File I/O and The Operating System

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

pollev.com/tqm

- How are you? Any feedback?

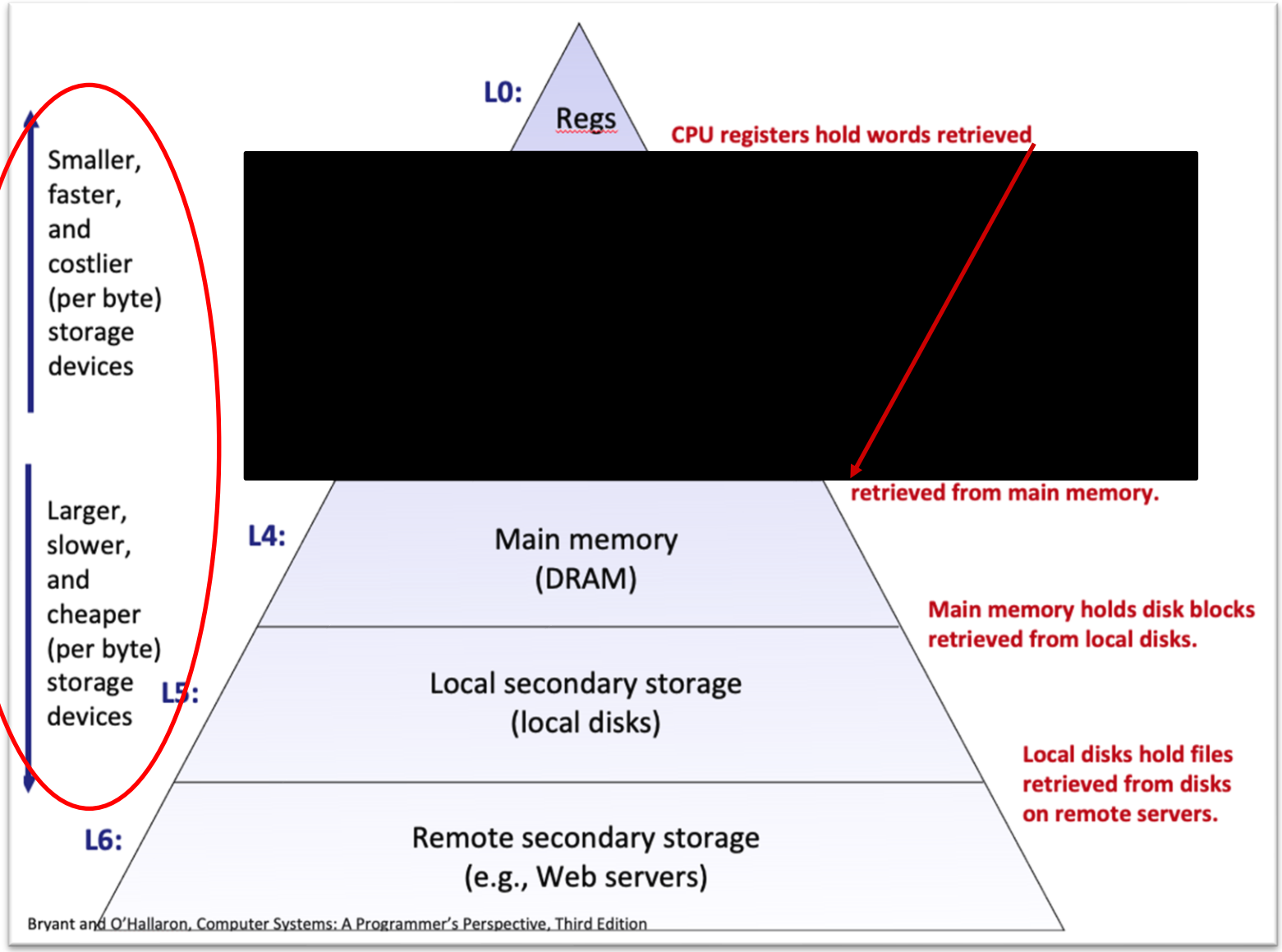
Administrivia

- HW03 Due Tomorrow (was due Tuesday)
  - We are grading this on style, so please try to clean up your code if you didn't do any style.
  - Can look at the rubric we had from HW01 and code quality doc.
  - (note that rubric for HW03 will have more items then HW01. HW03 is C++, HW01 is mostly C)
- HW04 posted Yesterday
  - Should be less work than HW03 I hope?
  - You implement some file reader objects
- Check-in Due posted tonight / tomorrow

Lecture Outline

- Memory Hierarchy Overview
- Buffering
- Memory Locality & Caching

Memory Hierarchy (So Far)



Principle of Locality

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

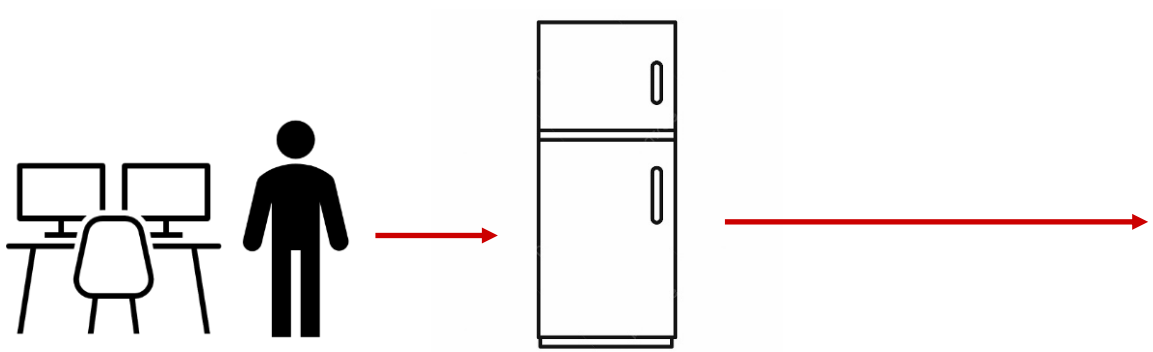
- Two main types:
  - **Temporal Locality**: If we access data in memory/storage, we will likely reference it again soon.

  - **Spatial Locality**: If we access data in memory/storage, we will likely reference data close to it soon.

- Data that is accessed frequently can be stored in hardware that is quicker to access.

Locality Analogy

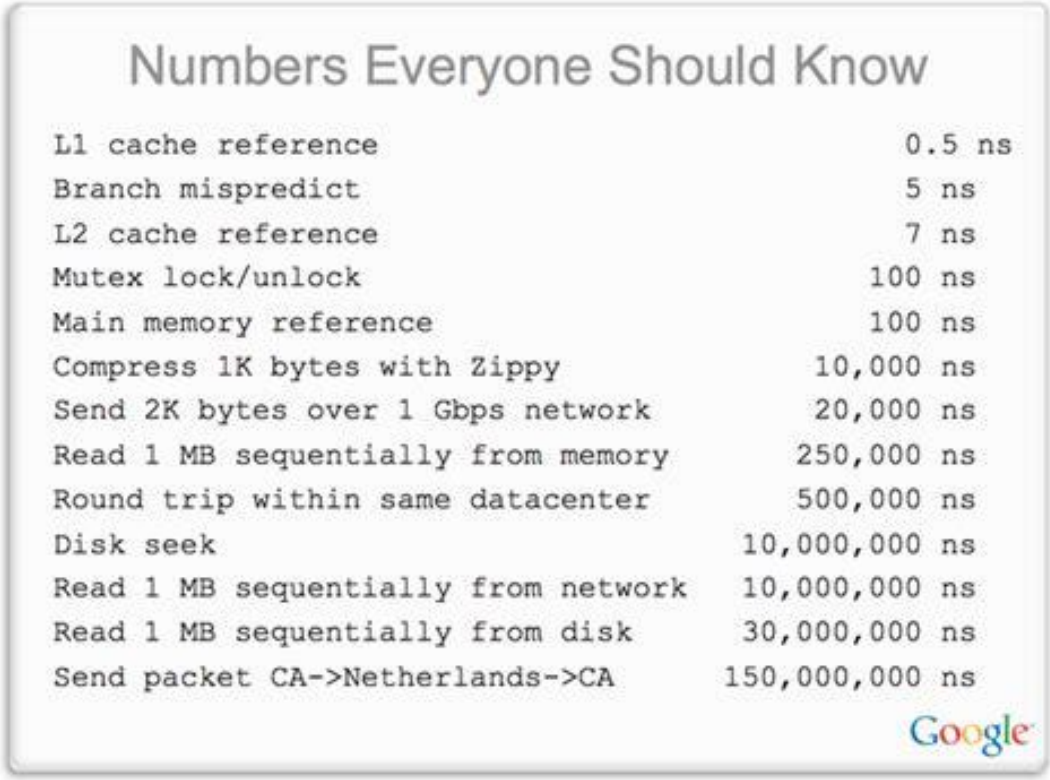
- If we are at home and we are hungry, where do we get food from?
  - We get it from our refrigerator!
  - If the refrigerator is empty, we go to the grocery store
  - When at the grocery store, we don't just get what we want right now, but also get other things we think we want in the near future (so that it will be in our fridge when we want it)



Numbers Everyone Should Know

- There is a set of numbers that called “numbers everyone you should know”

- From Jeff Dean in 2009
- Numbers are out of date but the relative orders of magnitude are about the same



- More up to date numbers: https://colin-scott.github.io/personal_website/research/interactive_latency.html

Lecture Outline

- Memory Hierarchy Overview
- Buffering
- Memory Locality & Caching

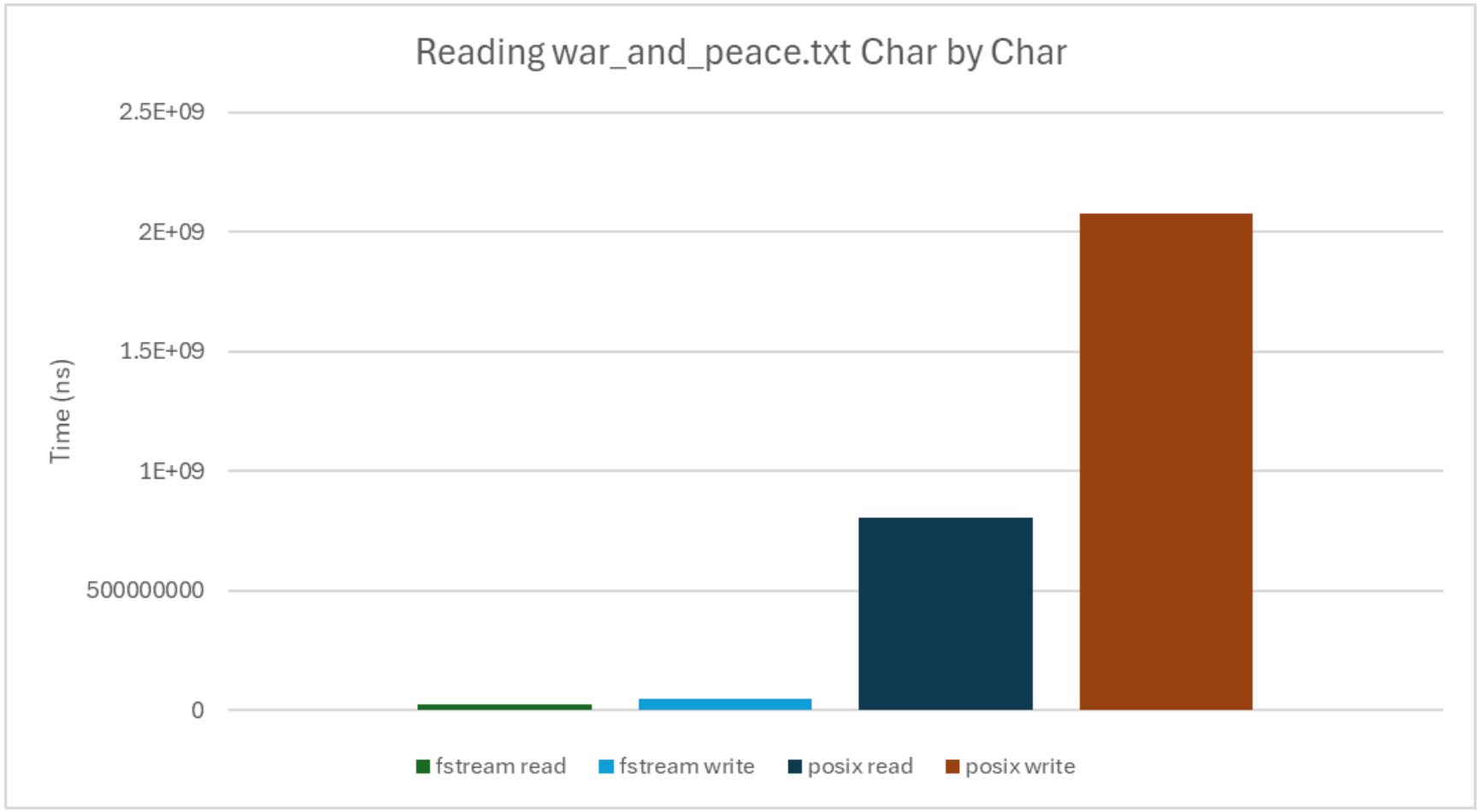
Raise Hands

- Which function implementation do you think is faster?

```
string read_file() {
  char c;
  int fd = open("war_and_peace.txt", O_RDONLY);

  ssize_t res = read(fd, &c, 1);
  string data;
  // 0 means EOF
  while (res != 0) {
    data += c;
    res = read(fd, &c, 1);
  }

  close(fd);
  return data;
}
```

```
string read_file() {
  char c;
  ifstream reader("war_and_peace.txt");

  c = reader.get();
  string data;
  // 0 means EOF
  while (reader) {
    data += c;
    c = reader.get();
  }


  return data;
}
```

Experiment Results



C & C++ streams vs POSIX

- Why are we getting these different outputs?

- Both use different ways of writing to standard out.

  - C++ iostream: user level portable library for **i**nput/**o**utput streams. Should work on any environment that has the C++ standard library
    - E.g. cout, operator<<, endl, cin, operator>>, getline, etc.

  - POSIX C API: **P**ortable **O**perating **S**ystem **I**nterface. Functions that are supported by many operating systems to support many OS-level concepts (Input/Output, networking, processes, pipes, threads…)

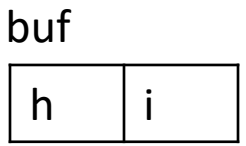
Buffered writing

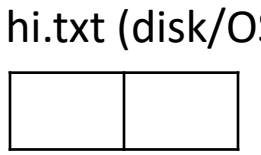
- By default, C++ `iostream` usually uses buffering on top of POSIX:
  - When one writes with **`cout`**, the data being written is copied into a buffer allocated by `C++ iostream` inside your process' address space
  - As some point, once enough data has been written, the buffer will be "flushed" to the operating system.
    - When the buffer fills (often 1024 or 4096 bytes)
  - This prevents invoking the write system call and going to the filesystem too often

Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```



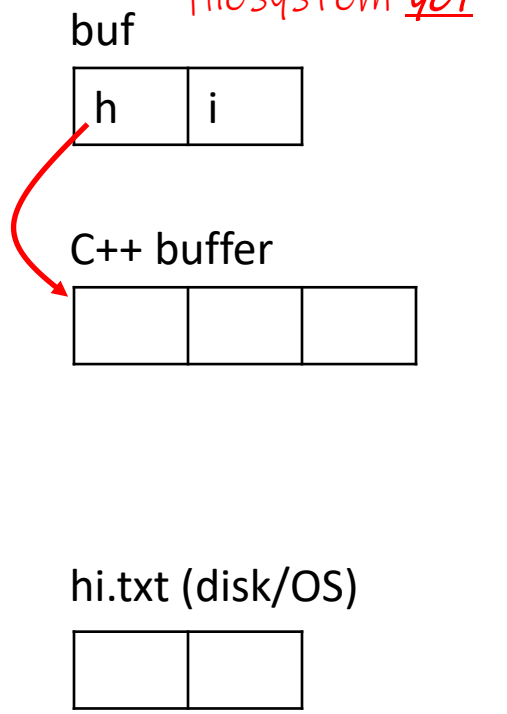


Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

Store 'h' into buffer, so that we do not go to filesystem *yet*

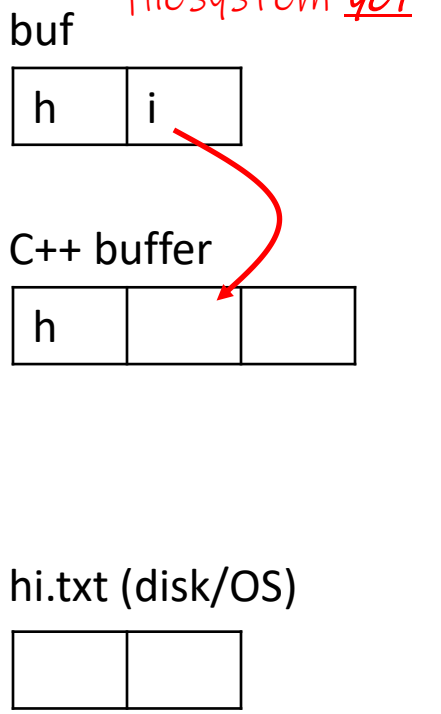


Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

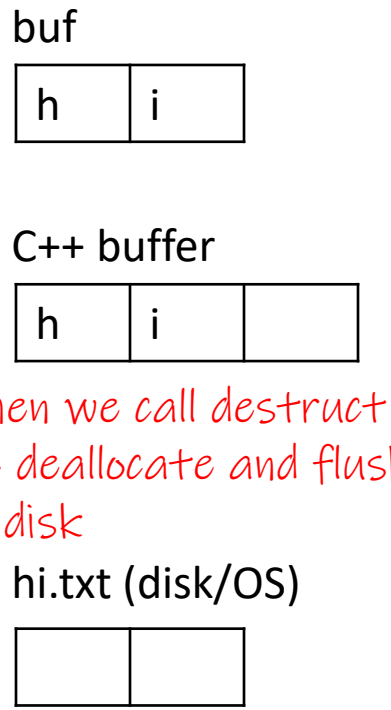
Store 'i' into buffer, so that we do not go to filesystem *yet*



Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```



Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

buf

h	i

hi.txt (disk/OS)

h	i

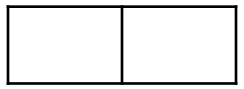
Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf

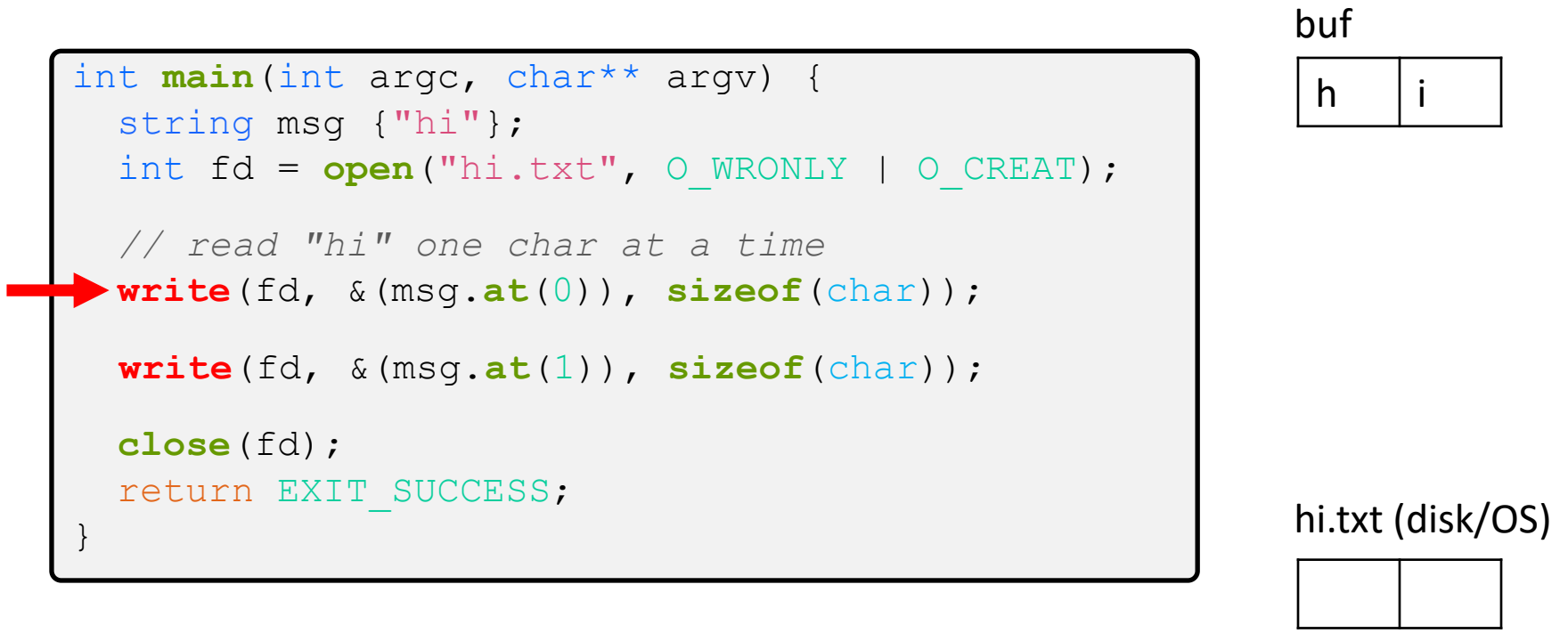
hi.txt (disk/OS)



Unbuffered Writing Example

Arrow signifies what will be executed next

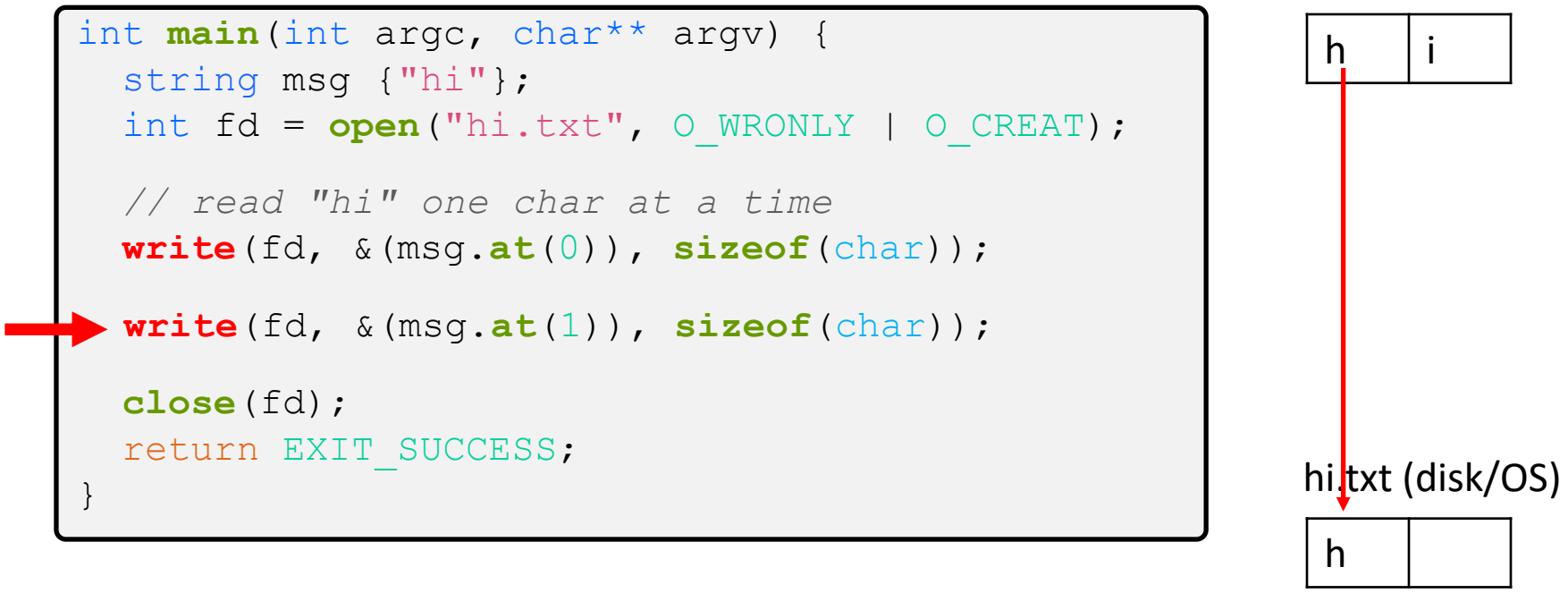
```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

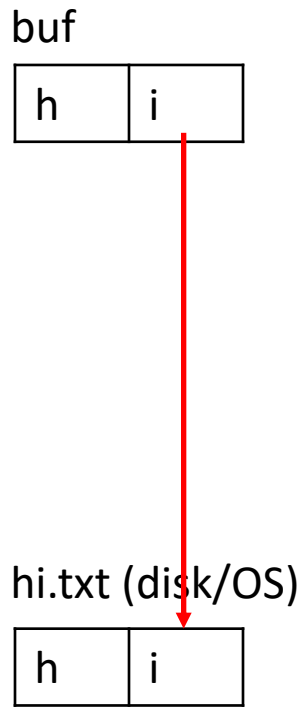
```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf

h	i

Two OS/File system accesses instead of one ☹

hi.txt (disk/OS)

h	i

Buffered Reading

- By default, C++ `fstream` uses buffering on top of POSIX:
  - When one reads with **`fstream`**, a lot of data is copied into a buffer allocated by the `fstream` inside your process' address space
  - Next time you read data, it is retrieved from the buffer
    - This avoids having to invoke a system call again
  - As some point, the buffer will be "refreshed":
    - When you process everything in the buffer (often 1024 bytes)
  - Similar thing happens when you write to a file

Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```

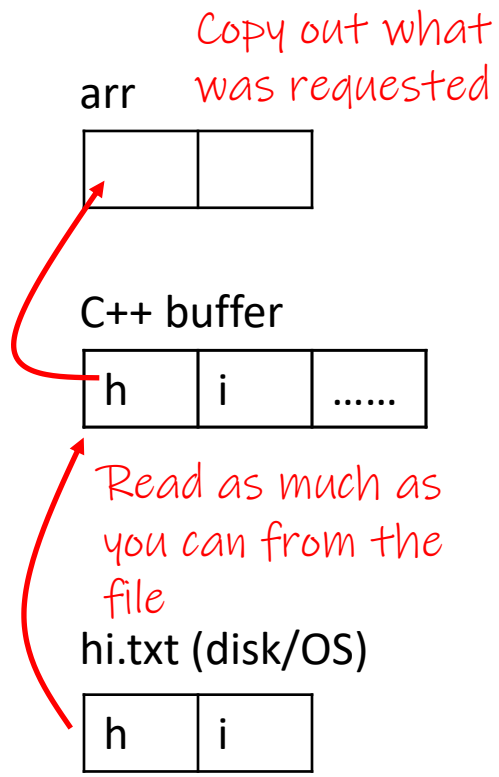
arr

hi.txt (disk/OS)

Buffered Reading Example

Arrow signifies what will be executed next

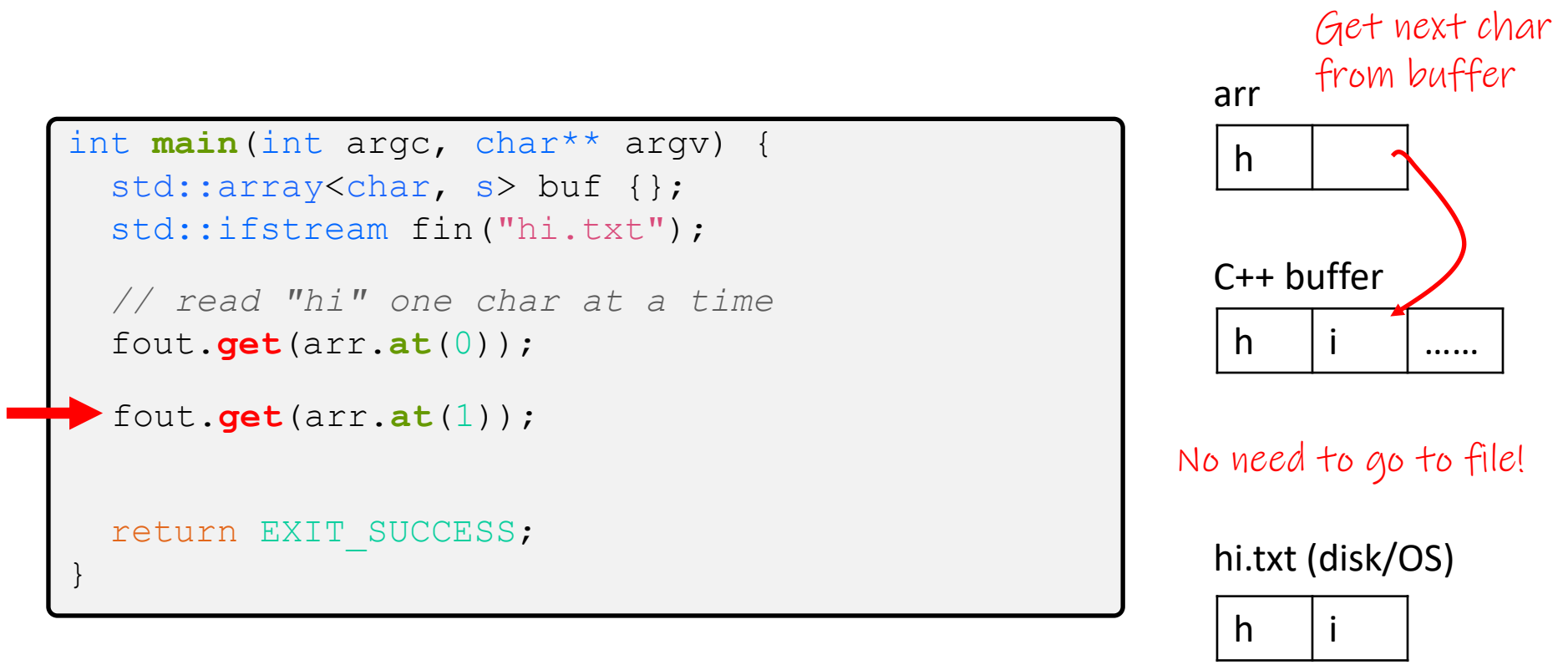
```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```

arr

h	i

C++ buffer

h	i	.....

hi.txt (disk/OS)

h	i

Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```

arr

h	i

hi.txt (disk/OS)

h	i

Raise Hands

- Which function implementation do you think is faster?

```
string read_file() {
  char c;
  int fd = open("war_and_peace.txt", O_RDONLY);

  ssize_t res = read(fd, &c, 1);
  string data;
  // 0 means EOF
  while (res != 0) {
    data += c;
    res = read(fd, &c, 1);
  }

  close(fd);
  return data;
}
```

```
string read_file() {
  array<char, 1024> buf{};
  int fd = open("war_and_peace.txt", O_RDONLY);

  ssize_t res = read(fd, buf.data(), 4096);
  string data;
  // 0 means EOF
  while (res != 0) {
    data += string(buf.data(), res);
    res = read(fd, buf.data(), 4096);
  }

  close(fd);
  return data;
}
```

Raise Hands

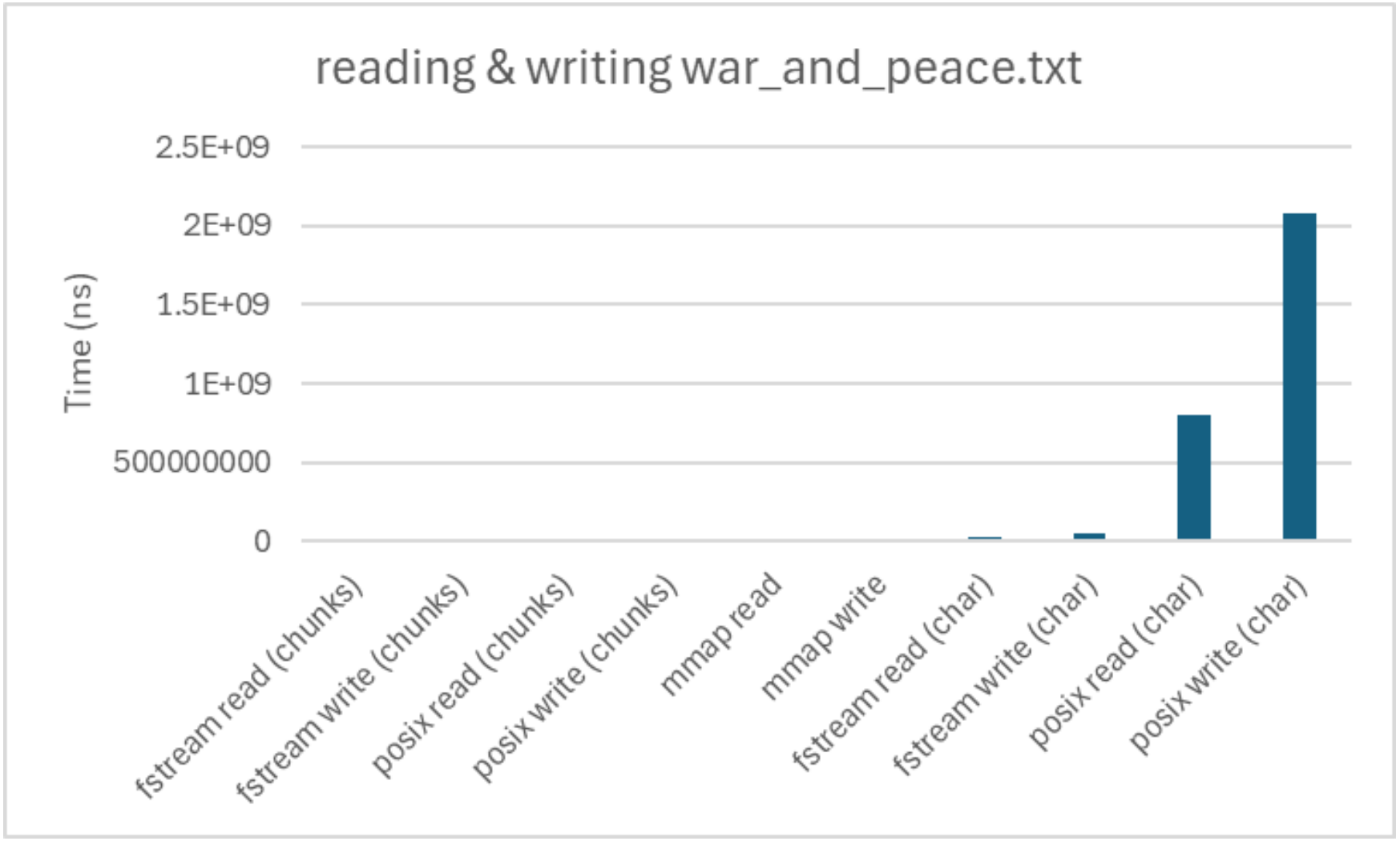
- Which function implementation do you think is faster?

```
string read_file() {
  array<char, 1024> buf{};
  ifstream reader("war_and_peace.txt");

  reader.read(buf, 4096);
  string data;
  // 0 means EOF
  while (reader) {
    data += string(buf.data());
    reader.read(buf, 4096);
  }


  return data;
}
```

```
string read_file() {
  array<char, 1024> buf{};
  int fd = open("war_and_peace.txt", O_RDONLY);

  ssize_t res = read(fd, buf.data(), 4096);
  string data;
  // 0 means EOF
  while (res != 0) {
    data += string(buf.data(), res);
    res = read(fd, buf.data(), 4096);
  }

  close(fd);
  return data;
}
```

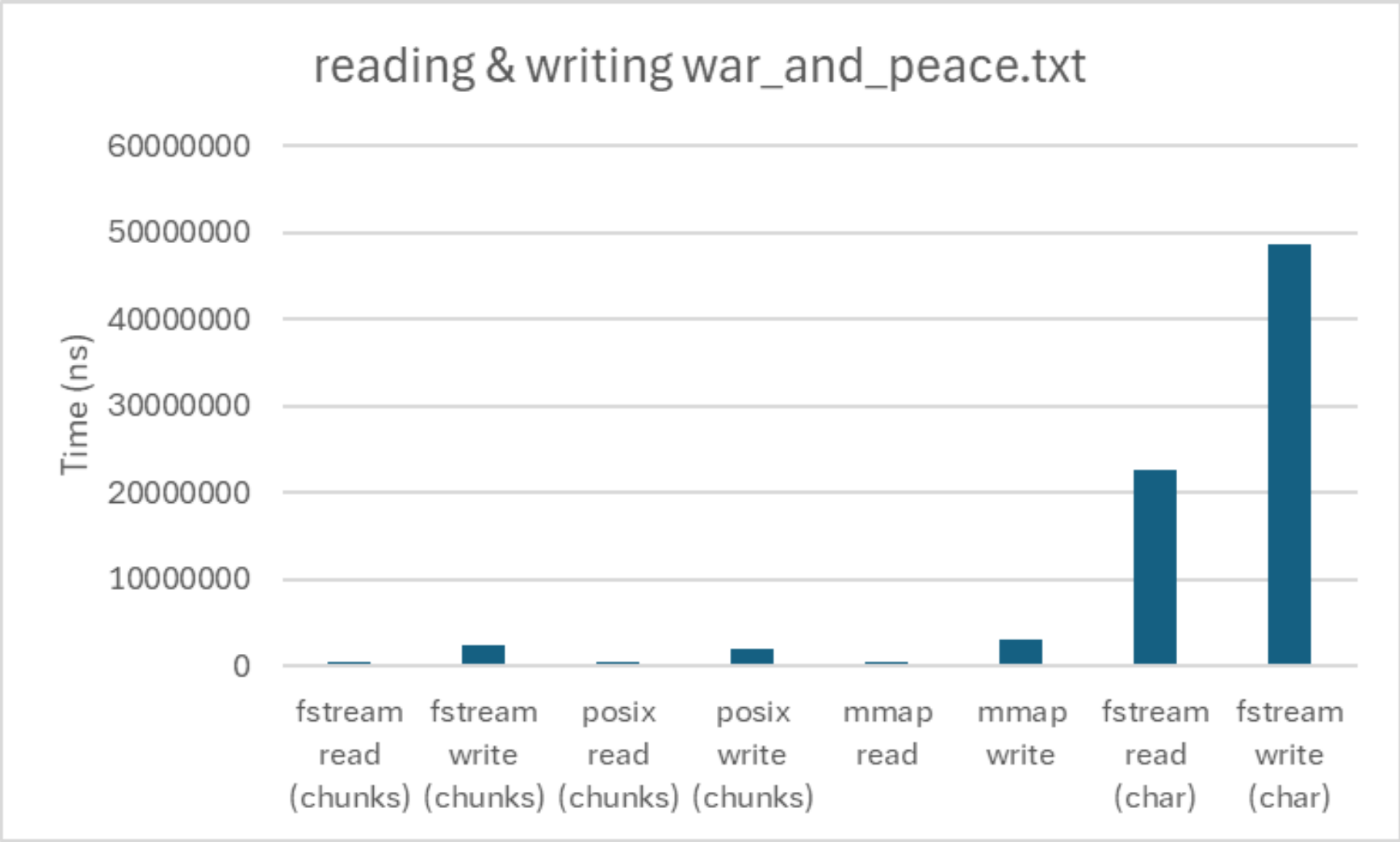
Experiment Results

All Results



Experiment Results

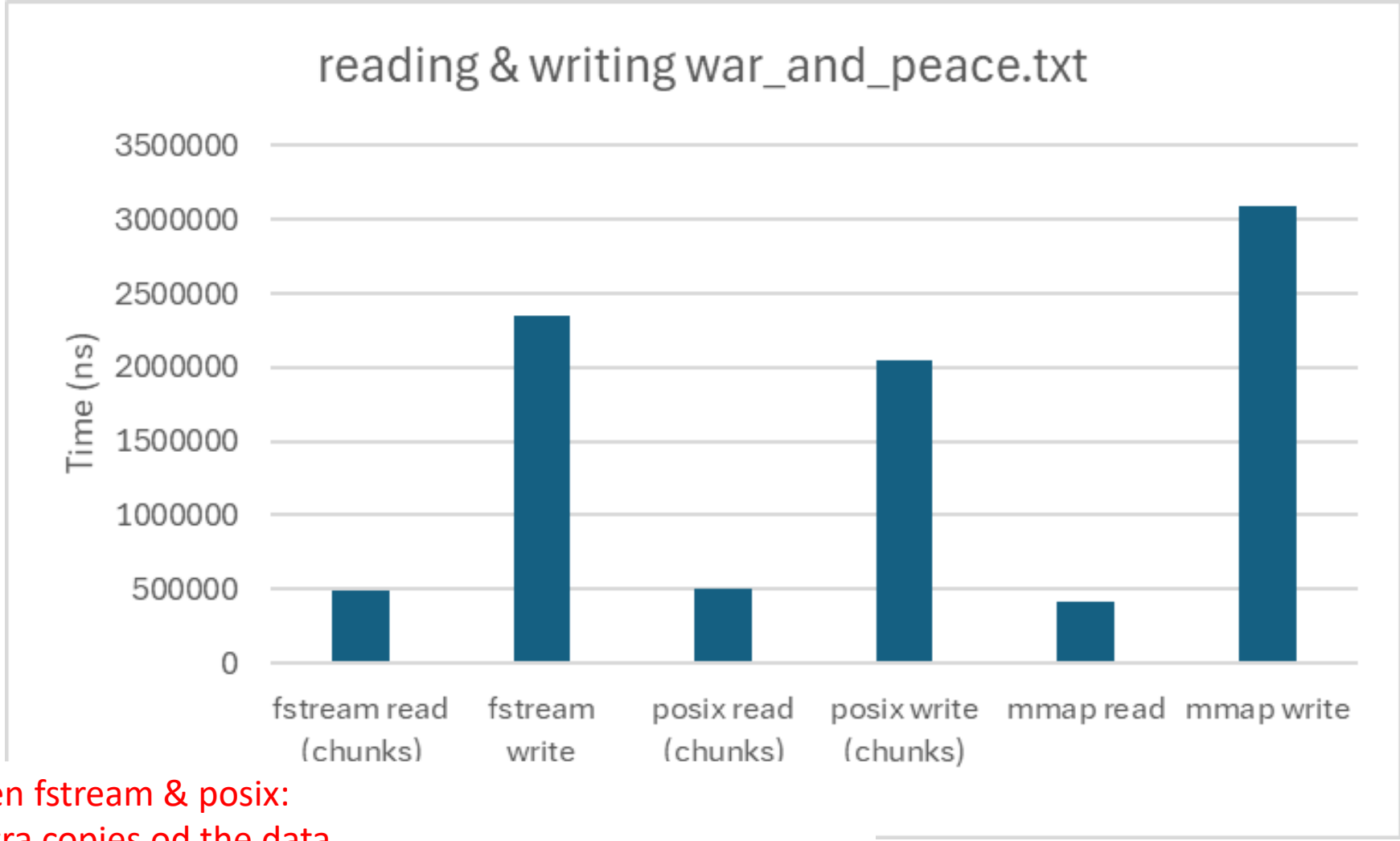
After removing char by char posix



Experiment Results

This is just war_and_peace
difference can amplify as we have more data to handle
can also a vary a lot from machine to machine

After removing
char by char posix
and
char by char stream



Difference between fstream & posix:
fstream makes extra copies od the data.
Copying data is a very fast thing to do though, so difference is not as big.

Optional Aside: what is mmap?

- mmap() is a posix system call that directly maps memory to a file. (among other things)
  - Reading/writing to memory will read/write to the file
  - Once the mapping is setup, no need to go through the OS or make extra copies of the data, just access it directly.
  - Can take some time to setup the mapping initially.
- Not enough time to talk about it in detail now.
  Take OS or ask how it works after class (if you want to know more)

Raise Hands

- What gets printed here?

```
int main() {
  int* x = nullptr;

  // write to file, and clear the log when we open it
  fstream log("log.txt", ios_base::out | ios_base::trunc);

  log << "I'M GONNA DO IT, I'M GONNA DEREF NULLPTR\n";

  *x = 5;


  log << "I'm alive?" << endl;

  return EXIT_FAILURE;
}
```

endl

- endl is more than just “newline”. It also “flushes” the buffer
- Flush the buffer: "take everything we have accumulated things in the buffer and send it to the destination".
- Quick: Which is faster?

```
int main() {
  fstream log("log.txt");

  log << "I\n";
  log << "am\n";
  log << "ok" << endl;

  return EXIT_FAILURE;
}
```

```
int main() {
  fstream log("log.txt");

  log << "I" << endl;
  log << "am" << endl;
  log << "ok" << endl;

  return EXIT_FAILURE;
}
```

- A flush is another system call to write to the destination…

Raise Hands

- What gets printed here?

```
int main() {
  int* x = nullptr;



  cout << "I'M GONNA DO IT, I'M GONNA DEREF NULLPTR\n";


  *x = 5;


  cout << "I'm alive?" << endl;

  return EXIT_FAILURE;
}
```

- What gets printed here?

```
int main() {
  int* x = nullptr;



  cout << "I'M GONNA DO IT, I'M GONNA DEREF NULLPTR\n";


  *x = 5;


  cout << "I'm alive?" << endl;

  return EXIT_FAILURE;
}
```

Cout is line buffered: it will flush when on newline
cerr is unbuffered

Why NOT Buffer?

- Reliability – the buffer needs to be flushed
  - Loss of computer power = loss of data
  - "Completion" of a write (*i.e.* return from `fwrite()`) does not mean the data has actually been written

- Performance – buffering takes time
  - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)

- When is buffering faster? Slower?

Many small writes
Or only writing a little

Large writes

Lecture Outline

- Memory Hierarchy Overview
- Buffering
- Memory Locality & Caching

Raise Hands

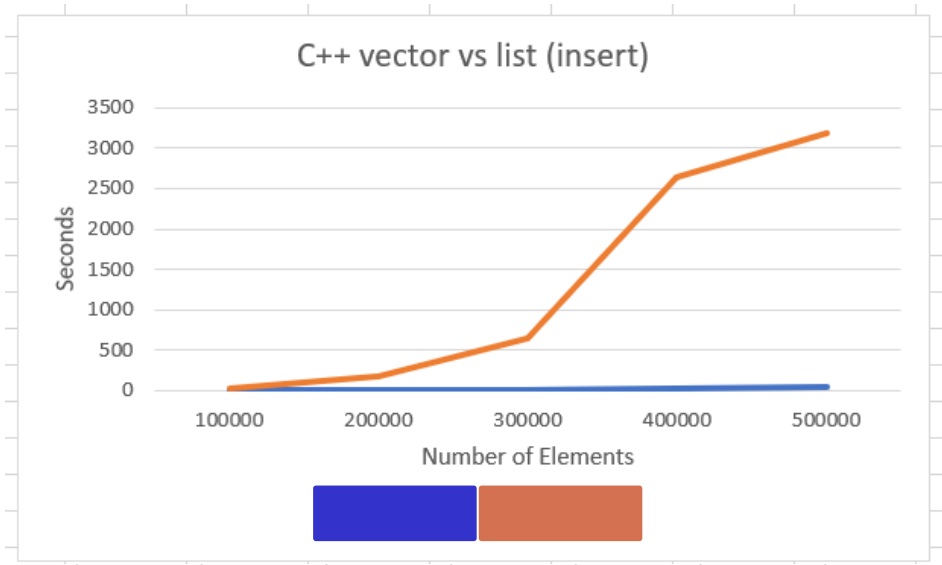
- Data Structures Review: I want to randomly generate a sequence of sorted numbers. To do this, we generate a random number and insert the number so that it remains sorted. Would a std::list (LinkedList) or a std::vector (ArrayList) work better?
  - What if we need to use Linear search?

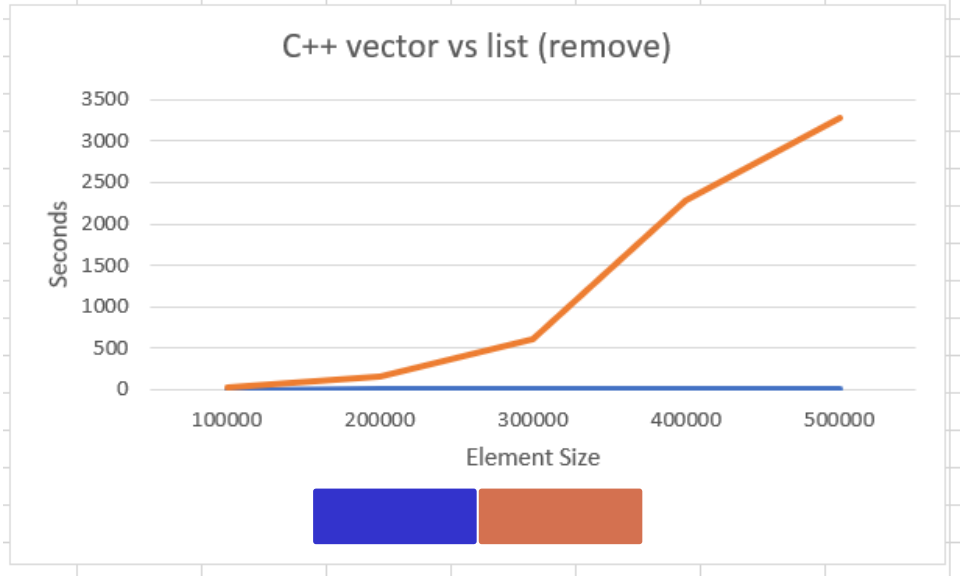
e.g. if I have sequence [5, 9, 23] and I randomly generate 12, I will insert 12 between 9 and 23

- Part 2: Let's say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a std::list (LinkedList) or a std::vector (ArrayList)?
  - What if we need to use Linear search?

Answer:

- I ran this in C++ on this laptop:
- Terminology
  - Vector == ArrayList
  - List == LinkedList
- On Element size from 100,000 -> 500,000

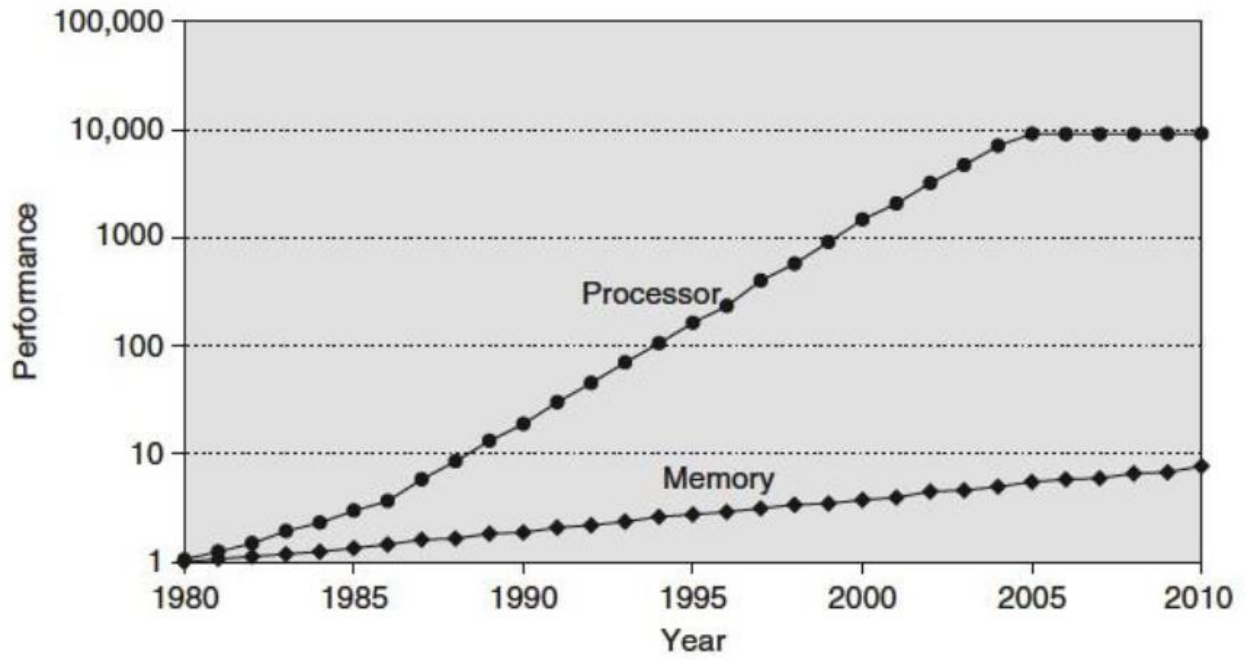




Data Access Time

- 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

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**
  - **It is important that data is quick to access to get better CPU utilization**

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

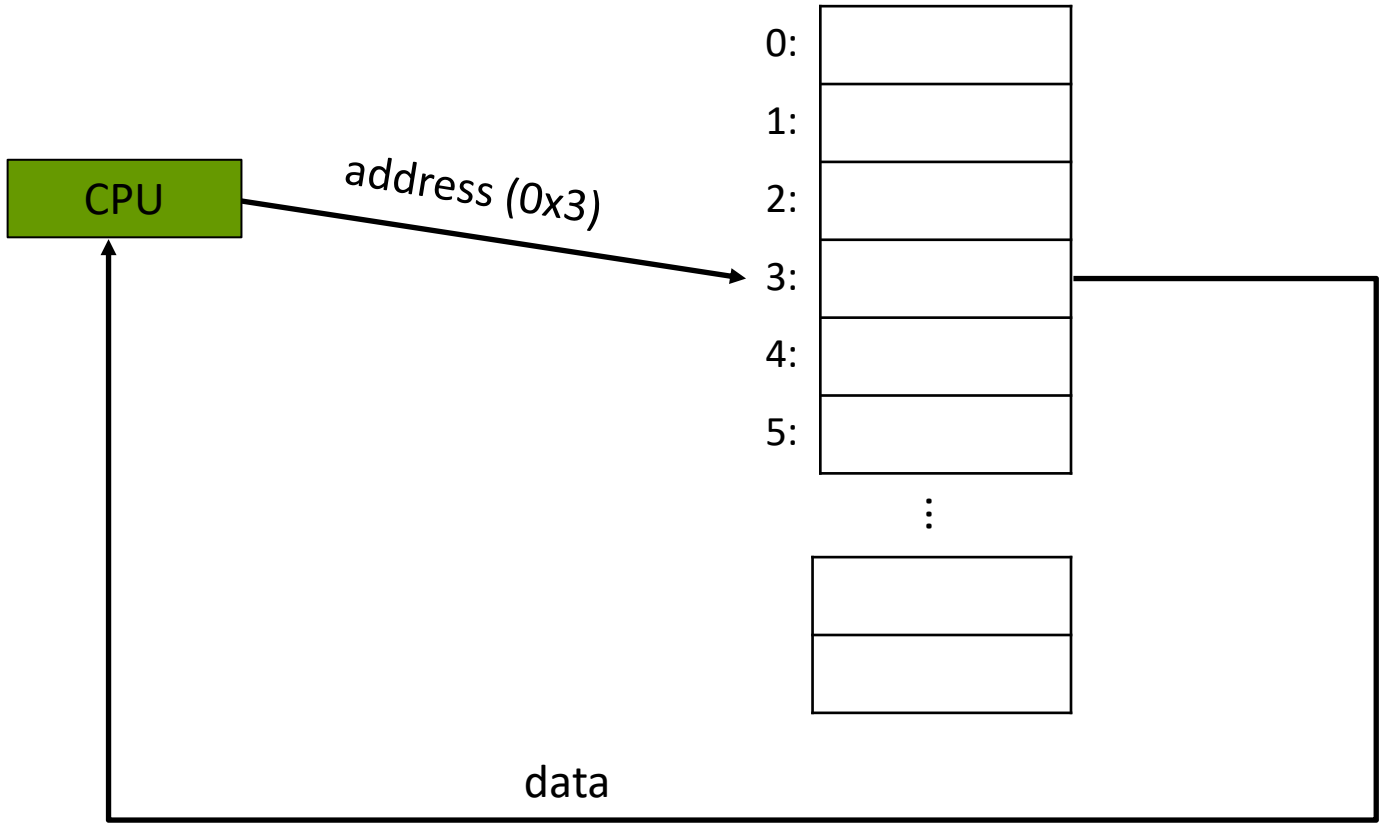
- (Generally) as we go further from the CPU, storage space goes up, but access times increase

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. (Instead of going to disk, we can go to physical memory which is quicker to access)

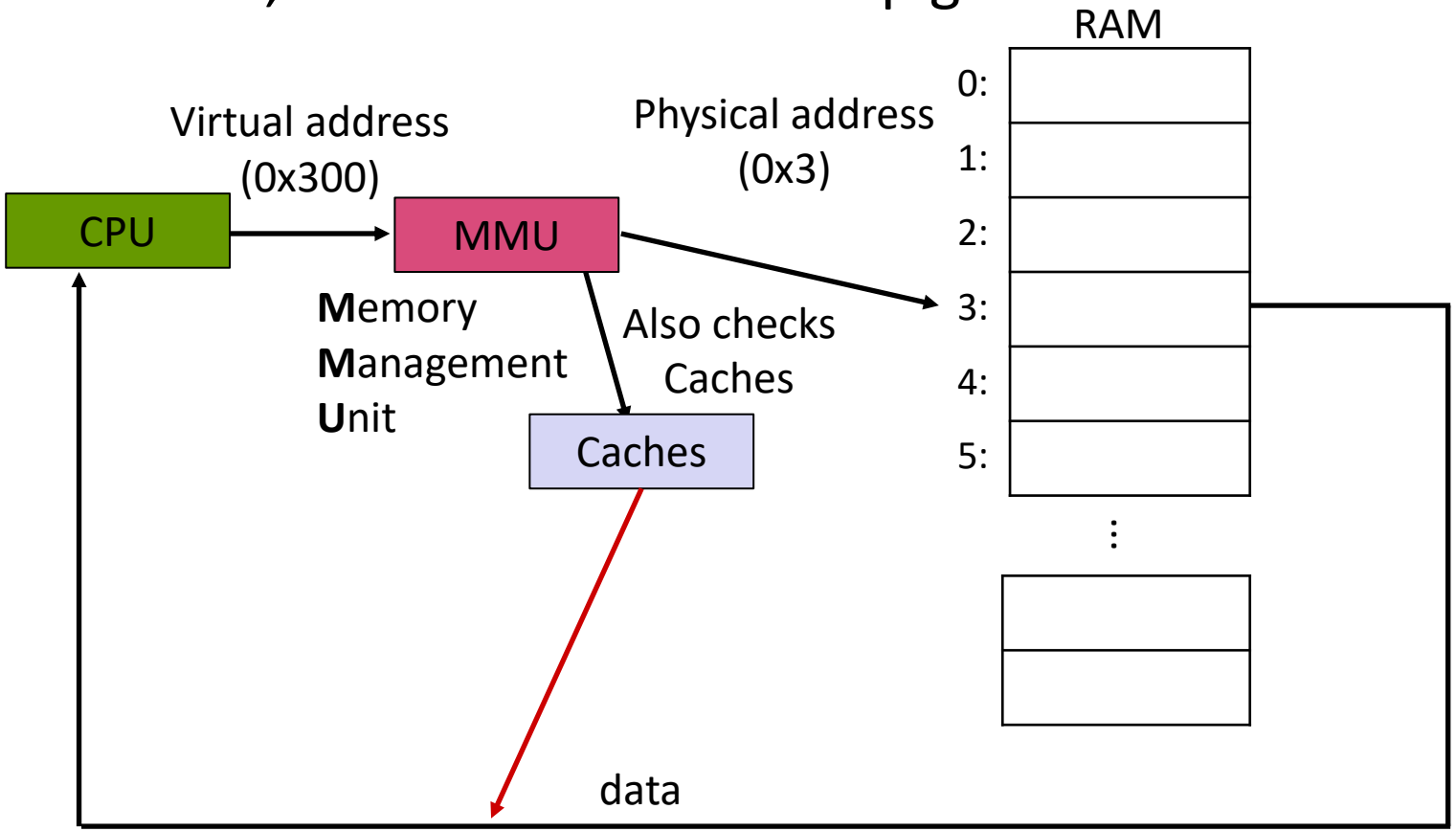
Memory (as we know it now)

- The CPU directly uses an address to access a location in memory



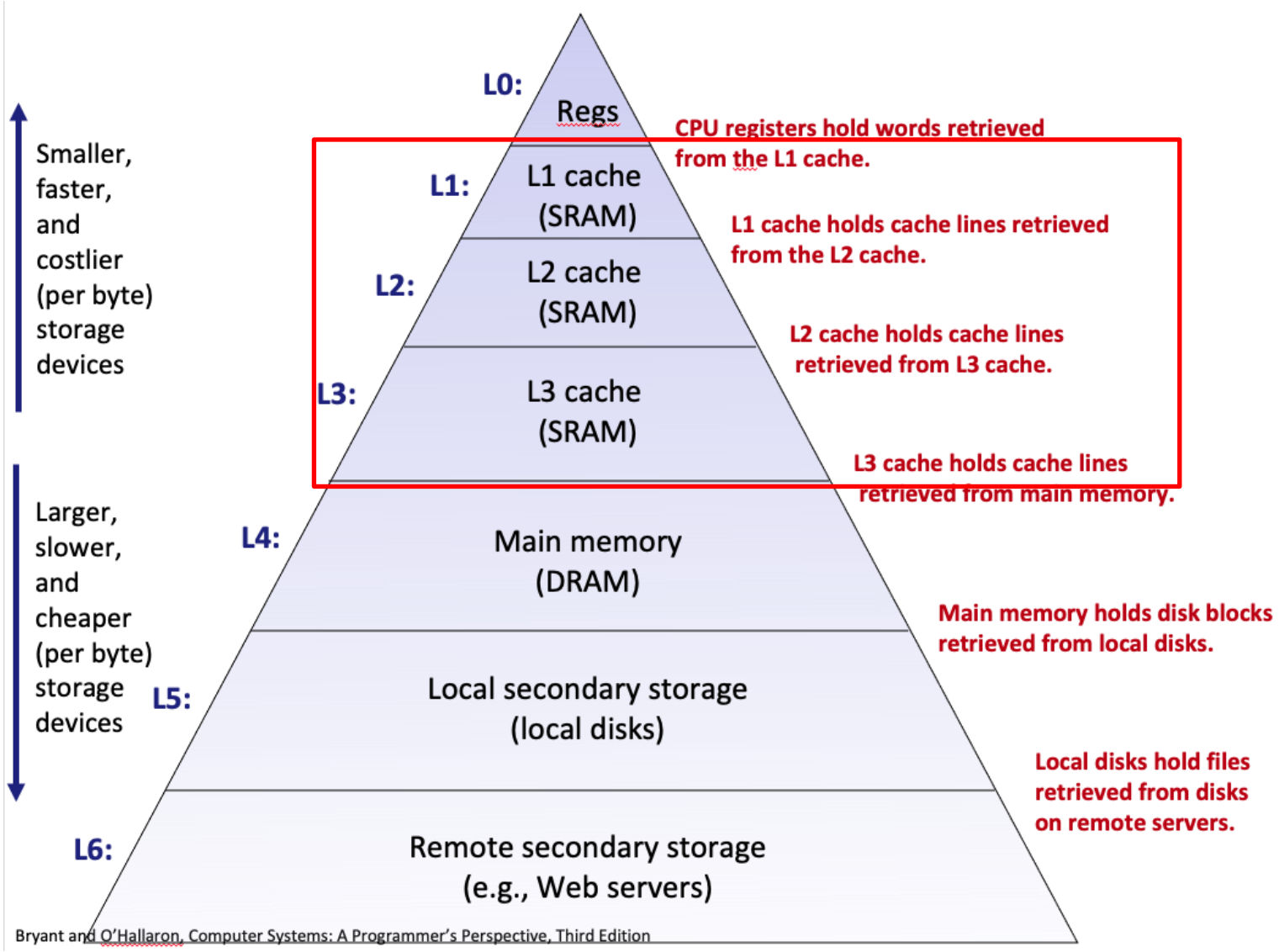
Virtual Address Translation

- Programs don't know about many of things going on under the hood with memory. they send an address to the MMU, and the MMU will help get the data



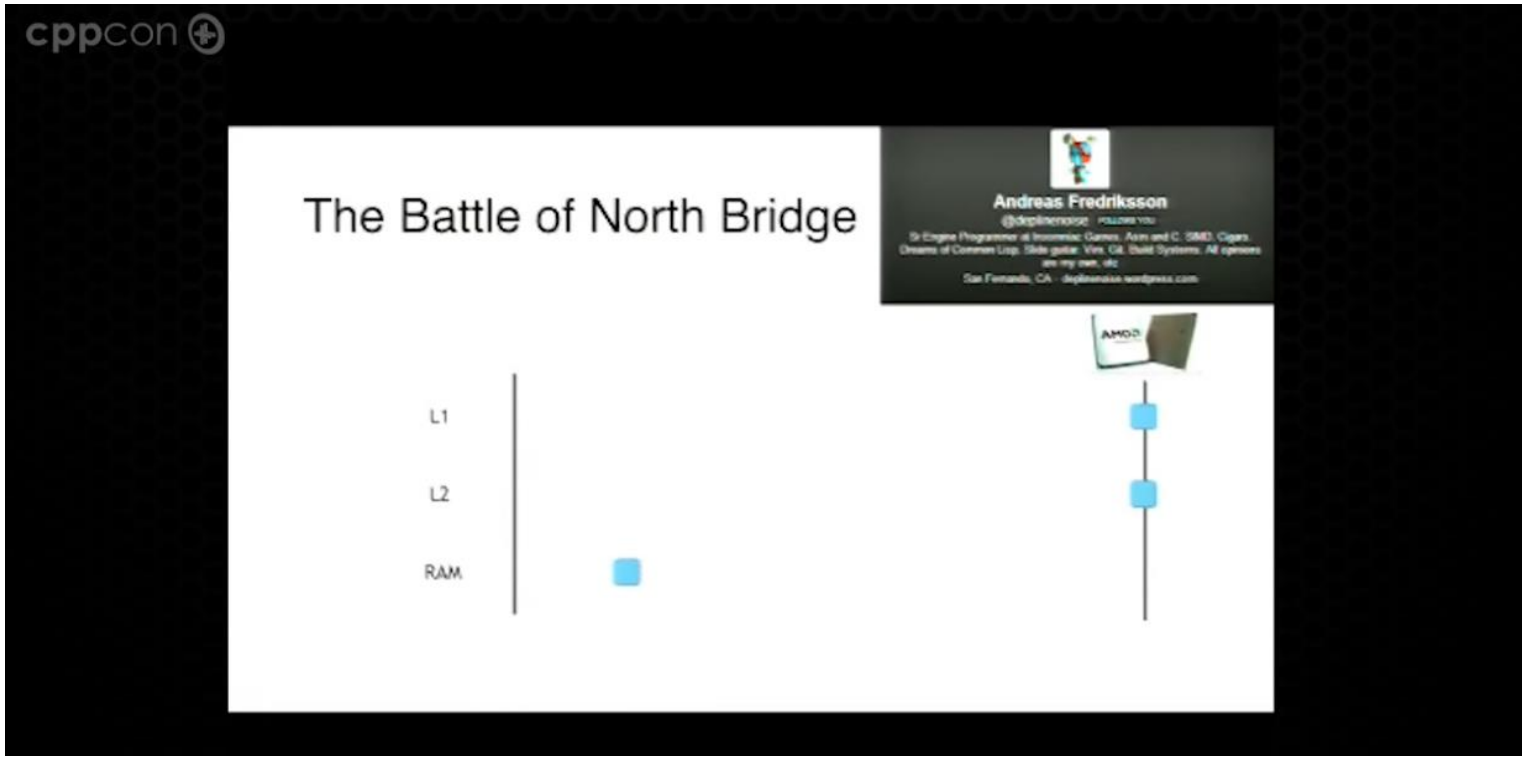
Memory Hierarchy

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



Cache vs Memory Relative Speed

- Animation from Mike Acton's Cppcon 2014 talk on "data oriented design".
  - https://youtu.be/rX0ItVEVjHc?si=MRTeW3taRmRU1fpB&t=1830
  - Animation starts at 30:30, ends 31:07 ish

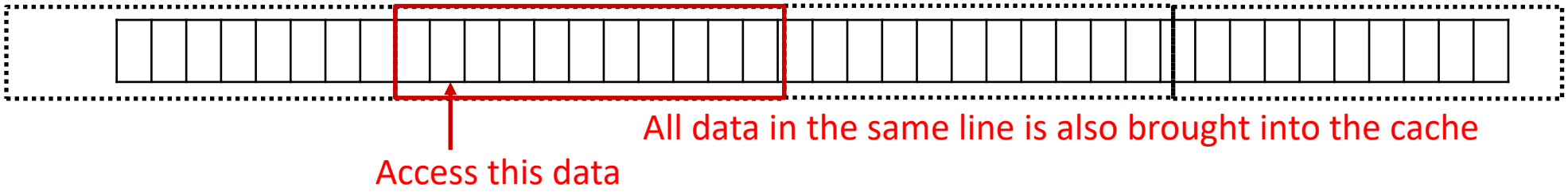


Cache Performance

- Accessing data in the cache allows for much better utilization of the CPU
- Accessing data **not** in the cache can cause a bottleneck: CPU would have to wait for data to come from memory.
- How is data loaded into a Cache?

Cache Lines

- Imagine memory as a big array of data:



- We can split memory into contiguous non-overlapping 64-byte “lines” or “blocks”(64 bytes on most architectures)

- When we access data at an address, we bring the whole cache line (cache block) into the L1 Cache
  - Data next to address access is thus also brought into the cache!

What about other languages?

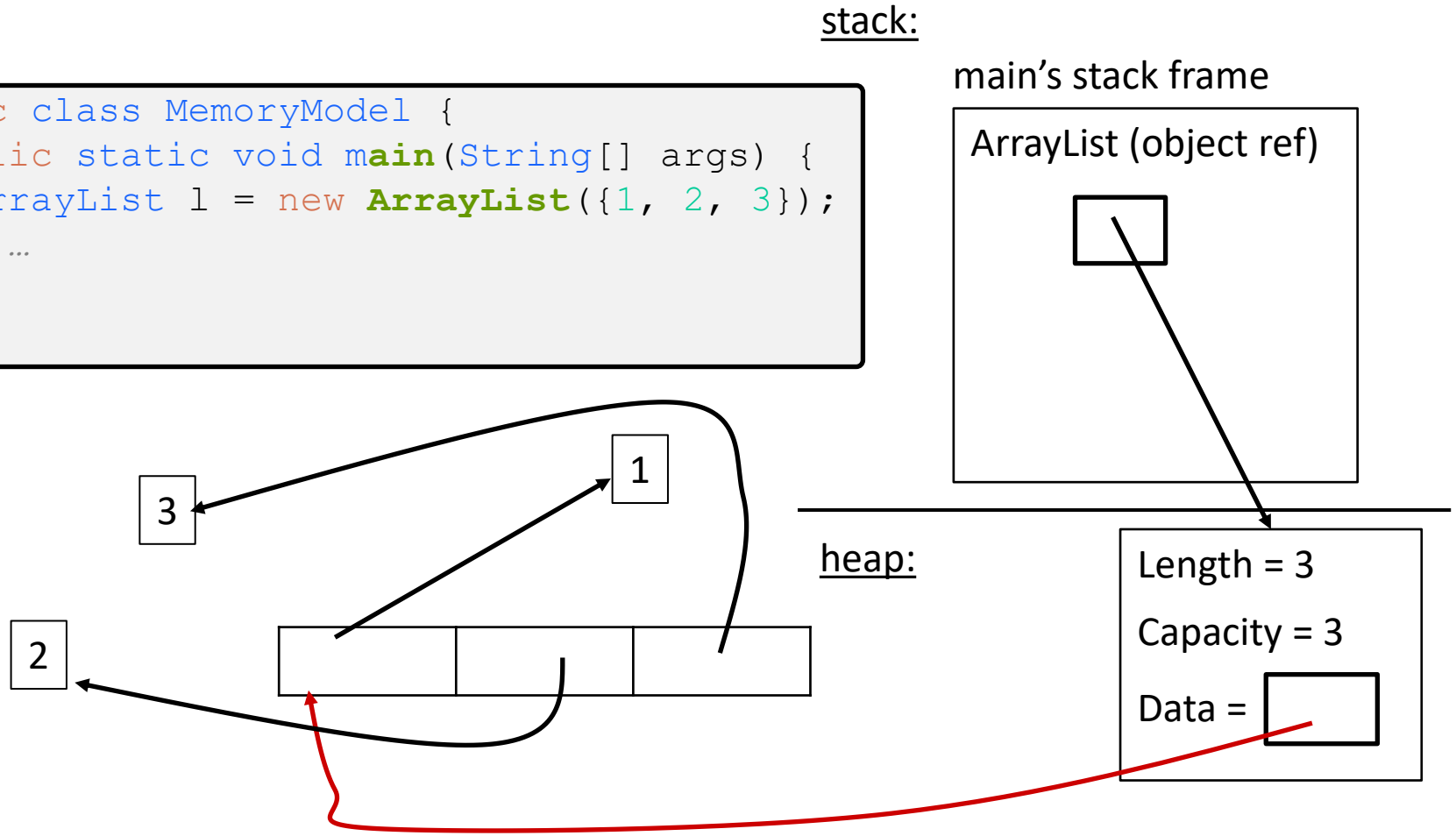
- In C++ (and C, Rust, Zig …) when you declare an object, you have an instance of that object. If you declare it as a local variable, it **exists on the stack**

- In most other languages (including Java, Python, etc.), the memory model is slightly different. Instead, **all object variables are object references, that refer to an object on the heap**

ArrayList in Java Memory Model

- In Java, the memory model is slightly different. all object variables are object references, that refer to an object on the heap

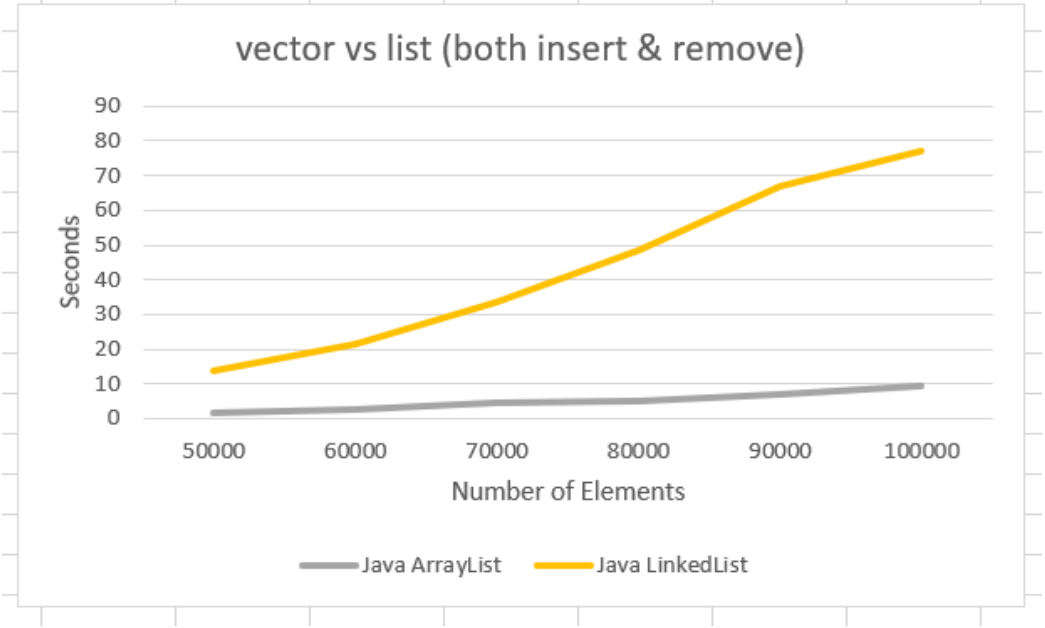
```
public class MemoryModel {
  public static void main(String[] args) {
    ArrayList l = new ArrayList({1, 2, 3});
    // …
  }
}
```



Does Caching apply to Java?

- I believe so, yes. Doing the same experiment in java got:

- Note: did this on smaller number of elements.
  50,000 -> 100,000



Raise Hands

- Let's say I had a matrix (rectangular two-dimensional array) of integers, and I want the sum of all integers in it

- Would it be faster to traverse the matrix row-wise or column-wise?
  - row-wise (access all elements of the first row, then second)
  - column:-wise (access all elements of the first column, …)

1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

Raise Hands

- Let's say I had a matrix (rectangular two-dimensional array) of integers, and I want the sum of all integers in it

- Would it be faster to traverse the matrix row-wise or column-wise?
  - row-wise (access all elements of the first row, then second)
  - column:-wise (access all elements of the first column, …)

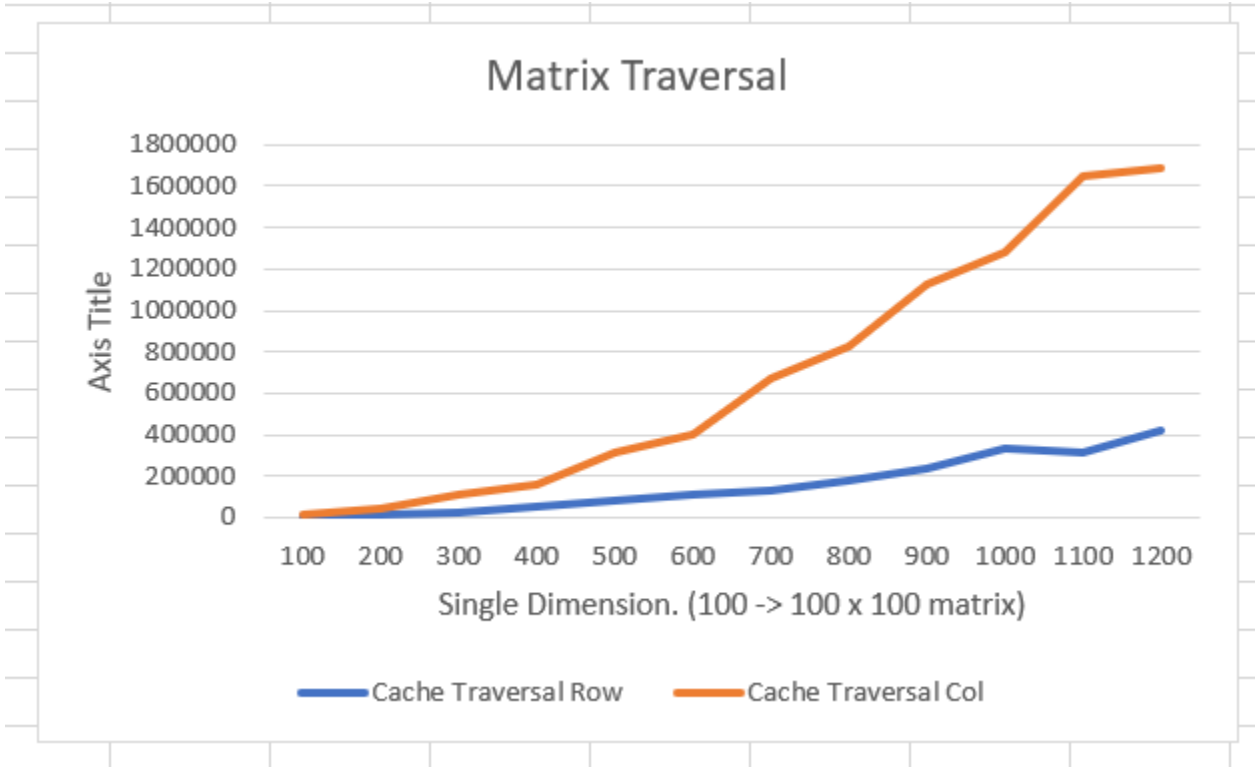
1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

Hint: Memory Representation in C & C++

1	5	8	10	11	2	6	9	14	12	3	7	0	15	13	4

Experiment Results

- I ran this in C:



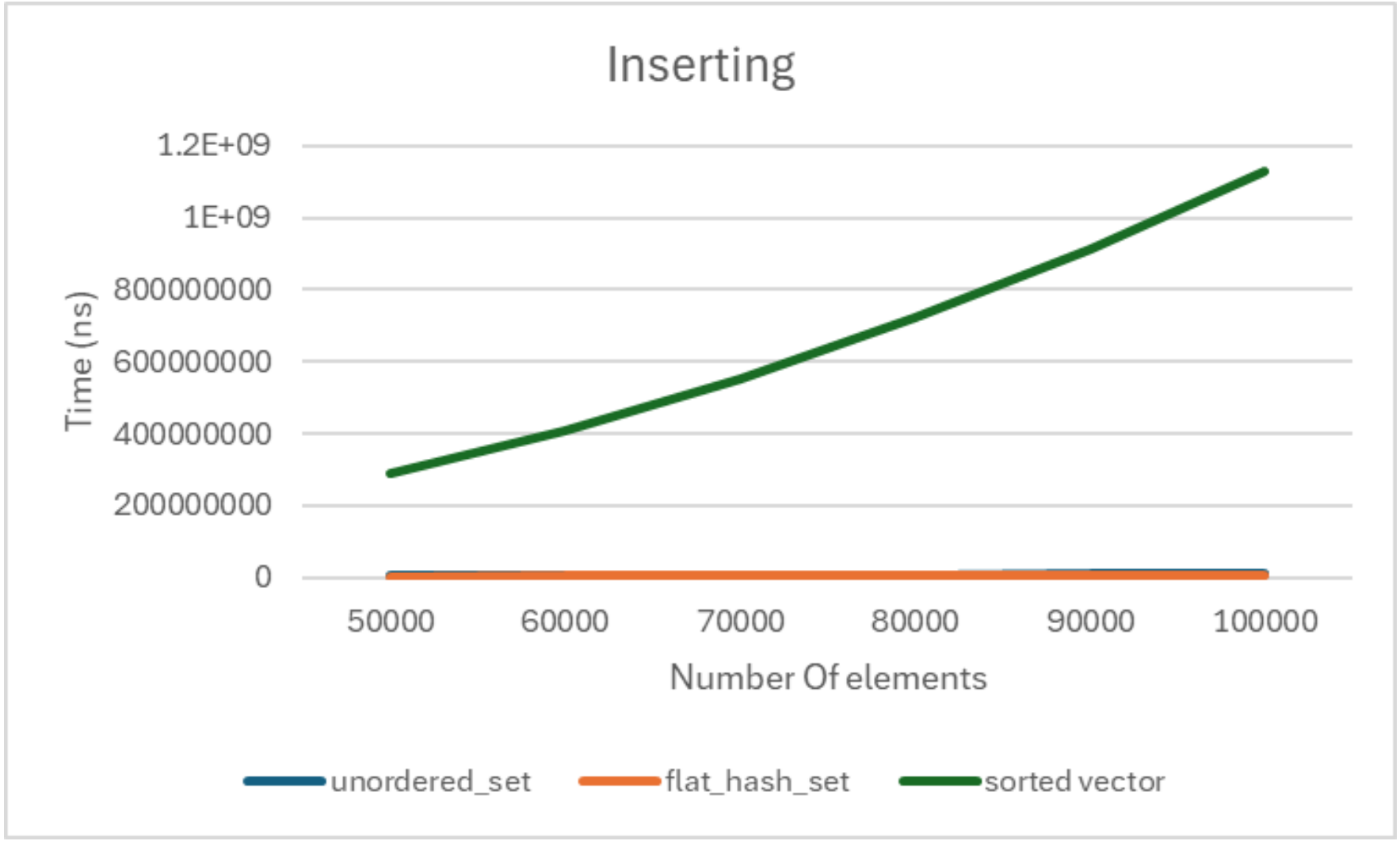
- Row traversal is better since it means you can take advantage of the cache

Raise Hands

- I randomly generate 1,000,000 doubles that I want to keep unique. I insert them into a container to make sure there are no duplicates.
  - Which container do you think I should use?

- Part 2: Let's say we take the container from part 1, I then need to iterate over all of the values and set them to their inverse square root (x = 1.0 / sqrt(x))
  - Which container would work best?

Results


Inserting
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
Time (ns)
50000
60000
70000
80000
90000
100000
Number Of elements
unordered_set
flat_hash_set
sorted vector

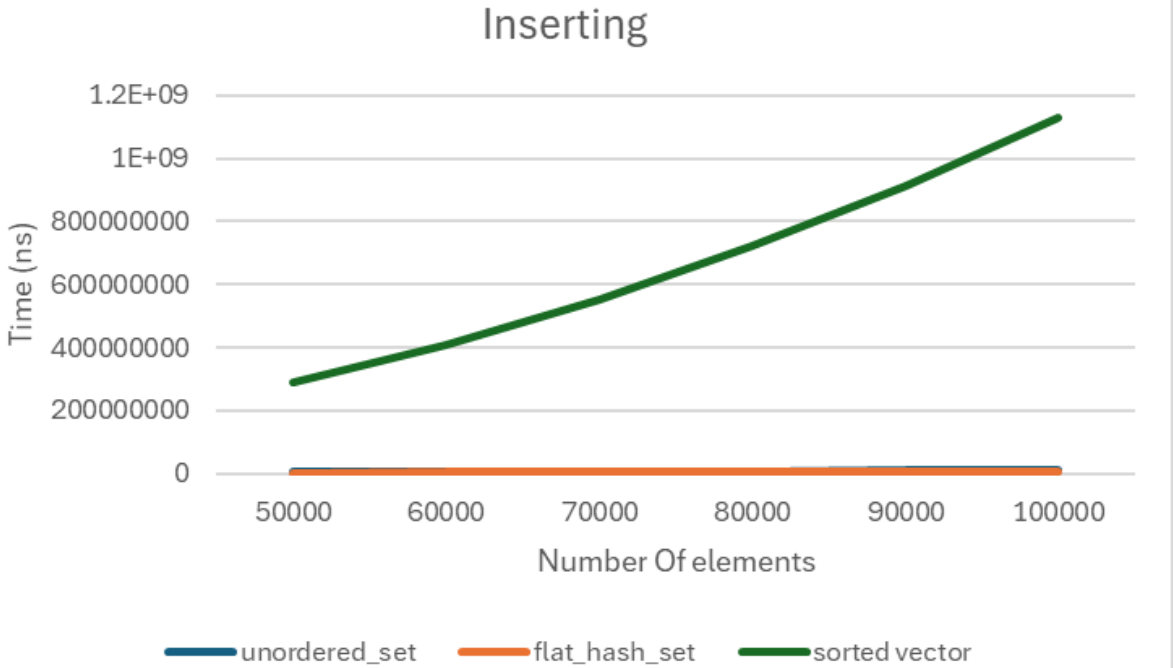
Aside: Swiss Hash Table (abesil flat_hash_map/set)

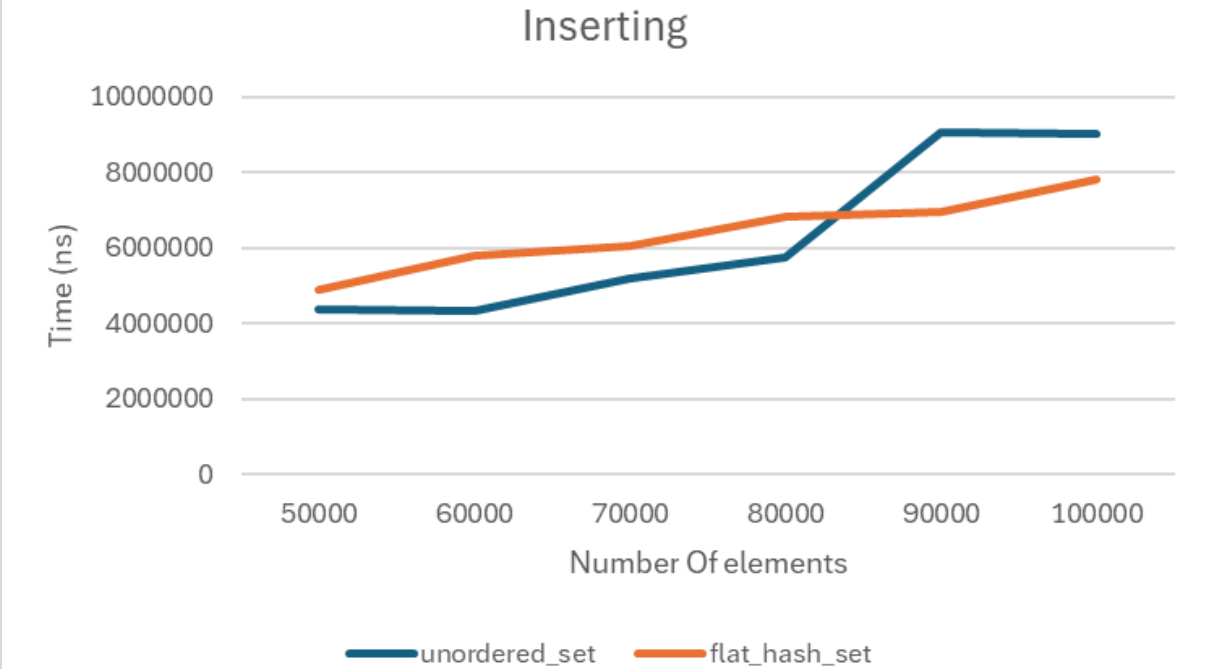
- Arrays are great, but not everything.
  A great talk that goes through many topics related to performance, memory and caching.

- How to take advantage of systems knowledge to make better data structures

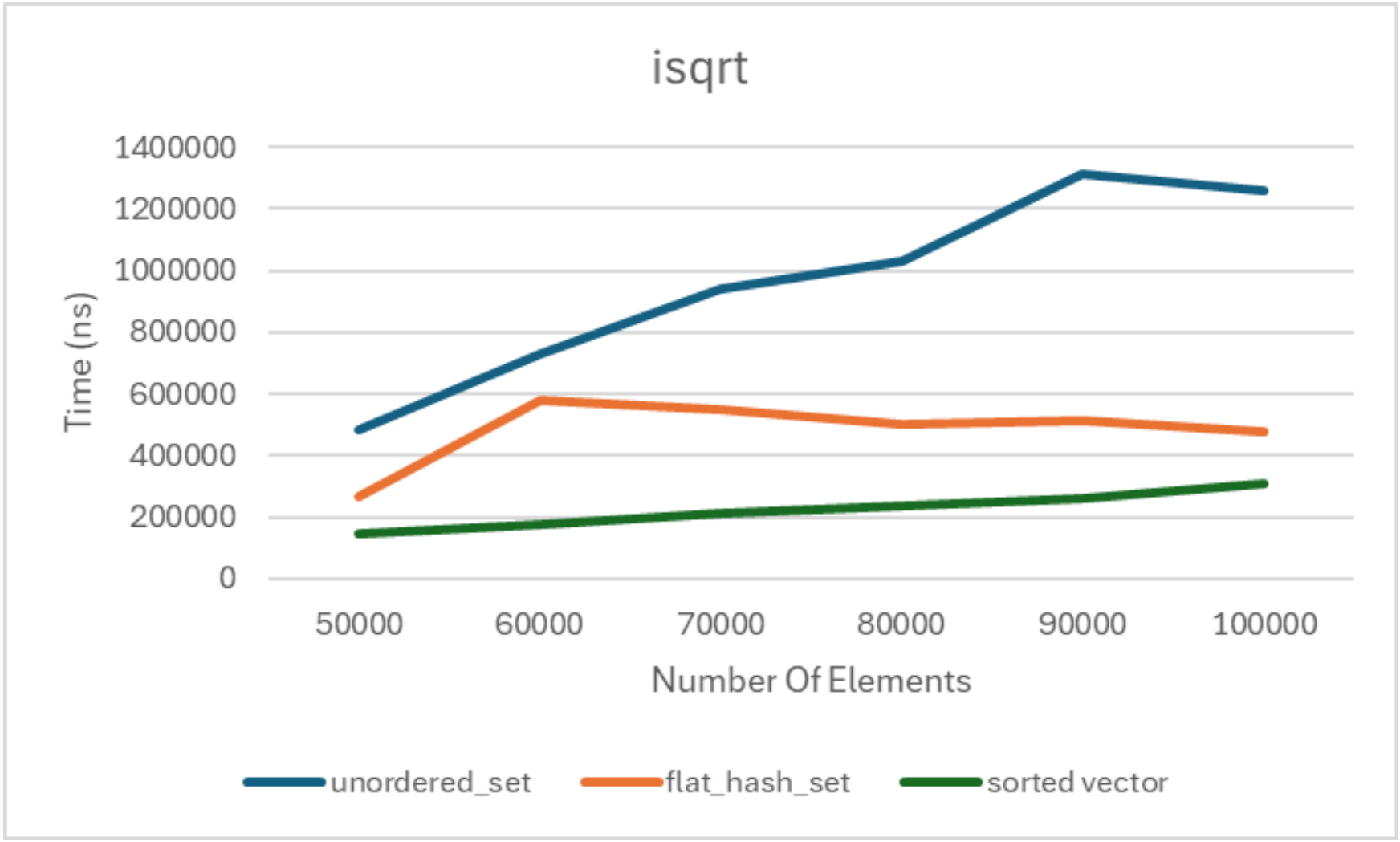
- **CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"**
  - https://www.youtube.com/watch?v=ncHmEUmJZf4

Results


Inserting
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
Time (ns)
50000
60000
70000
80000
90000
100000
Number Of elements
unordered_set
flat_hash_set
sorted vector


Inserting
10000000
8000000
6000000
4000000
2000000
0
Time (ns)
50000
60000
70000
80000
90000
100000
Number Of elements
unordered_set
flat_hash_set

Results


isqrt
1400000
1200000
1000000
800000
600000
400000
200000
0
Time (ns)
50000
60000
70000
80000
90000
100000
Number Of Elements
unordered_set
flat_hash_set
sorted vector

Choosing a Data Structure & Such

- Choosing a data structure /algorithm is not just thinking about minimizing CPU computation (Big O analysis)

- Keeping in mind:
  - Hardware Utilization & Data Locality
    - Caching memory
    - Mindful of I/O operations
  - Memory allocations
  - Other things we haven't gotten to yet

- This systems knowledge applies beyond this course. (Example: training LLMs)

That's all for now!

- Next time:
  - Performance wrapup
  - Git!

- Hopefully you are doing well ☺