Memory & STL

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat
Jialin Cai
Mati Davis
Donglun He
Chandravaran Kunjeti
Heyi Liu
Shufan Liu
Eddy Yang

Poll Everywhere

pollev.com/tqm

- How familiar are you with:
  - ArrayList
  - LinkedList
  - Sets & Maps

Upcoming Due Dates

- Midterm
  - Take-home style on Wednesday 3/1 @ Noon till Friday 3/3 @ noon
  - Logistics released on Course Website

Lecture Outline

- **Memory Hierarchy**
- STL
  - vector
  - list
- Containers & memory

Data Access Time

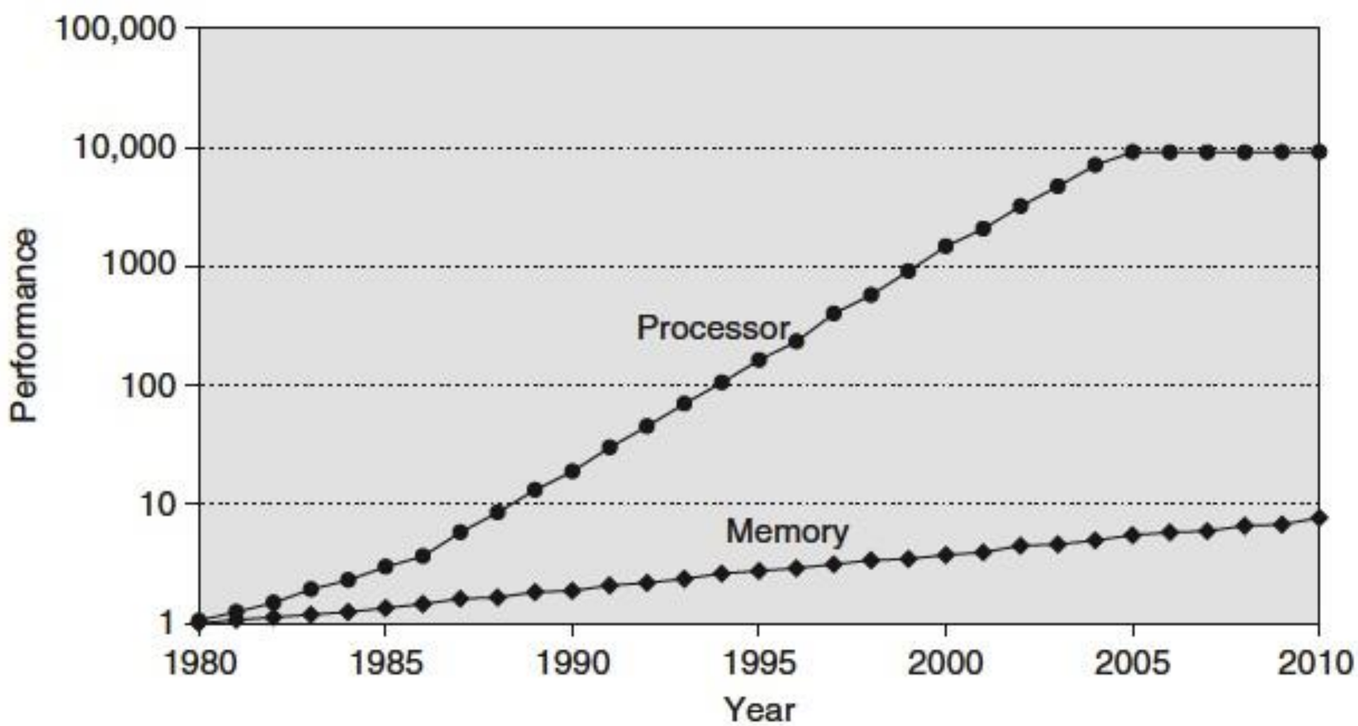
- Data is stored on a physical piece of hardware
- The distance data must travel on hardware affects how long it takes for that data to be processed
- Example: data stored closer to the CPU is quicker to access
  - We see this already with registers. Data in registers is stored on the chip and is faster to access than registers

Memory Hierarchy so far

- So far, we know of three places where we store data
  - CPU Registers
    - Small storage size
    - Quick access time
  - Physical Memory
    - In-between registers and disk
  - Disk
    - Massive storage size
    - Long access time

- As we go further from the CPU, storage space goes up, but access times increase

Processor-Memory Gap



- Processor speed kept growing ~55% per year
- Time to access memory didn't grow as fast ~7% per year
- Memory access would create a bottleneck on performance

Cache

- Pronounced "cash"

- English: A hidden storage space for equipment, weapons, valuables, supplies, etc.

- Computer: Memory with shorter access time used for the storage of data for increased performance. Data is usually either something frequently and/or recently used.
  - Physical memory is a "Cache" of page frames which may be stored on disk
  - In HW1, the buffer in the BufferedFileReader was a "Cache" of file contents

Cache Policies

- Caches are of a fixed size
- Caches need to choose which data gets to be in the cache

- Like page replacement, cache's have their own policies to decide what data to evict/keep
  - LRU is a strategy used for caches
  - Some caches use other policies like tracking frequency of access
  - Generally, caches stores chunks of data together

Principle of Locality

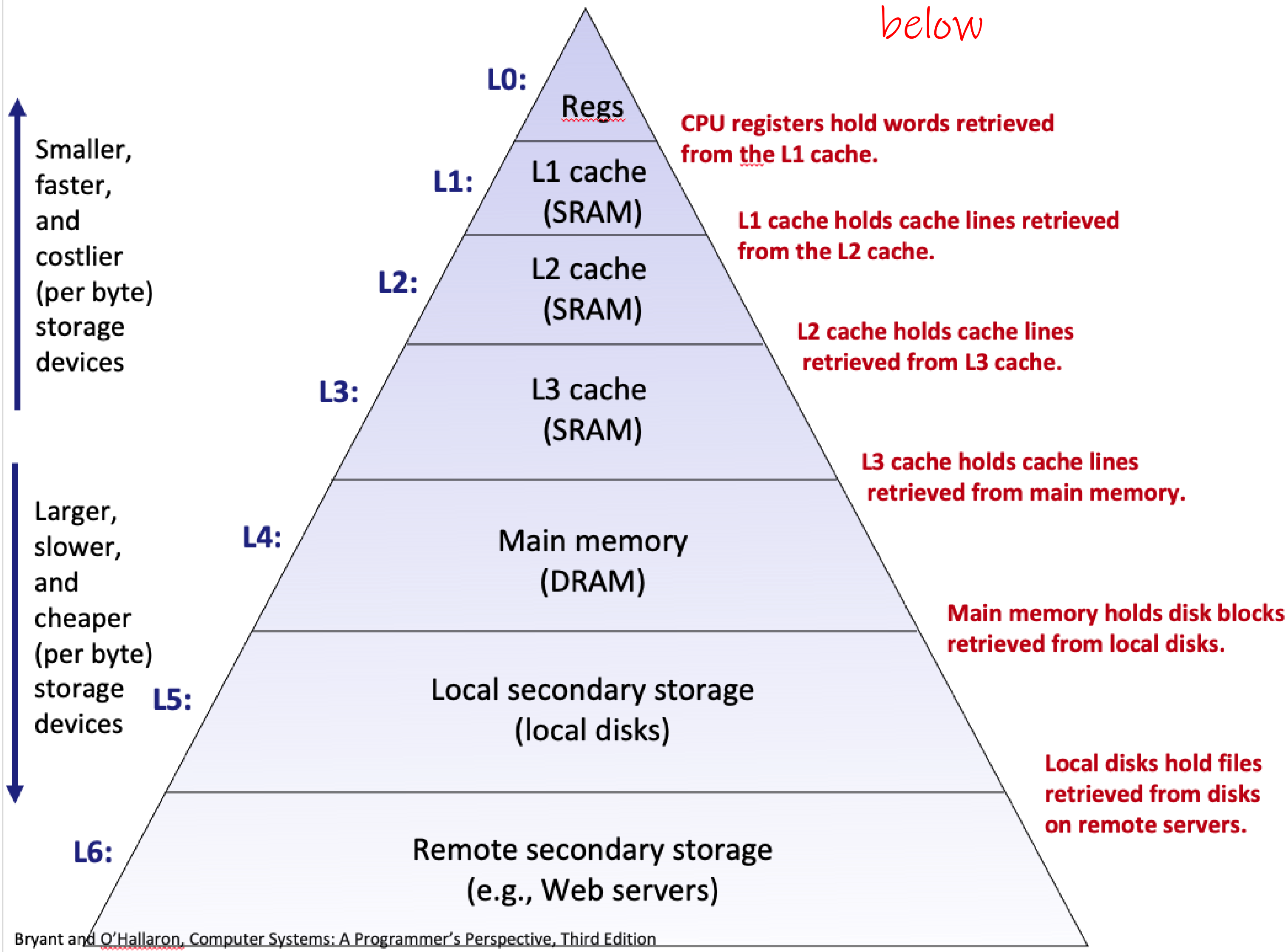
- The tendency for the CPU to access the same set of memory locations over a short period of time

- Two main types:
  - **Temporal Locality**: If we access a portion of memory, we will likely reference it again soon
  - **Spatial Locality**: If we access a portion of memory, we will likely reference memory close to it in the near future.

- Caches take advantage of these tendencies with the cache policies.

Memory Hierarchy

Each layer can be thought of as a "cache" of the layer below



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Details left out

- Virtual Memory
  - COW Fork (Copy On Write)
  - Details about shared process memory
  - Transition Lookaside Buffers (TLB)

- Memory Hierarchy
  - Cache Associativity
  - Writing Policies
  - Instruction Caches
  - DRAM vs SRAM
  - Writing code that consider locality

- A bunch of details that would be system-specific

Lecture Outline

- Memory Hierarchy
- **STL**
  - **vector**
  - **list**
- Containers & memory

C++'s Standard Library

- C++'s Standard Library consists of four major pieces:
  1) The entire C standard library
  2) C++'s input/output stream library
     - `std::cin`, `std::cout`, `stringstreams`, `fstreams`, etc.
  3) C++'s standard template library (**STL**) ☜
     - Containers, iterators, algorithms (sort, find, etc.), numerics
  4) C+'+'s miscellaneous library
     - Strings, exceptions, memory allocation, localization

STL Containers ☺

- A container is an object that stores (in memory) a collection of other objects (elements)
  - Implemented as class templates, so hugely flexible
  - More info in *C++ Primer* §9.2, 11.2

- Several different classes of container
  - Sequence containers (`vector`, `deque`, `list, ...`)
  - Associative containers (`set`, `map`, `multiset`, `multimap`, `bitset, ...`)
  - Differ in algorithmic cost and supported operations

STL Containers

- STL containers store by *value*, not by *reference*
  - When you insert an object, the container makes a *copy*
  - If the container needs to rearrange objects, it makes copies
    - *e.g.* if you sort a `vector`, it will make many, many copies
    - *e.g.* if you insert into a `map`, that may trigger several copies

  - What if you don't want this (disabled copy constructor or copying is expensive)?
    - You can insert a wrapper object with a pointer to the object
      - We'll learn about these "smart pointers" soon

STL `vector`

- A generic, dynamically resizable array
  - https://cplusplus.com/reference/vector/vector/
  - Elements are store in *contiguous* memory locations ← Like a normal C array!
    - Elements can be accessed using pointer arithmetic if you'd like
    - Random access is O(1) time ← Pointer arithmetic, then acces
  - Adding/removing from the end is cheap (amortized constant time)
  - Inserting/deleting from the middle or start is expensive (linear time) ← Need to shift all of the elements in the array
  - Most common member function: **`push_back()`**
    - Adds an element to the end of the vector

Poll Everywhere

pollev.com/tqm

- What is the final value of v by the end of the main() function?

```
void mystery(vector<int> v) {
  v.push_back(5950);
}

int main(int argc, char **argv) {
  vector<int> v;
  mystery(v);

  cout << v.size() << endl;

  for (unsigned int i = 0; i < v.size(); i++) {
    cout << v.at(i) << endl;
  }

  return EXIT_SUCCESS;
}
```

A. [595]
B. [ ] // empty
C. nullptr
D. Program does not reach the end of main()
E. We're lost…

Our Tracer Class

Two fields:
value
id (unique to the instance)

Sets id_ to be unique for each instance

- Wrapper class for a `char` `value_`
  - Also holds unique `unsigned int` `id_` (increasing from `0`)
  - Default ctor, cctor, dtor, `op=`, `op<` defined
  - `friend` function `operator<<` defined
  - Private helper method **`PrintID()`** to return `"(id_,value_)"` as a string
  - Class and member definitions can be found in `Tracer.h` and `Tracer.cc`
- Useful for tracing behaviors of containers
  - All methods print identifying messages
  - Unique `id_` allows you to follow individual instances

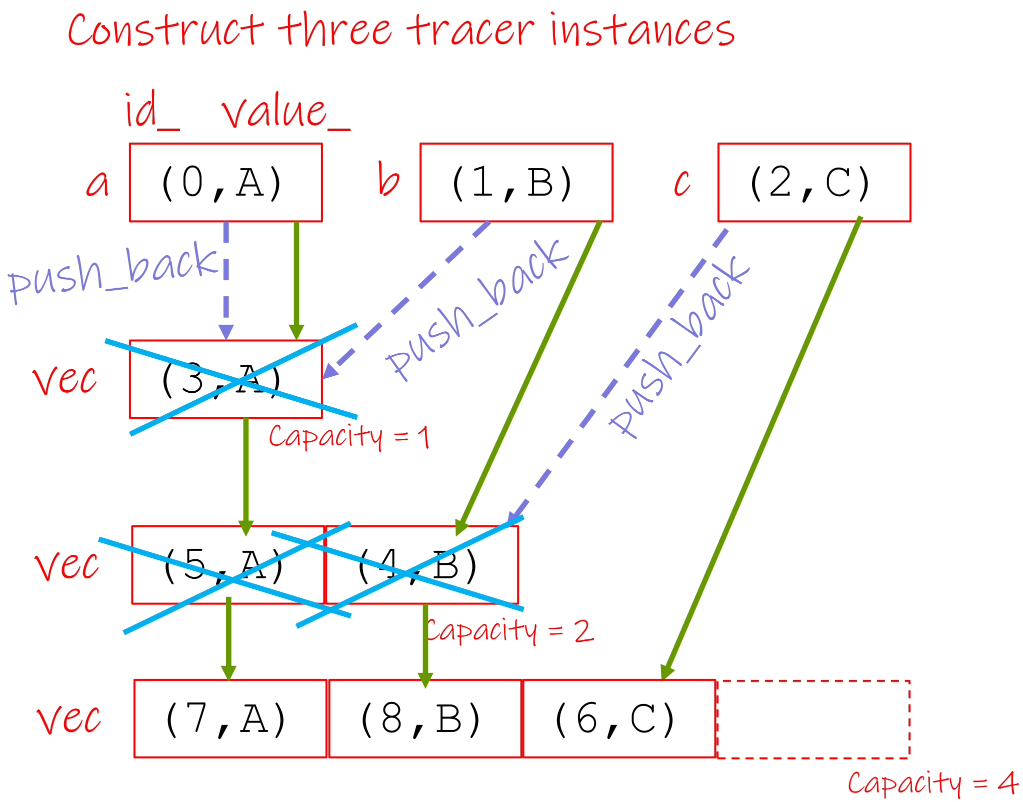
vector/Tracer Example

vectorfun.cc

```
#include <iostream>
#include <vector>          // Most containers are declared in library of same name
#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
  Tracer a, b, c;          // Construct three tracer instances & empty vector
  vector<Tracer> vec;

  cout << "vec.push_back " << a << endl;
  vec.push_back(a);        // Add tracers to end of vector
  cout << "vec.push_back " << b << endl;
  vec.push_back(b);
  cout << "vec.push_back " << c << endl;
  vec.push_back(c);

  cout << "vec[0]" << endl << vec[0] << endl;
  cout << "vec[2]" << endl << vec[2] << endl;
                           // Array syntax to access elements
  return EXIT_SUCCESS;
}
```

Why All the Copying?

Key:
Copy constructor
Destructed



Push back calls	Tracers constructed
0	3 (a,b,c)
1	4
2	6
3	9
4	10
5	15

Note:
- Capacity doubles each time capacity is reached
- Exact construction order when resizing is not important

Other `vector` utilities

- `pop_back()`
  - Removes the last element of the `vector`
- `operator[](index)`
  - Access an element at a specific index of the vector
- `at(index)`
  - Same as above, but throws an exception on invalid index
- `clear()`
  - Removes all elements currently in the vector
- A bunch more:
  - https://www.cplusplus.com/reference/vector/vector/

STL `iterator`

Specific to container and & element type

- Each container class has an associated iterator class (*e.g.* `vector<int>::iterator`) used to iterate through elements of the container
  - http://www.cplusplus.com/reference/std/iterator/
  - Iterator range is from `begin` up to `end` i.e., [`begin` , `end`)
    - `end` is one past the last container element!
  - Some container iterators support more operations than others
    - All can be incremented (`++`), copied, copy-constructed
    - Some can be dereferenced on RHS (*e.g.* `x = *it;`)
    - Some can be dereferenced on LHS (*e.g.* `*it = x;`)
    - Some can be decremented (`--`)
    - Some support random access (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

iterator Example

vectoriterator.cc

```
#include <vector>

#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
  Tracer a, b, c;
  vector<Tracer> vec;

  vec.push_back(a);
  vec.push_back(b);
  vec.push_back(c);

  cout << "Iterating:" << endl;
  vector<Tracer>::iterator it;
  for (it = vec.begin(); it < vec.end(); it++) {
    cout << *it << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

First element

One past the end

Dereference to access element

Increment to next element

Type Inference (C++11)

- The auto keyword can be used to infer types
  - Simplifies your life if, for example, functions return complicated types
  - The expression using auto must contain explicit initialization for it to work

```
// Calculate and return a vector
// containing all factors of n
std::vector<int> Factors(int n);

void foo(void) {
  // Manually identified type
  std::vector<int> facts1 =
    Factors(324234);

  // Inferred type
  auto facts2 = Factors(12321);

  // Compiler error here
  auto facts3;
}
```

Compiler knows return value of Factors()

?????????

No information to infer type

auto and Iterators

- Life becomes much simpler!

```
for (vector<Tracer>::iterator it = vec.begin(); it < vec.end(); it++) {
  cout << *it << endl;
}
```

```
for (auto it = vec.begin(); it < vec.end(); it++) {
  cout << *it << endl;
}
```

Look at all this space!!!

Another beautiful feature of C++ ☺

Range `for` Statement (C++11)

- Syntactic sugar similar to Java's `foreach`

```
for ( declaration : expression ) {
  statements
}
```

  - *`declaration`* defines loop variable
  - *`expression`* is an object representing a sequence
    - Strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators

str = sequence of characters

```
// Prints out a string, one
// character per line
std::string str("hello");

for ( auto c : str ) {
  std::cout << c << std::endl;
}
```

Updated `iterator` Example

vectoriterator_2011.cc

```
#include <vector>

#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
  Tracer a, b, c;
  vector<Tracer> vec;

  vec.push_back(a);
  vec.push_back(b);
  vec.push_back(c);

  cout << "Iterating:" << endl;
  // "auto" is a C++11 feature not available on older compilers
  for (auto& p : vec) {
    cout << p << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

Look at how much more simplified this is!
No begin(), end(), or dereferencing! **:O**

STL Algorithms

- A set of functions to be used on ranges of elements
  - Range: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
  - General form: `algorithm(begin, end, ...);`

Rest depends on the algo

Takes a range of a sequence to operate on

- Algorithms operate directly on range *elements* rather than the containers they live in
  - Make use of elements' copy ctor, =, ==, !=, <
  - Some do not modify elements
    - *e.g.* **find**, **count**, **for_each**, **min_element**, **binary_search**
  - Some do modify elements
    - *e.g.* **sort**, **transform**, **copy**, **swap**

Appropriate operator(s) must be defined for the element to use an STL algorithm

Algorithms Example

vectoralgos.cc

```
#include <vector>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer& p) {
  cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
  Tracer a, b, c;
  vector<Tracer> vec;

  vec.push_back(c);
  vec.push_back(a);
  vec.push_back(b);
  cout << "sort:" << endl;
  sort(vec.begin(), vec.end());
  cout << "done sort!" << endl;
  for_each(vec.begin(), vec.end(), &PrintOut);
  return 0;
}
```

Not in order ☹

Sort elements from [vec.begin(), vec.end())

Runs function on each element.
In this case, prints out each element

STL `list`

- A generic doubly-linked list
  - https://cplusplus.com/reference/list/list/
  - Elements are ***not*** stored in contiguous memory locations
    - Does not support random access (*e.g.* cannot do `list[5]`)
  - Some operations are much more efficient than vectors
    - Constant time insertion, deletion anywhere in list
      - **`push_front()`** and **`pop_front()`** now exist!
      - Can iterate forward or backwards
  - Has a built-in sort member function
  - Doesn't copy! Manipulates list structure instead of element values

Iterate backward: --
Iterate forward: ++

- **<u>Different containers have different functions available</u>**

`list` Example

listexample.cc

```
#include <list>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer& p) {
  cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
  Tracer a, b, c;
  list<Tracer> lst;

  lst.push_back(c);
  lst.push_back(a);
  lst.push_back(b);
  cout << "sort:" << endl;
  lst.sort();
  cout << "done sort!" << endl;
  for_each(lst.begin(), lst.end(), &PrintOut);
  return 0;
}
```

Use case is similar to vector, but internal implementation is different

Won't copy elements, just modifies the next and prev pointers

Lecture Outline

- Memory Hierarchy
- STL
  - vector
  - list
- **Containers & memory**

Choosing a Container

- A common problem in CS (probably done in 5940/5960 more) is choosing which data structure to use for a certain problem.

- You need to consider:
  - How data is stored in that container
  - How our program interacts with that data

Poll Everywhere

pollev.com/tqm

- If I wanted to maintain a sequence of numbers where I very often had to add and remove things from the front, what should I theoretically use?

A. vector
B. list
C. Neither one is particularly better than the other
D. We're lost…

Poll Everywhere

pollev.com/tqm

- If I wanted to maintain a sequence of numbers where I very often had to access elements via an index, which should I theoretically use?

A. **vector**

B. **list**

C. **Neither one is particularly better than the other**

D. **We're lost…**

Poll Everywhere

pollev.com/tqm

- If I wanted to maintain a sequence of numbers where I repeatedly generated a random number and inserted that number into the sequence so that it was in order… which should I theoretically use?
  - Can assume that insertion is done using a linear search

A. vector

B. list

C. Neither one is particularly better than the other

D. We’re lost…

Poll Everywhere

pollev.com/tqm

- If I wanted to maintain a sequence of numbers where the sequence already has 500,000 numbers, I generate a random index, and remove the number at that index. Repeat till the sequence is empty. Which should I theoretically use?

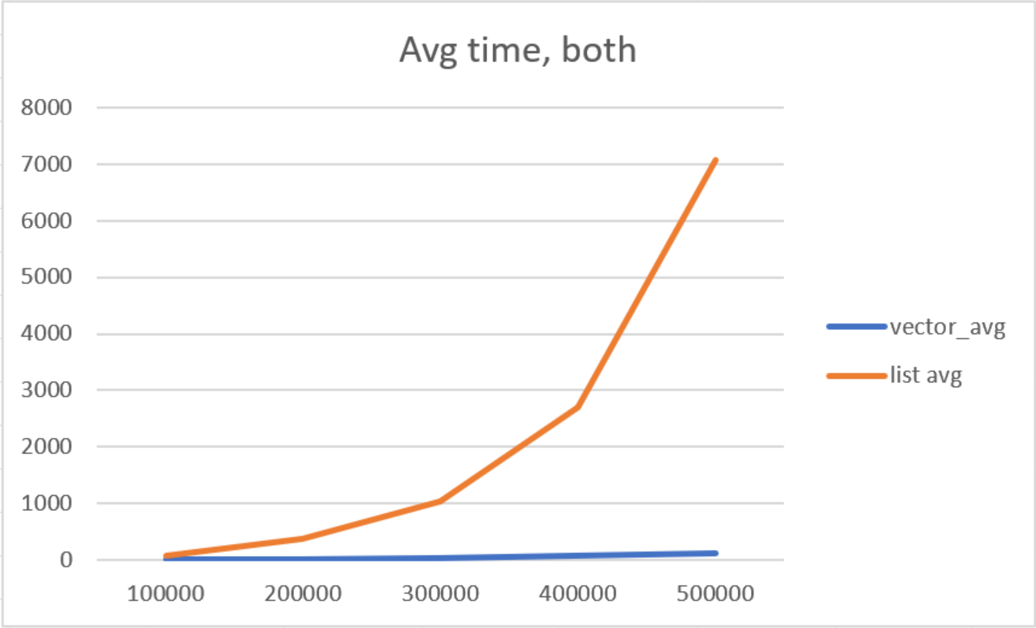
A. vector
B. list
C. Neither one is particularly better than the other
D. We're lost…

Experiment:

- Do the random sorted insertion and random removal and time it for 100,000, 200,00, … 500,000 elements. Average over 5 iterations

- Both do a linear search to insert and to remove

- Result:



- Why? Spatial locality, data in a vector is next to each other. Easy for better cache performance & optimization

vector

- The "default" container for storing a sequence of data is a vector

- Much better optimization and cache performance for vector

- You should almost always use a vector instead of a list
- If you think you may want to use a list…
- Use a vector
- If you think really hard and are sure you want a list…
- Ok fine, you can use a list.