

Caches, Memory Allocation, `std::move`

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk

Ashwin Alaparthi

Lobi Zhao

Angie Cao

Austin Lin

Pearl Liu

Aniket Ghorpade

Hassan Rizwan

Perrie Quek



pollev.com/tqm

❖ How are you?

Administrivia

- ❖ pipe_shell (HW05)
 - Demo'd in recitation last week
 - Like retry shell, but piping instead of retrying
 - Extended autograder opening to Sunday this week.

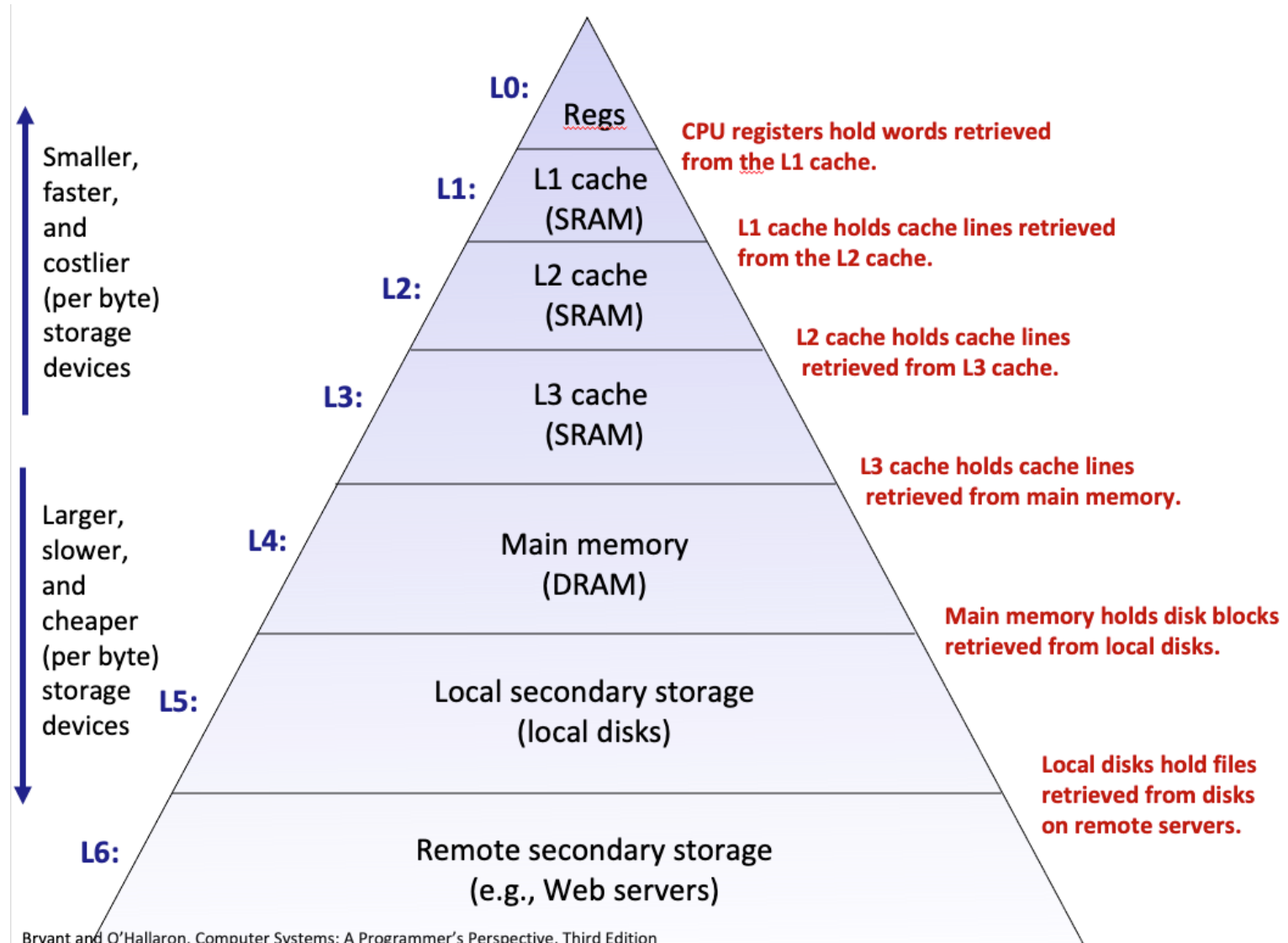
- ❖ Midterm next week 😊
 - Midterm review in recitation this week
 - Midterm review in lecture on Tuesday
 - Policies posted soon

- ❖ Travis' Office Hours on Friday moved to Sunday
 - Travis at a conference 😊

Lecture Outline

- ❖ **Locality**
- ❖ Caches
- ❖ Memory Allocation & fragmentation
- ❖ Being aware of memory allocation in C++
- ❖ `std::move` **Did not get to `std::move` in lecture**

Memory Hierarchy



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

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- ❖ Numbers are out of date but the relative orders of magnitude are about the same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

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns



 **Poll Everywhere**pollev.com/tqm

- ❖ Suppose we have a large vector of strings that we want to print to a file (an ofstream). Which code prints the vector faster?
 - (You can assume this code compiles)

```
void print_vec(ofstream& to_print, const vector<string>& words) {  
    for (auto& word : words) {  
        to_print << word << endl;  
    }  
}
```

```
void print_vec(ofstream& to_print, vector<string>& words) {  
    for (size_t i = 0; i < words.size(); i++) {  
        string& word = words[i];  
        to_print << word;  
        to_print << "\n";  
    }  
}
```


How to flush/modify an iostream buffer

❖ For C++ iostream stdio:

- `std::flush`

- Flushes the stream to the OS/filesystem

- `std::endl`

- Flushes the stream to the OS/filesystem and prints a new line

- `std::pubsetbuf`

- Can set the stream to be unbuffered or a specified buffer

Lecture Outline

- ❖ Locality
- ❖ **Caches**
- ❖ Memory Allocation & fragmentation
- ❖ Being aware of memory allocation in C++
- ❖ `std::move`

 **Poll Everywhere**pollev.com/tqm

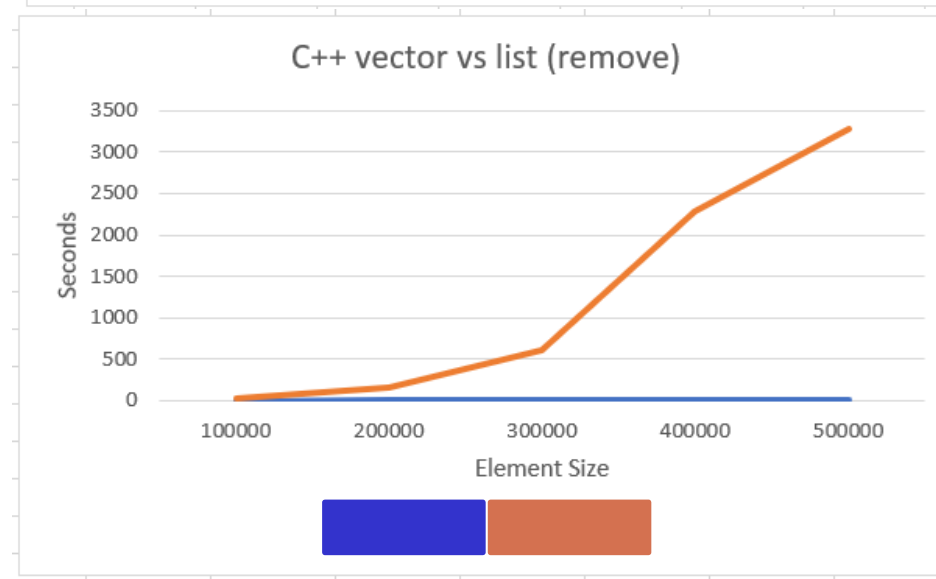
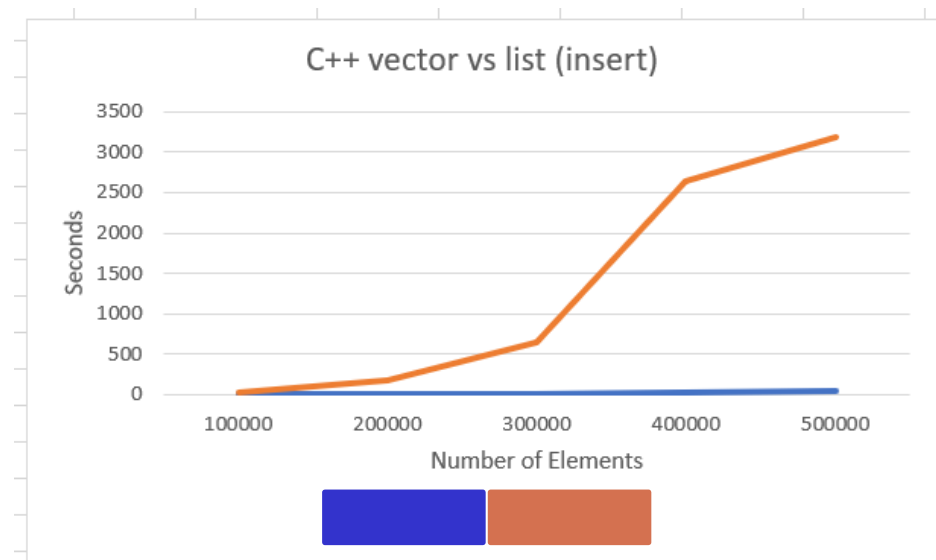
- ❖ Data Structures Review: I want to randomly generate a sequence of sorted numbers. To do this, we generate a random number and insert the number so that it remains sorted. Would a LinkedList or an ArrayList work better?

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 LinkedList or an ArrayList?

Answer:

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

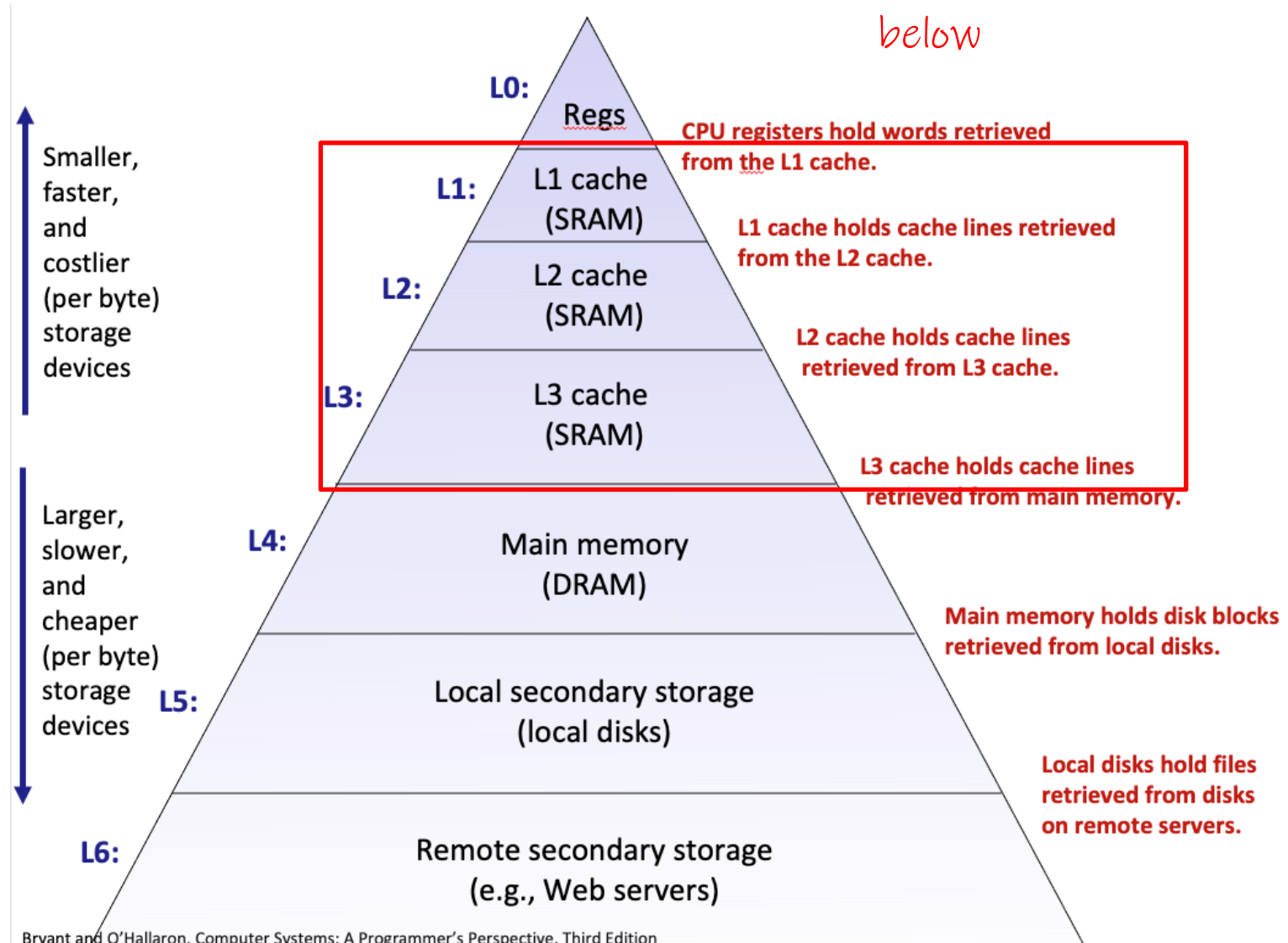


Data Access Time

- ❖ 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

Memory Hierarchy

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

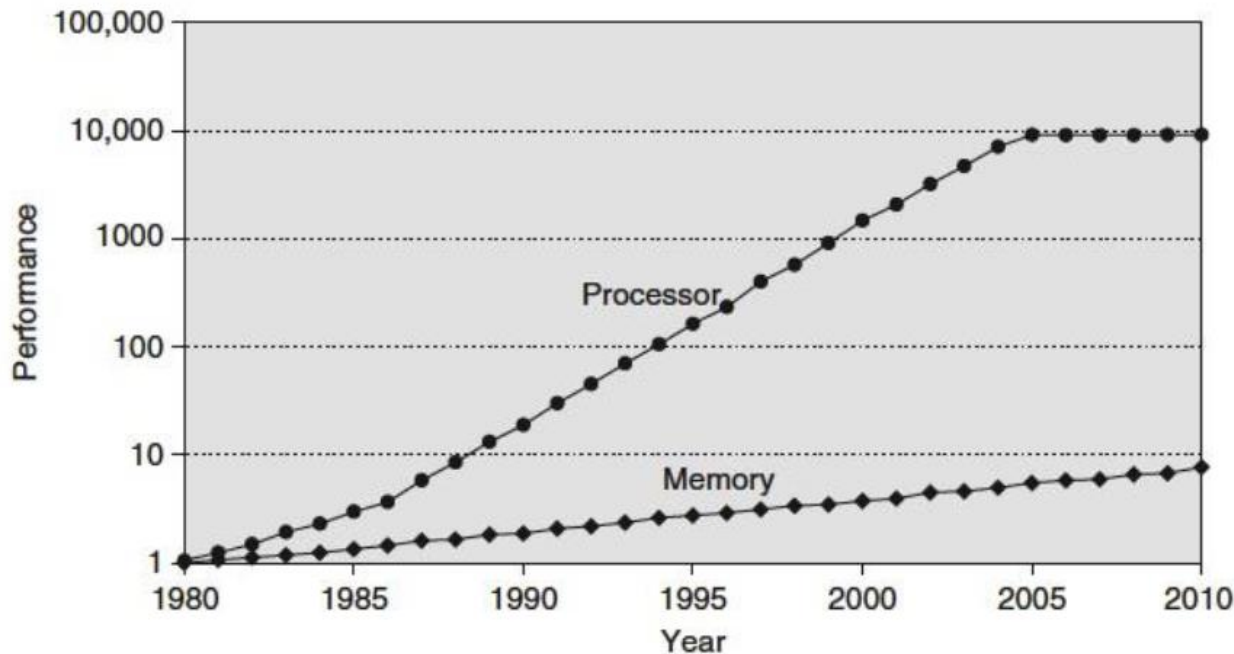


Memory Hierarchy so far

- ❖ 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

Processor Memory Gap



- ❖ Processor speed kept growing $\sim 55\%$ per year
- ❖ Time to access memory didn't grow as fast $\sim 7\%$ per year
- ❖ **Memory access would create a bottleneck on performance**
 - **It is important that data is quick to access to get better CPU utilization**

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)

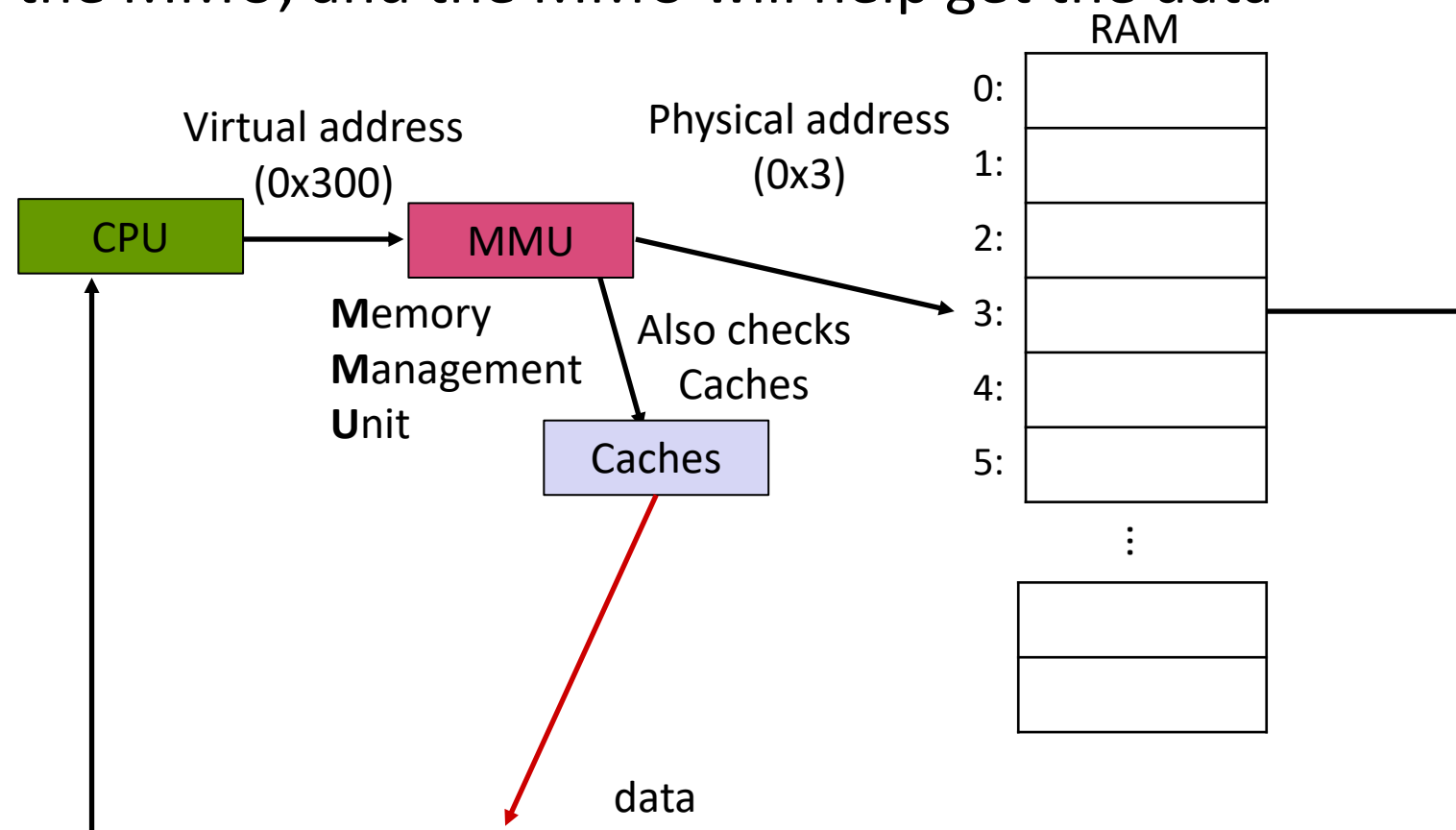
Memory (as we know it now)

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



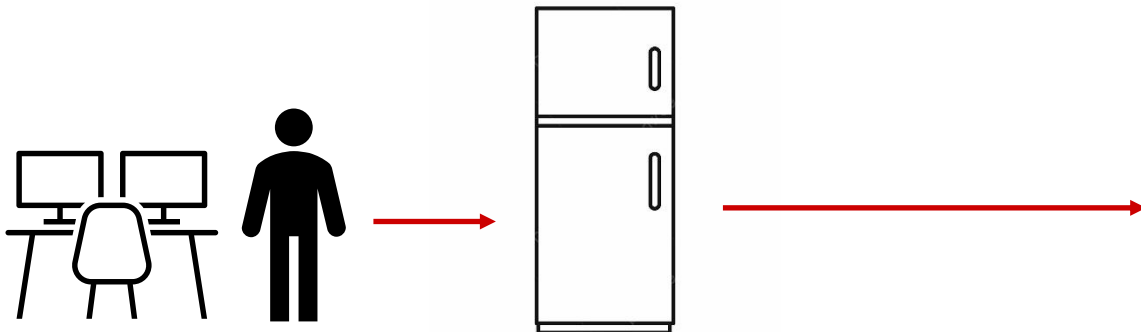
Virtual Address Translation

- ❖ 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



Cache 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)



Cache vs Memory Relative Speed

- ❖ Animation from Mike Acton's Cppcon 2014 talk on "data oriented design".
 - <https://youtu.be/rX0ltVEVjHc?si=MRTeW3taRmRU1fpB&t=1830>
 - Animation starts at 30:30, ends 31:07 ish

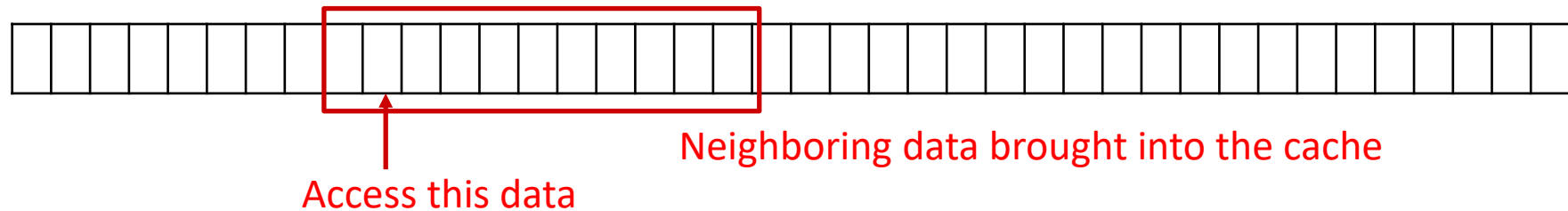


Cache Performance

- ❖ 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 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!

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 to help with cache management

Cache Replacement Policy

- ❖ Caches are small and can only hold so many cache lines inside it.
- ❖ When we access data not in the cache, and the cache is full, we must evict an existing entry.
- ❖ When we access a line, we can do a quick calculation on the address to determine which entry in the cache we can store it in. (Depending on architecture, 1 to 12 possible slots in the cache)
 - Cache's typically follow an LRU (Least Recently Used) on the entries a line can be stored in

LRU (Least Recently Used)

- ❖ If a cache line is used recently, it is likely to be used again in the near future
- ❖ Use past knowledge to predict the future
- ❖ Replace the cache line that has had the longest time since it was last used

Back to the Poll Questions

- ❖ 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 LinkedList or an ArrayList work better?

- ❖ 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 LinkedList or an ArrayList?

Data Structure Memory Layout

- ❖ Important to understanding the poll questions, we understand the memory layout of these data structures

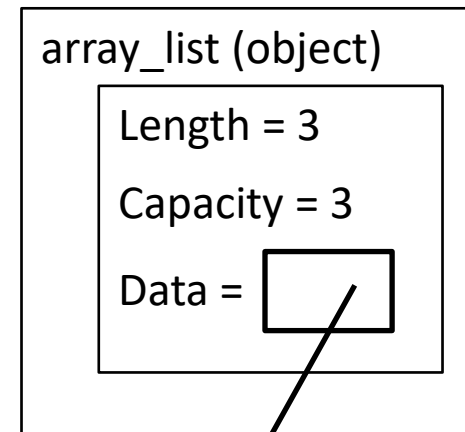
- ❖ ArrayList In C++:

```
int main() {  
    vector<int> array_list {1, 2, 3};  
    // ...  
}
```

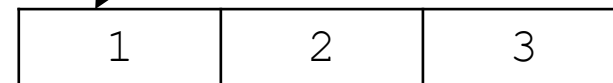
Elements are next to each other in memory 😊

stack:

main's stack frame



heap:



Data Structure Memory Layout

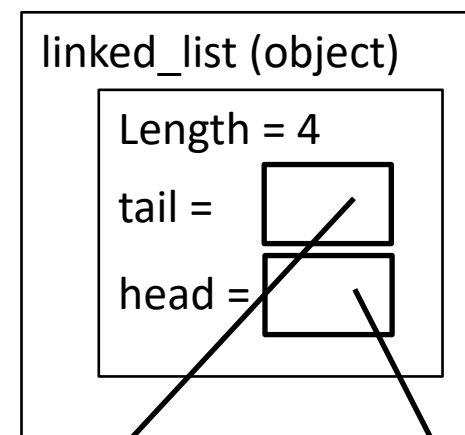
- ❖ Important to understanding the poll questions, we understand the memory layout of these data structures

- ❖ LinkedList In C++:

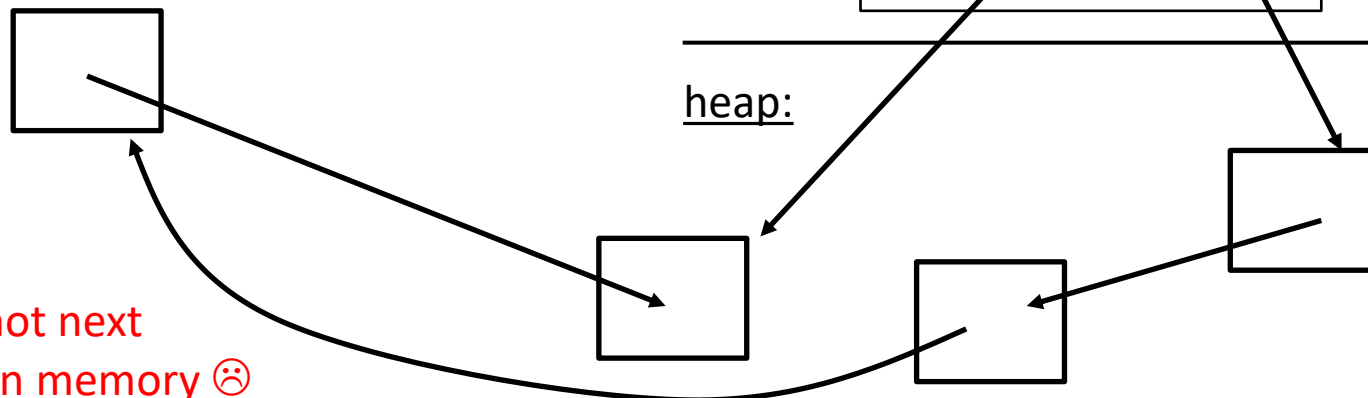
```
int main() {  
    list<int> linked_list {1, 2, 3, 4};  
    // ...  
}
```

stack:

main's stack frame



heap:



Elements are not next
to each other in memory ☹️

Poll Question: Explanation

- ❖ Vector wins in-part for a few reasons:
 - Less memory allocations
 - Integers are next to each other in memory, so they benefit from spatial complexity (and temporal complexity from being iterated through in order)
- ❖ Does this mean you should always use vectors?
 - No, there are still cases where you should use lists, but your default in C++, Rust, etc should be a vector
 - If you are doing something where performance matters, your best bet is to experiment try all options and analyze which is better.

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**

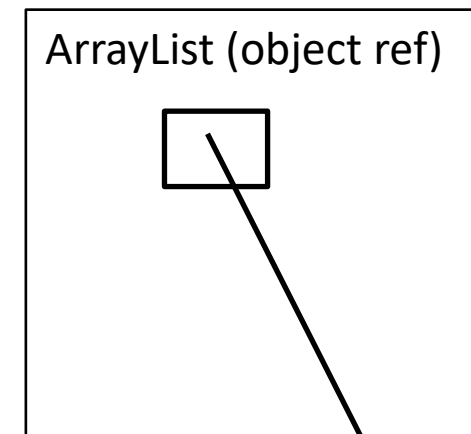
ArrayList in Java Memory Model

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

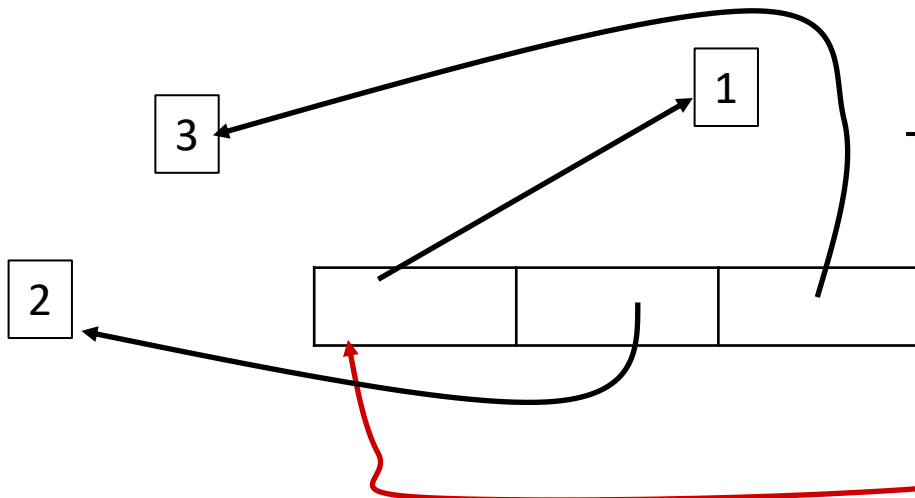
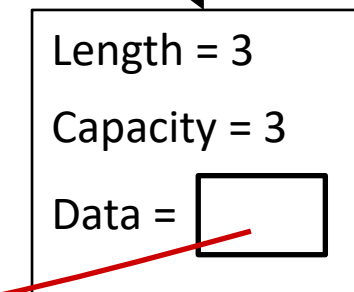
stack:

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

main's stack frame



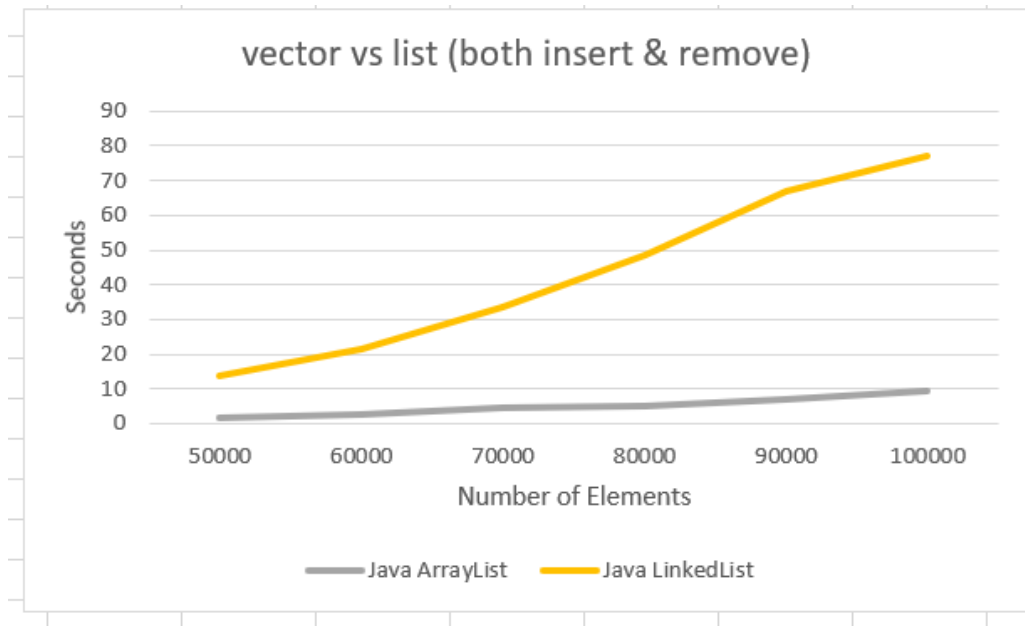
heap:



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



 **Poll Everywhere**pollev.com/tqm

- ❖ 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

 **Poll Everywhere**pollev.com/tqm

- ❖ 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, ...)

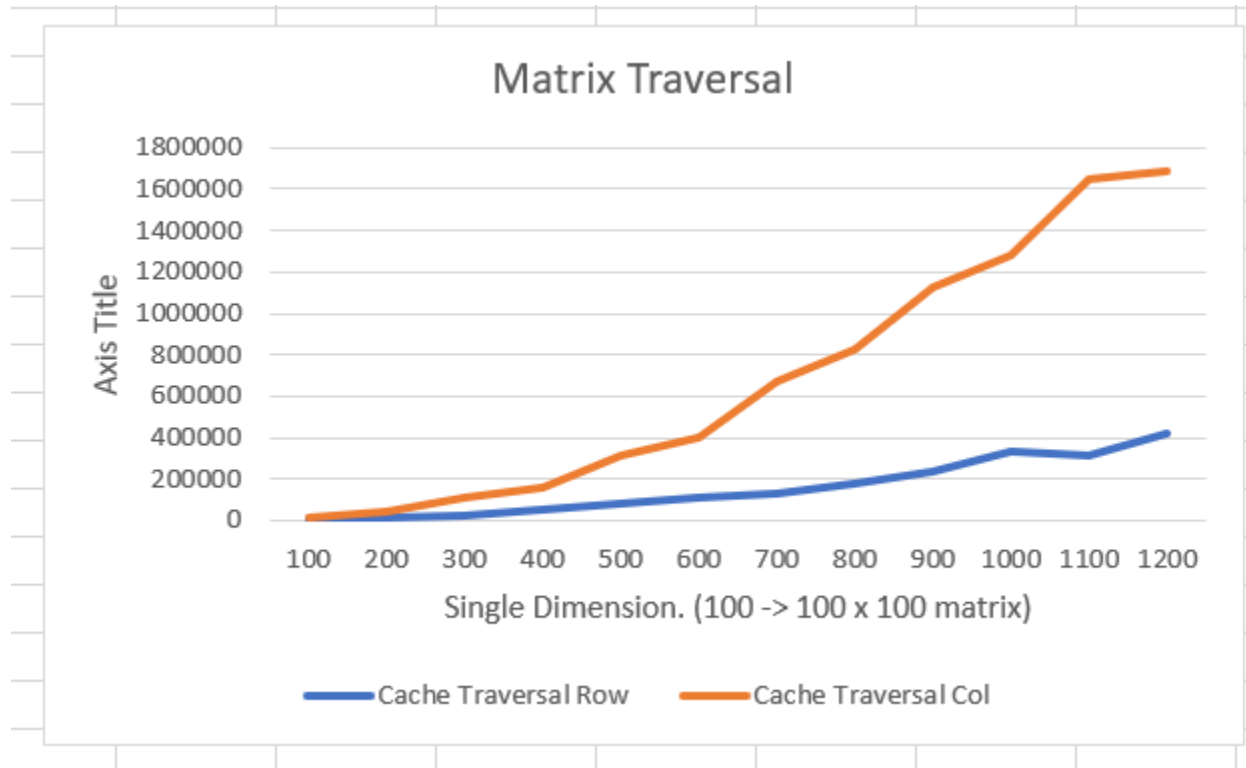
1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

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

Instruction Cache

- ❖ The CPU not only has to fetch data, but it also fetches instructions. There is a separate cache for this
 - which is why you may see something like L1I cache and L1D cache, for Instructions and Data respectively
- ❖ Consider the following three fake objects linked in inheritance

```
public class A {  
    public void compute () {  
        // ...  
    }  
}
```

```
public class B extends A {  
    public void compute () {  
        // ...  
    }  
}  
  
public class C extends A {  
    public void compute () {  
        // ...  
    }  
}
```

Instruction Cache

❖ Consider this code

```
public class ICacheExample {  
    public static void main(String[] args) {  
        ArrayList<A> l = new ArrayList<A>();  
        // ...  
        for (A item : l) {  
            item.compute();  
        }  
    }  
}
```

- ❖ When we call `item.compute` that could invoke `A`'s `compute`, `B`'s `compute` or `C`'s `compute`
- ❖ Constantly calling different functions, may not utilize instruction cache well

```
public class A {  
    public void compute() {  
        // ...  
    }  
}
```

```
public class B extends A {  
    public void compute() {  
        // ...  
    }  
}
```

```
public class C extends A {  
    public void compute() {  
        // ...  
    }  
}
```

Instruction Cache

- ❖ Consider this code new code: makes it so we always do
A.compute() -> B.compute() -> C.compute()

- ❖ Instruction Cache
is happier with this

```
public class ICacheExample {
    public static void main(String[] args) {
        ArrayList<A> la = new ArrayList<A>();
        ArrayList<B> lb = new ArrayList<B>();
        ArrayList<C> lc = new ArrayList<C>();
        // ...
        for (A item : la) {
            item.compute();
        }
        for (B item : lb) {
            item.compute();
        }
        for (C item : lc) {
            item.compute();
        }
    }
}
```

Lecture Outline

- ❖ Locality
- ❖ Caches
- ❖ **Memory Allocation & fragmentation**
- ❖ Being aware of memory allocation in C++
- ❖ `std::move`

The Heap

- ❖ The Heap is a large pool of available memory to use for Dynamic allocation
- ❖ This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- ❖ **new**
 - searches for a large enough unused block of memory
 - marks the memory as allocated.
 - Returns a pointer to the beginning of that memory
- ❖ **delete:**
 - Takes in a pointer to a previously allocated address
 - Marks the memory as free to use.

Free Lists

- ❖ One way that malloc can be implemented is by maintaining an implicit list of the space available and space allocated.
- ❖ Before each chunk of allocated/free memory, we'll also have this metadata:

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```

KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Dynamic Memory Example

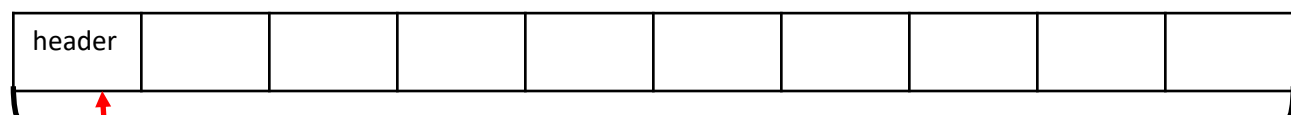
```

int main() {
  char* ptr = new char[4];
  int* ptr2 = new int[6];
  ...           // do stuff with ptr
  delete ptr;
  delete ptr2;
}

```

This diagram is
not to scale

❖ free_list ->



```

{
  NULL,
  NULL,
  false,
  1024
}

```

The metadata is at
the beginning of the
chunk of memory

KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Dynamic Memory Example

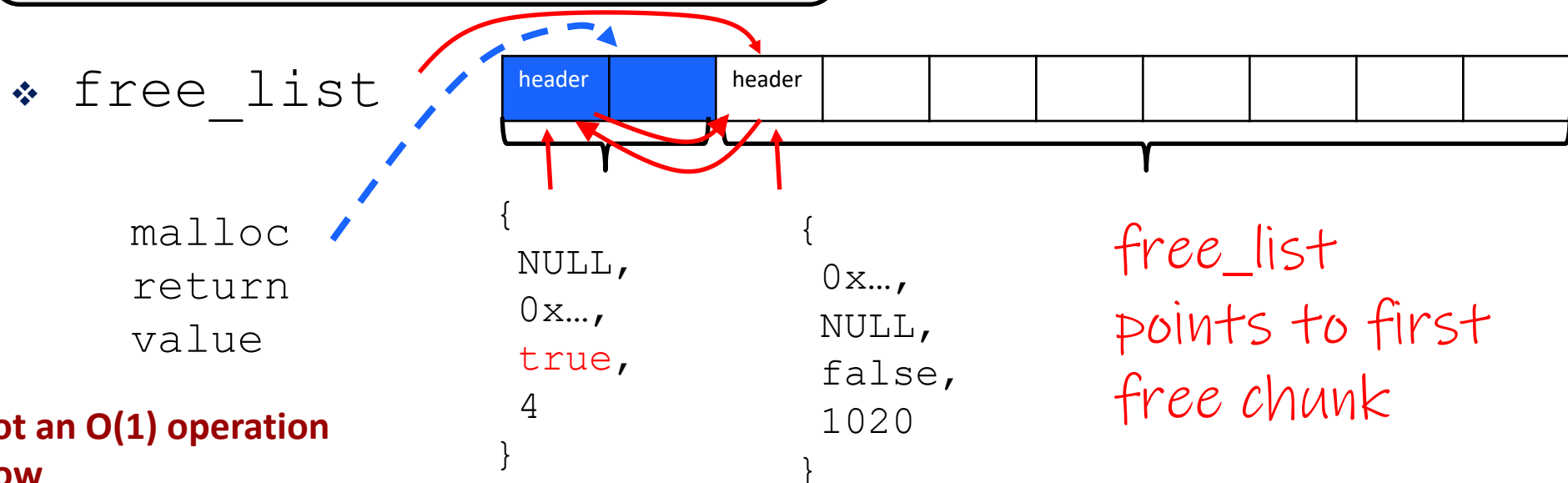
```

int main() {
  char* ptr = new char[4];
  int* ptr2 = new int[6];
  ...           // do stuff with ptr
  delete ptr;
  delete ptr2;
}

```

Free chunks can be split to allocate blocks of specific size

new returns a pointer to just after the metadata



KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

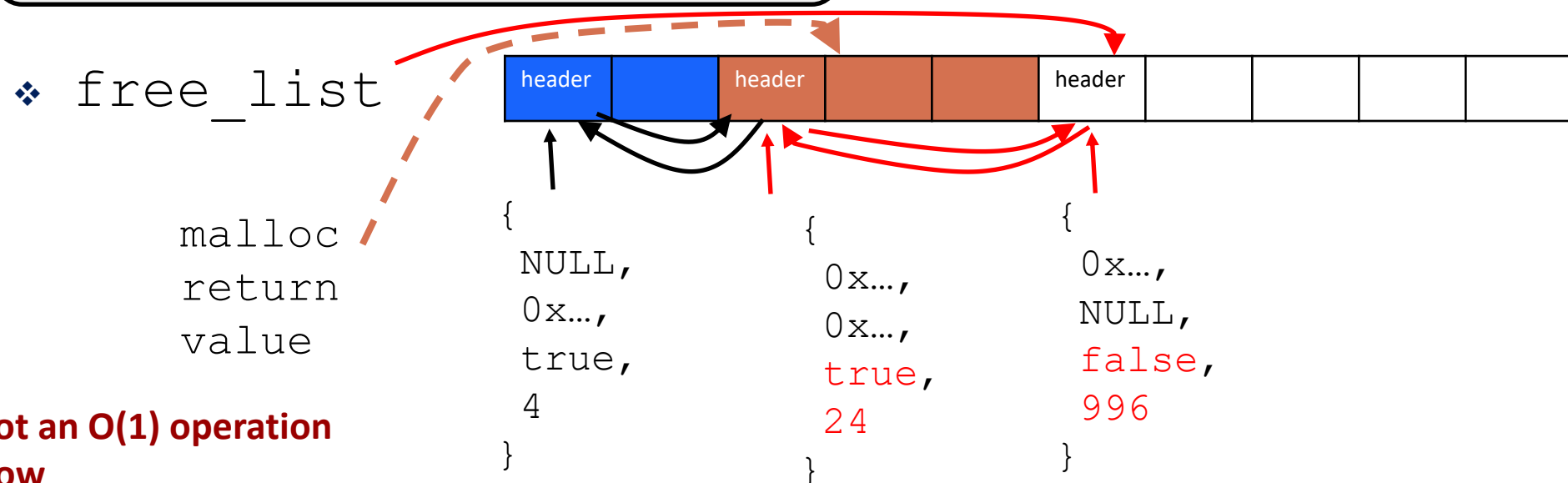
It is complex and slow

Dynamic Memory Example

```

int main() {
  char* ptr = new char[4];
  int* ptr2 = new int[6];
  ...           // do stuff with ptr
  delete ptr;
  delete ptr2;
}

```



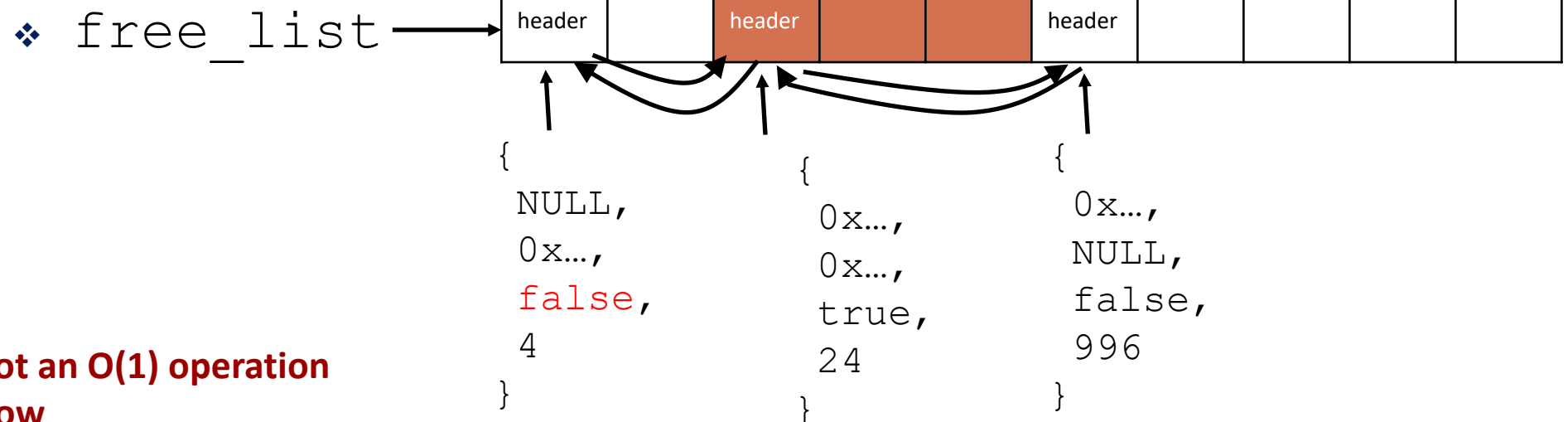
KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Dynamic Memory Example

```
int main() {  
    char* ptr = new char[4];  
    int* ptr2 = new int[6];  
    ...           // do stuff with ptr  
    delete ptr;  
    delete ptr2;  
}
```



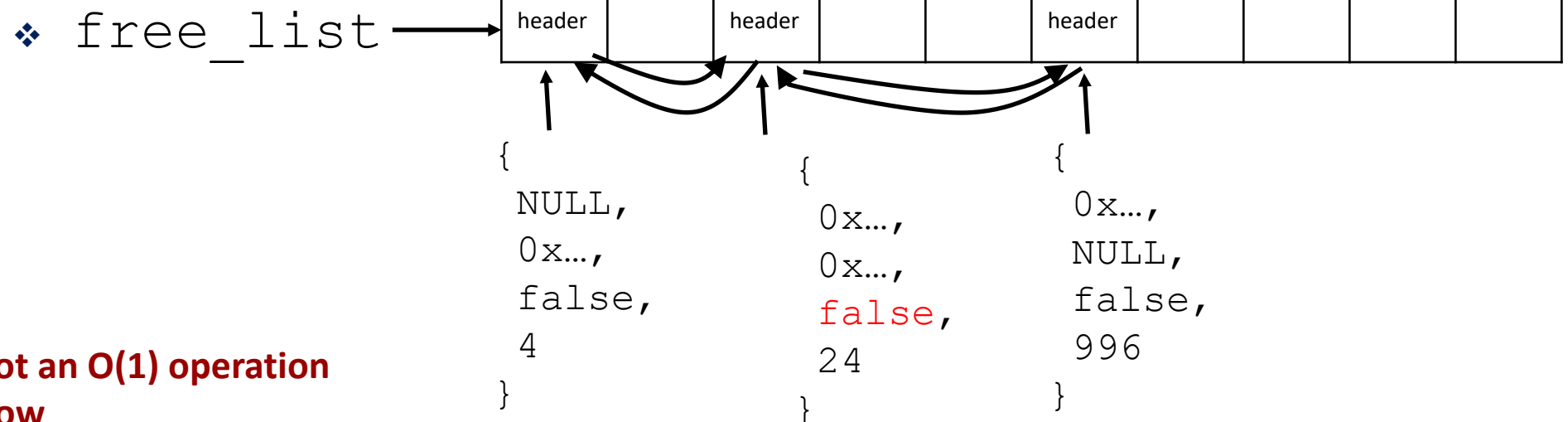
KEY TAKEAWAY:
Using the heap is not an $O(1)$ operation
It is complex and slow

Dynamic Memory Example

```

int main() {
    char* ptr = new char[4];
    int* ptr2 = new int[6];
    ...           // do stuff with ptr
    delete ptr;
    → delete ptr2;
}

```



KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Dynamic Memory Example

```

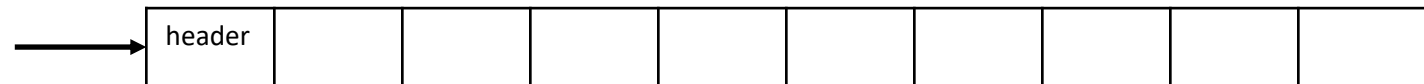
int main() {
    char* ptr = new char[4];
    int* ptr2 = new int[6];
    ...           // do stuff with ptr
    delete ptr;
    delete ptr2;
}

```

Once a block has been freed, we can try to "coalesce" it with their neighbors

The first free couldn't be coalesced, only neighbor was allocated

❖ free_list



```

{
    NULL,
    0x...,
    false,
    1024
}

```

KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Heap

- ❖ **new** and **delete** are not system calls, they are implemented as part of the C++ std library
 - **new** and **delete** will sometimes internally invoke system calls to expand the heap if needed
 - Instead, these functions just manipulate memory already given to the process, marking some as free and some as allocated
- ❖ System calls used by **new** and **delete**:
 - **brk ()** and **sbrk ()**
 - Used to grow/shrink the data segment of memory
 - **mmap ()**, **munmap ()**
 - creates / or destroys a mapping in virtual address space

KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Fragmentation

- ❖ Fragmentation: when storage is used inefficiently, which can hurt performance and ability to allocate things.

Specifically, when there is something that prevents "unused" memory from otherwise being used

- ❖ External Fragmentation: when free memory is spread out over small portions that cannot be coalesced into a bigger block that can be used for allocation

KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

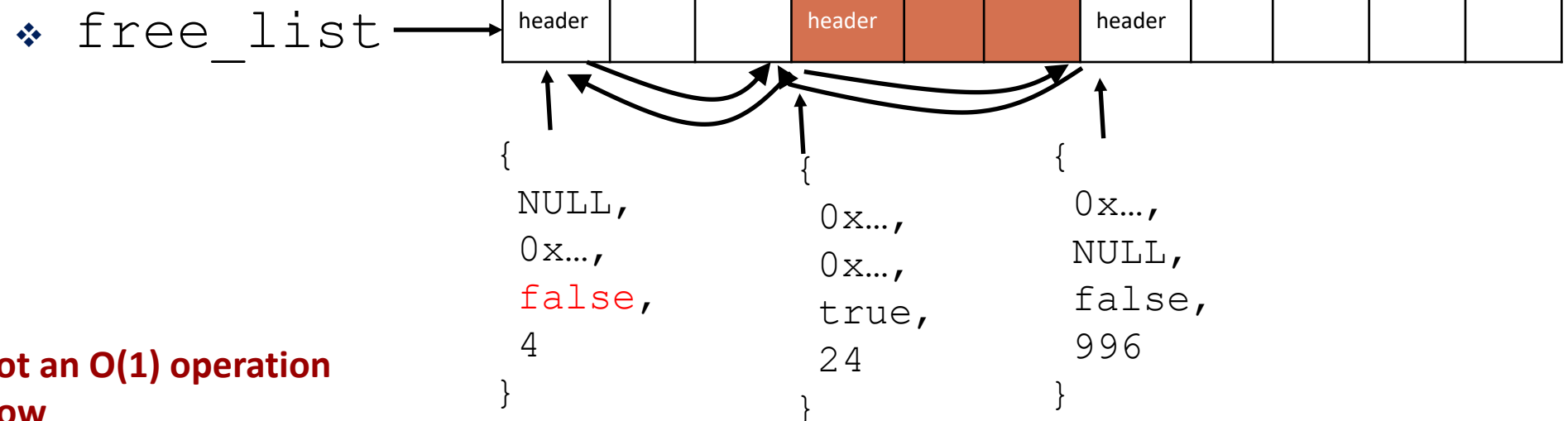
It is complex and slow

External Fragmentation Example

```

int main() {
    char* ptr = new char[4];
    int* ptr2 = new int[6];
    ...           // do stuff with ptr
    delete ptr;
    ptr = new char[2];
    ...
}

```



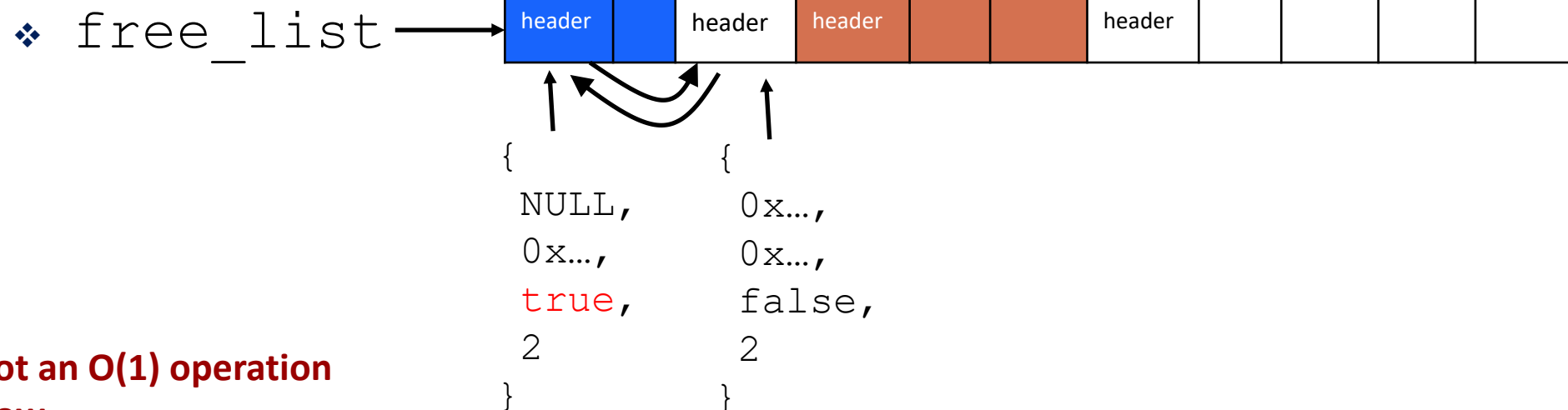
KEY TAKEAWAY:
Using the heap is not an $O(1)$ operation
It is complex and slow

External Fragmentation Example

```

int main() {
    char* ptr = new char[4];
    int* ptr2 = new int[6];
    ...           // do stuff with ptr
    delete ptr;
    ptr = new char[2];
    ...
}

```



KEY TAKEAWAY:
 Using the heap is not an $O(1)$ operation
 It is complex and slow

External Fragmentation Example

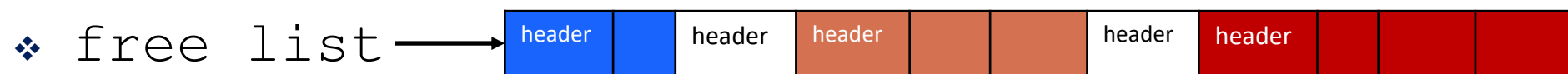
```

int main() {
  char* ptr = new char[4];
  int* ptr2 = new int[6];
  ...           // do stuff with ptr
  delete ptr;
  ptr = new char[2];
  ...
}

```

After some more series of allocations and frees (not shown), we get this:

Let's say `new char[4]` gets called (trying to allocate 4 bytes) what happens?



There are 4 bytes of free space, but they aren't next to each other and can't be coalesced into something that can be used. Heap would need to grow to make space (if possible)

```

{
  0x...,
  0x...,
  false,
  2
}

```

```

{
  0x...,
  0x...,
  false,
  2
}

```

KEY TAKEAWAY:

Using the heap is not an $O(1)$ operation

It is complex and slow

Lecture Outline

- ❖ Locality
- ❖ Caches
- ❖ Memory Allocation & fragmentation
- ❖ **Being aware of memory allocation in C++**
- ❖ `std::move`

Memory Allocation in C++

- ❖ We rarely call `new` or `delete` directly in C++ code, but it is called implicitly all the time if we are not careful
 - Whenever a data structure needs more space
 - Whenever we copy construct an object that needs allocation
 - Etc.

 **Poll Everywhere**pollev.com/tqm

❖ Which function is faster?

```
void print_vec(ofstream& to_print, const vector<string>& words) {  
    for (const string word : words) {  
        to_print << word << "\n";  
    }  
}
```

```
void print_vec(ofstream& to_print, vector<string>& words) {  
    for (size_t i = 0; i < words.size(); i++) {  
        string& word = words[i];  
        to_print << word;  
        to_print << "\n";  
    }  
}
```




Poll Everywhere

pollev.com/tqm

- ❖ How many memory allocations occur in each piece of code?
 - Assume vector resizes will double capacity
 - `std::list` is a linked list in C++

```
int main() {  
    vector nums {4, 8}; // size and capacity == 2  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
}
```

```
int main() {  
    list nums {4, 8};  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
}
```

Minimizing Allocations

- ❖ As we saw previously, memory allocations require time, sometimes a lot of time to compute.
- ❖ If performance is our goal, we should minimize the number of allocations we make.
- ❖ This can include
 - Making references instead of copies
 - Using functions like `vector::reserve(size_t new_capacity)`
 - Java arraylist lets you specify capacity in the constructor.
 - `std::string` also has a reserve function
 - Using move semantics

Lecture Outline

- ❖ Locality
- ❖ Caches
- ❖ Memory Allocation & fragmentation
- ❖ Being aware of memory allocation in C++
- ❖ **std::move** Did not get to move in lecture

That's it for now!

❖ See you next lecture 😊