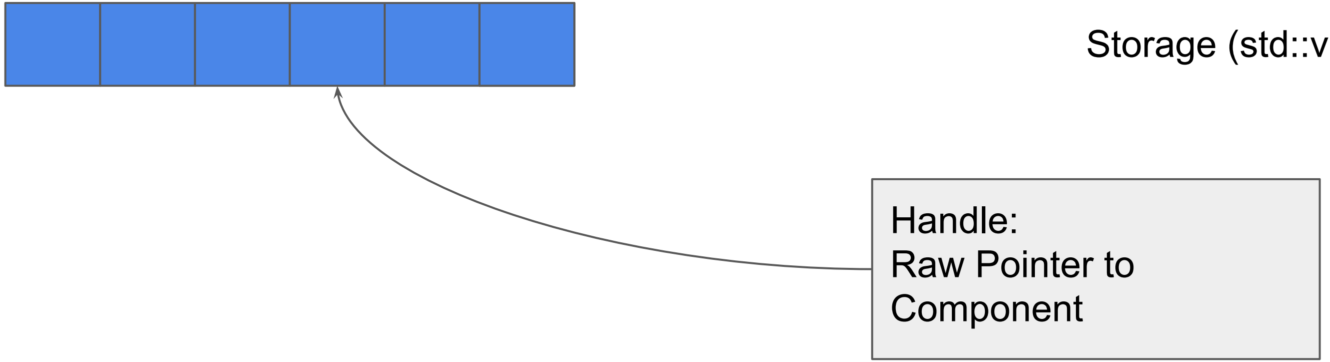
Availability

- Virtual Memory Support
- Desktop
  - Linux
  - macOS
  - Windows
- Mobile
  - Android
  - iOS (reserve limited by physical memory)

Use Case ECS

- ECS: Entity Component System
  - Entity: ID
  - Component: Data only storage
  - System: Uses Components to operate on these
- Data Oriented Design
  - Data oriented design in C++ by Mike Acton
  - Data-oriented design in practice by Stoyan Nikolov
- Mostly used in Games

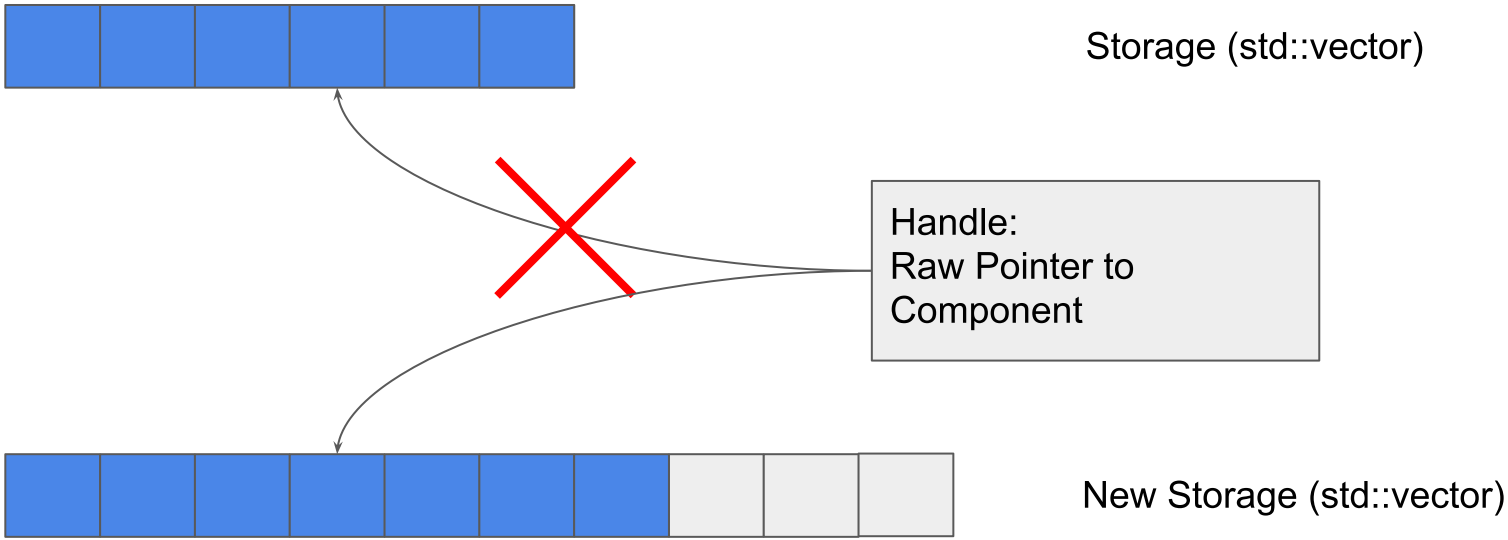
ECS with `std::vector`



Storage (std::vector)

Handle:
Raw Pointer to
Component

ECS with `std::vector`

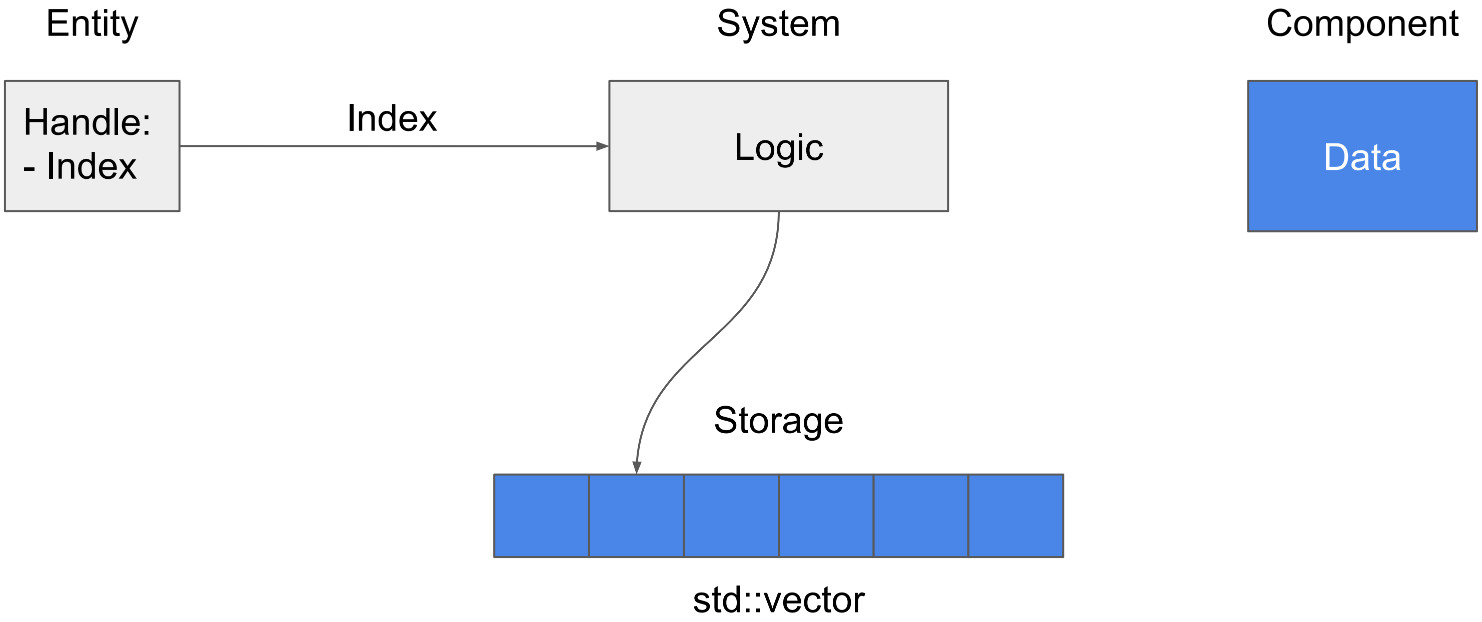


Storage (std::vector)

Handle:
Raw Pointer to
Component

New Storage (std::vector)

ECS with `std::vector`



Entity

Handle:
- Index

Index

System

Logic

Component

Data

Storage

std::vector

ECS with `std::vector`

Pro:

- Dynamic Storage
  - grow/shrink dynamically during runtime

Con:

- Use of Handles
  - e.g. index
  - Indirection

ECS with `std::array`

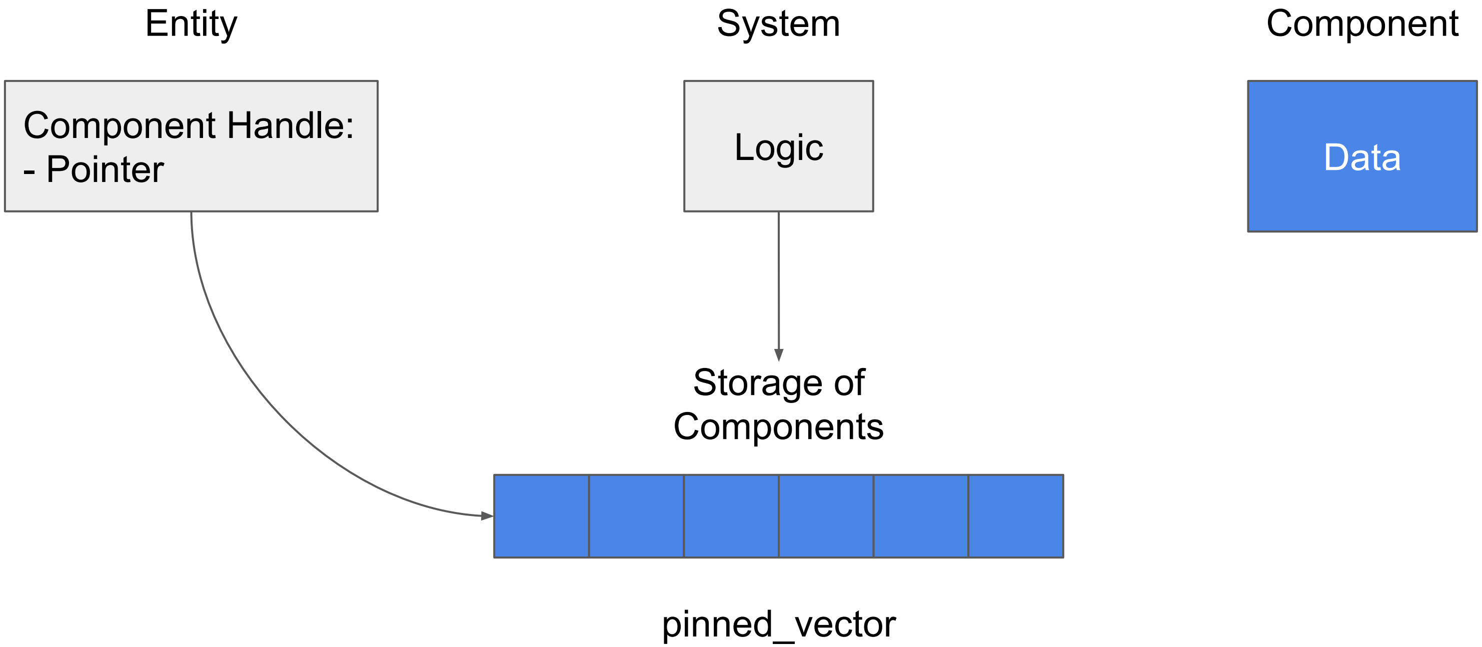
Pro:

- No Indirection

Con:

- Preallocate memory
  => waste of memory
- Need max size
- No dynamic resizing

ECS with `pinned_vector`



Entity

Component Handle:
- Pointer

System

Logic

Component

Data

Storage of Components

pinned_vector

Future Work

- `pinned_stack`
- Shared memory
- Page-fault avoiding hash table

Thank you

Jakob Schweisshelm
@jakouf

Implementation will be released at

https://github.com/mknejp/vmcontainer

Once all the finishing touches are done.

Miro Knejp
@mknejp