Effizientes Programmieren in C, C++ und Rust

Memory Management (in C)

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

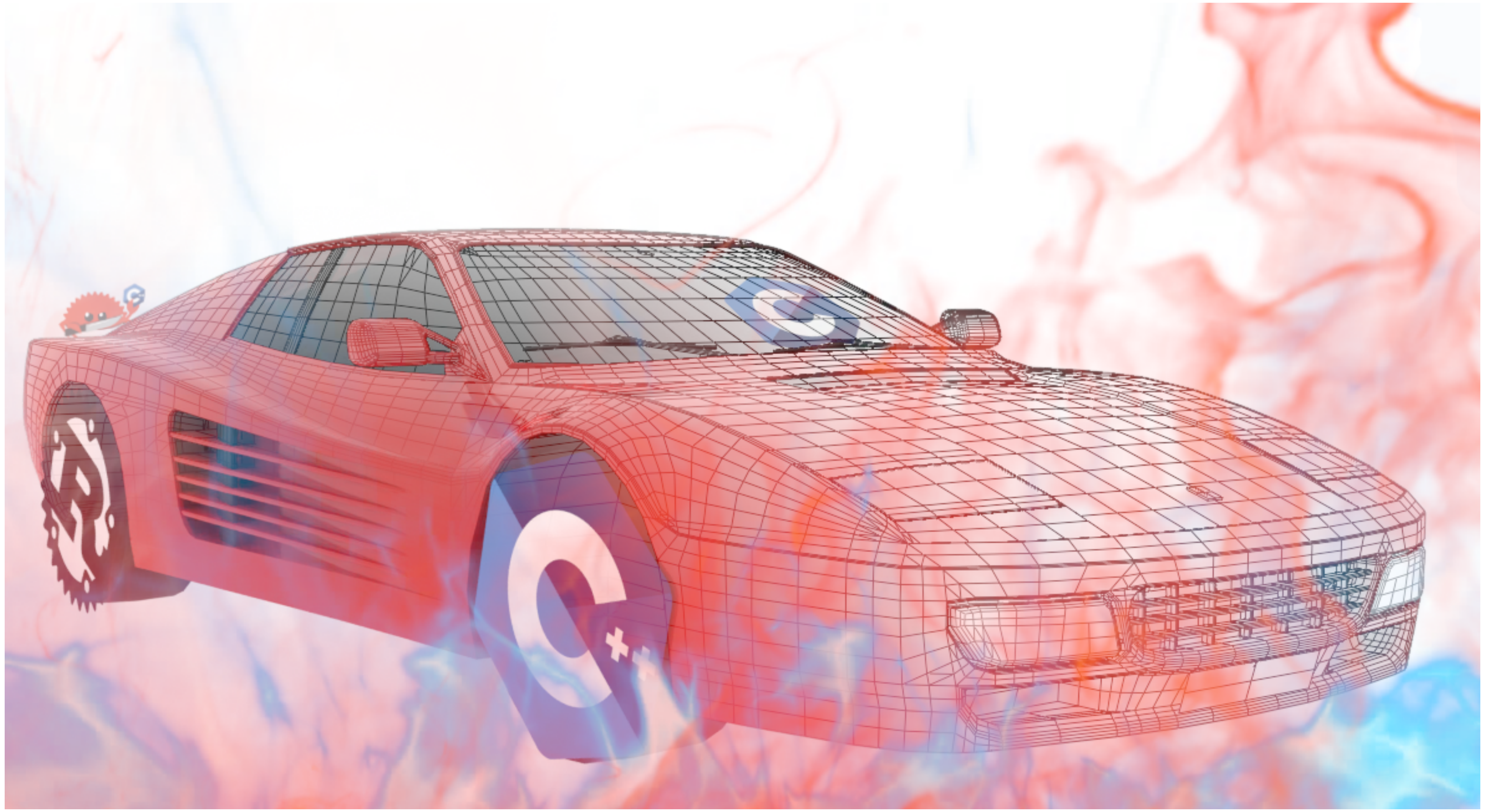
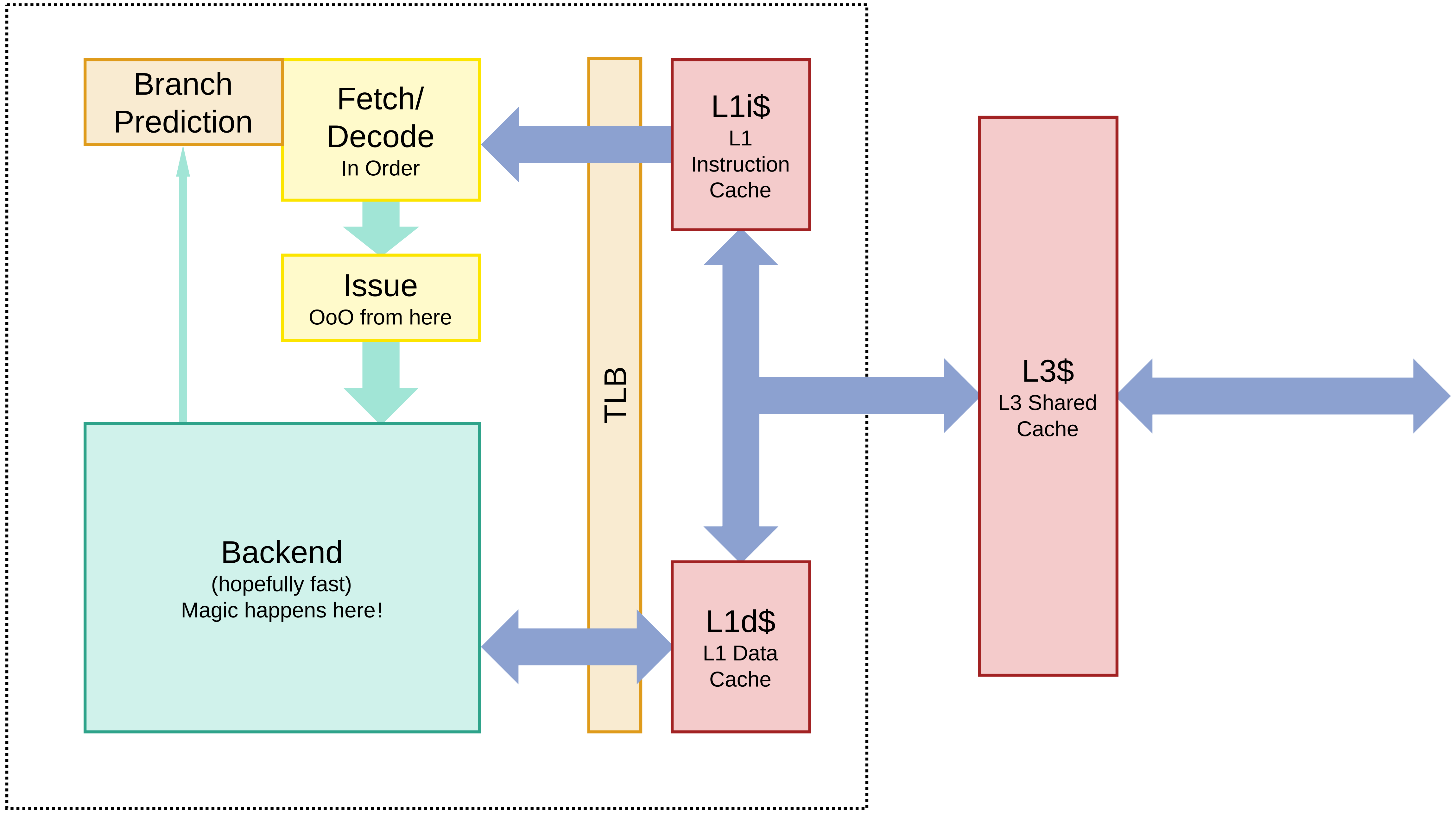


Table of Contents

Last Lecture

Of The C of Machines



This lecture: μEffCpp

C: EFFICIENCY & PERFORMANCE🚴

- Why? (not)
- Control Flow
- (Types)
- (Functions)
- (`struct`)

What Is a Memory?

The CPU Side of Things

HOW DOES THE CPU SEE MEMORY? DISCUSS!

Basics

- CPU reads memory
  - Explicit `load` for data
  - Implicit `fetch` for instructions
- CPU writes memory
  - Explicit `store`

Optimizations

- Caches
- Prefetchers
- ...

The OS

- Virtual memory (what we usually use)
- *Translated* into physical memory
- TLB: accelerates translation (see OS)

What Is a Memory?

Even more basic CPU view

`LOAD`

- load: $\mathbb{Z} \to \mathbb{Z}$
- load(address)$\to$`int8`/`int16`/`int32`/`int64`

`store`

- store: $(\mathbb{Z}, \mathbb{Z}) \to ?$
- store(address, value)
- $?$: Technically the memory state that was changed

Takeaway: We only have numbers in a computer

Judging Memory Performance

Case Study: Vector Addition

Data layout ideas (for O(N) algorithms)

- (Linked) list
  - Address: follow next pointer (of previous element)
- HashMap: index $\rightarrow$ number
  - Address: see Algo 1
  - Probably: follow a linked list
- Array of numbers
  - Address: base-address + offset
  - Offset: index * data width (`sizeof(data_element)`)
- Java-Style: array of "references" to numbers
  - Address: follow 1 pointer at array location

Best?🚴 Very probably the array. Lets find out why!

Judging Memory Performance

Implement in C: The Linked List

Notes: always implement `first += second;`

DATA DEFINITION

```c
struct ll_vector {
    ll_vector *next;
    int number;
};
```

Let's talk about pointers

- Pointer: contains address (or `0`)
- Address: "Index" of data in memory
- `0` or `NULL` or (C23) `nullptr`: Explicitly not a valid address
- Used to modify data at other location
- Very small memory footprint

IMPLEMENTATION

```c
void add_ll(struct ll_vector first, struct ll_vector second) {
    while(first.next != 0) {
        first.number += second.number;
        first = *first.next;
        second = *second.next;
    }
}
```

Is that correct? NO!

Issue: We modify a *copy!*

Fix:

```c
void add_ll(struct ll_vector *first, struct ll_vector *second) {
    while(first->next != 0 && second->next != 0) {
        first->number += second->number; // modify correct data
        first = first->next;
        second = second->next;
    }
}
```

Judging Memory Performance

Intermission: C Memory Operations

DATA ⇒ ADDRESS

- Operator: `&`
- Name: "Address-of" operator
- Example: `pointer = &number;`

ADDRESS ⇒ DATA

- Operator: `*`
- Name: "Dereference" operator
- Example (reading at address): `number = *pointer;`
- Example (writing to address): `*pointer = number;`
- Operator: `->`
- Name: "Struct Dereference" operator
- Example: `list->number += 7;`
- Equivalent to: `(*list).number += 7;`

Judging Memory Performance

Intermission: C Pointer Declarations

- Pointers need to know the type they are pointing to
- Why: `struct` and pointer arithmetic
- Declared: `TYPE *name;`
- Read *RIGHT TO LEFT* and inside out
- `int *declared_pointer;`
- Read: `declared_pointer` is a *pointer* (`*`) to an `int`.

TYPE QUIZ

- `int **p1;` `p1` is a *pointer* to a *pointer* to `int`
- `void *p2;` `p2` is a *pointer* to nothing.
- `void *`: if you do not know the type, but the address. Very common in C!
- `void (*p3)(void);` `p3` is a *pointer* to a *function* taking no parameters and returning nothing. "`void(void)`"
- `void* (*p4)(void (*p5)(void));` `p4` is a *pointer* to a *function* taking a *pointer* to a `void`→`void` *function* and returing a `void` *pointer*
- And `p5`? Is ignored ...

Implement in C: The Hash Map

We do not want to ...

Judging Memory Performance

Implement in C: Array

DATA DEFINITION

```
struct arr_vector {

    int *numbers;
    size_t num_elements;
};
```

IMPLEMENTATION

```
void add_arr(struct arr_vector fst,
                  struct arr_vector scn) {

    for (int i = 0; i < fst.num_elements; ++i) {
        fst.numbers[i] += scn.numbers[i];
    }
}
```

Is that correct? Yes

```
int *ptr;
addressof(ptr + 1) - addressof(ptr) == ????
```

Let's talk pointer arithmetic

- Pointer: "just a number"
- Arithmetic: Use offset for related memory
- Array: tuple (pointer to first address, number of elements)
- Still: very small memory footprint

- 4! Reason: `sizeof(int) == 4`
- Access into "object" not meaningful!
- If access into object required? cast to `(char *)`

Judging Memory Performance

Intermission: C Pointer Arithmetic

- `operator++` and `operator--`: increase and decrease pointer
- Points to next/previous *element, not byte*
- Modifies pointer
- Prefix and postfix: prefix returns new value, postfix old one
- `int arr[4] = {1, 2, 3, 4};`
  `int *ptr = arr;`
- `*(++ptr)`? returns 2
- `*(ptr++)`? (still) returns 2

- `operator+` and `operator-`: add to and subtract from pointer
- Again: steps in terms of elements
- `operator[foo]`: shorthand for *(ptr + foo)
- `operator+=`: also modifies ptr
- `int arr[4] = {1, 2, 3, 4};`
- `*(arr + 2)`? returns 3
- `arr[2]`? (still) returns 3
- `int *ptr = arr; *(ptr += 3)`? returns 4

Judging Memory Performance

Implement in C: Java-Style Array

DATA DEFINITION

```
struct JVector {
```

```
    int **numbers;
    size_t numElements;
};
```

IMPLEMENTATION

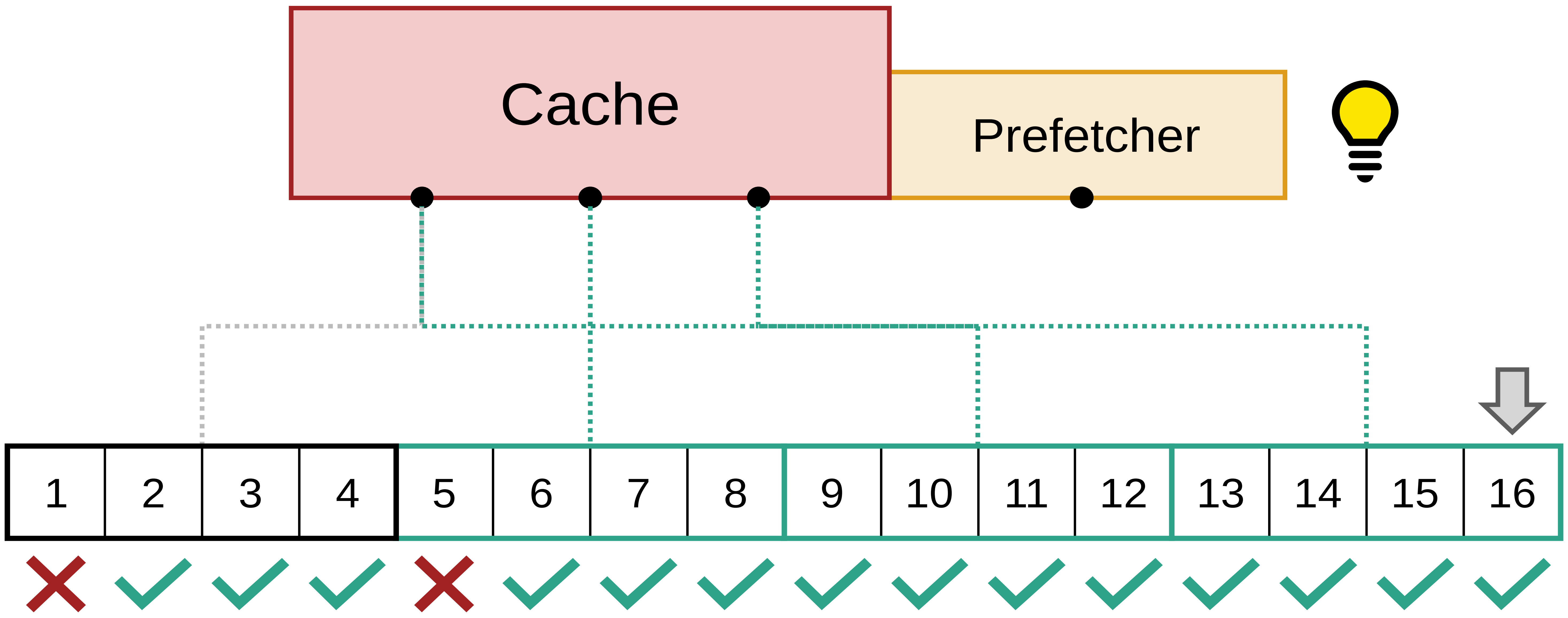
```
void add_arr(struct JVector first,
             struct JVector second) {
```

```
    for (int i = 0; i < first.numElements; ++i) {
        (*first.numbers[i]) += (*second.numbers[i]);
    }
}
```

JUHU, EXTRA INDIRECTION. GREAT!

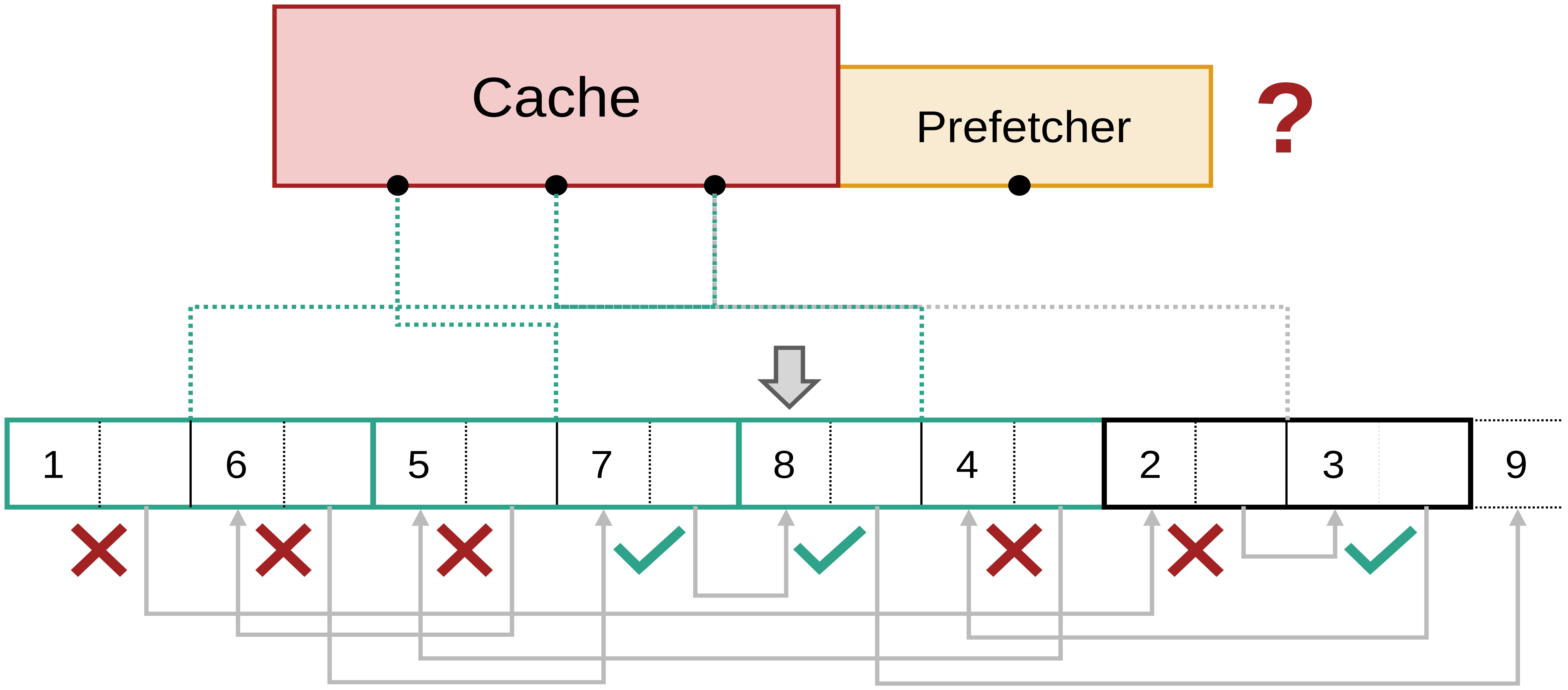
Judging Memory Performance

Scan over Plain Array



Judging Memory Performance

Scan over Linked List



Judging Memory Performance

Performance Discussion

LINKED LIST

- 👎: No random access
- 👎: Pointer chasing
- 👎: Hard to prefetch
- 👎: Memory overhead ⇒ less caching

HASHMAP

- 👍: Random access
- 👎: Hard calculations (hashing)
- 👎: Hard to prefetch (hashing)
- 👎: Memory overhead
- 👎: Hard to implement

ARRAY

- 👍: Random access
- 👍: One addition
- 👍: Easy to prefetch (linear)
- 👍: No memory overhead

JAVA-STYLE ARRAY

- 👍: Random access
- 👍: One addition + one fetch
- 👎: Hard to prefetch (indirection)
- 👎: Memory overhead
- 👎: Completely pointless

Aside: `struct` Performance

Yes, you can go wrong here

Assume (for this slide): 4 `int` cache lines, `sizeof(int)==sizeof(pointer)`

```
struct tree {
    struct tree *next;
    struct tree *previous;
    struct tree *parent;
    struct tree *child;
    int data;
};
```

Any issues with this data layout?

- Spans two cache lines (unavoidable)
- What most-used access pattern? (usually) traversal for data
- next + data always on different cache lines
- Does prefetcher help? Only somewhat, two accesses are faster than the time to prefetch

```
struct tree {
    struct tree *next;
    int data;
    struct tree *previous;
    struct tree *parent;
    struct tree *child;
};
```

Fixed?

- Much better, but not completely. Why?
- `tree` is `int` aligned, but 5 `int` long ⇒ sometimes still between to cache lines
- Fix? Padding

Actually Initializing Our Vectors

STACK

- Local memory of function
- Cannot be `return`ed (!) (Cleaned up on scope exit)
- Passing into called function ok? Yes
- Fixed size per function
- 🚴Performance: Extremely hot, basically always in L1 Cache ⇒ use often

HEAP

- Global memory, shared
- Can be `return`
- Dynamically sized
- 🚴Performance: Depends on usage pattern

STATIC MEMORY

- "Forbidden" Category...
- Use with extreme caution
- 🚴Performance: Similar to heap

Stacks and Scopes

Where Variables Deserve to Die

```
int main() {
    struct ll_vector last = {0, 2};
    struct ll_vector mid = {&last, 1};
    { // explicit scope
        struct ll_vector first = {&mid, 0};
    } // first stops living here
} // last, mid stop living here
```

- Scope: "area" of common livetime of variables
- Lifetime: time of valid use
- Lifetime: Starts at definition, ends at enclosing scope close
- Scopes (in C): basically everything between { and }
- Scopes: functions, loop/if bodies
- Not a scope: `struct` body

- (CPU) stack: More capable than Algo 1 stack data structure
  - New variable: *pushed* to end (usually low address)
  - After scope exit: *popped* from end
  - (Hence the name)
- Additionally: random access in stack frame (area of function)
- Recursion: One function has multiple stack frames, one for each recursive call

Heaps

Heap, Heap, I'm a Sheep

```c
int *iota(int num, int start) {
    /* create values start, start + 1,
    start + 2, ..., start + num - 1*/
    int *values = // help!!!
```

What kind of memory do we want?

- Persists over function boundaries
- Can be "safely" shared across threads
- Type-independent
- Dynamically sized
- "Leak-free"
- Fast allocations

C solution:

```c
void *malloc(size_t num_bytes);
void free(void *pointer);
```

Issues?

- **Alignment**: `char arr[sizeof(long long)];` and `long long` have same size, but different alignment
  - Solution: Assume worst-case alignment
- **Locality**: No option to allocate, e.g., list in adjacent memory
  - Solution: No real solution
- **Fragmentation**: No option to allocate memory with similar livetimes in similar memory
  - Solution: fancy heuristics in fast malloc implementations

`malloc()` Implementations

`free()`-styler!

DESIGN IMPLICATIONS

- `malloc()` needs to go into `free()`
  - Only information: the pointer
  - We need to track all sizes of allocated memory
  - Store somewhere (e.g., around allocated memory)
- Need to reuse memory: find previously `free()`d memory
  - Need a lookup structure
  - For fast implementations: acceleration structure
- Need to track whether new memory needs to be requested ⇒ OS!
  - Issue: Fragmented size does not count, only contiguous free memory
- Need to actually get memory from the OS...

I want to `brk()` free

`mmap()` men, `mmap()` men, `mmap()` `mmap()` `mmap()` men

GETTING MEMORY FROM THE (LINUX) OS

SBRK()

`sbrk()` sets "program break", the size of the shared data segment

- Can be used for small allocations
- Needs data structure that shows whether lowest data can be freed to shrink "break"
- Big performance issue: Extremely diverse size of allocations
  (1 - 128 kiB), similar sizes should be adjacent to combat fragmentation and to speed-up lookups.

MMAP()

`mmap()` maps files (and chunks of memory) into address space

- Can be used for all allocations
- Technical limitation: multiples of page size (usually 4 kiB)
- OS needs exact information on class="text-unimportant" `munmap()`
- Setup security features like "Guard Pages"
- Performance issue: demand paging and pre paging

`malloc()` Shenanigans

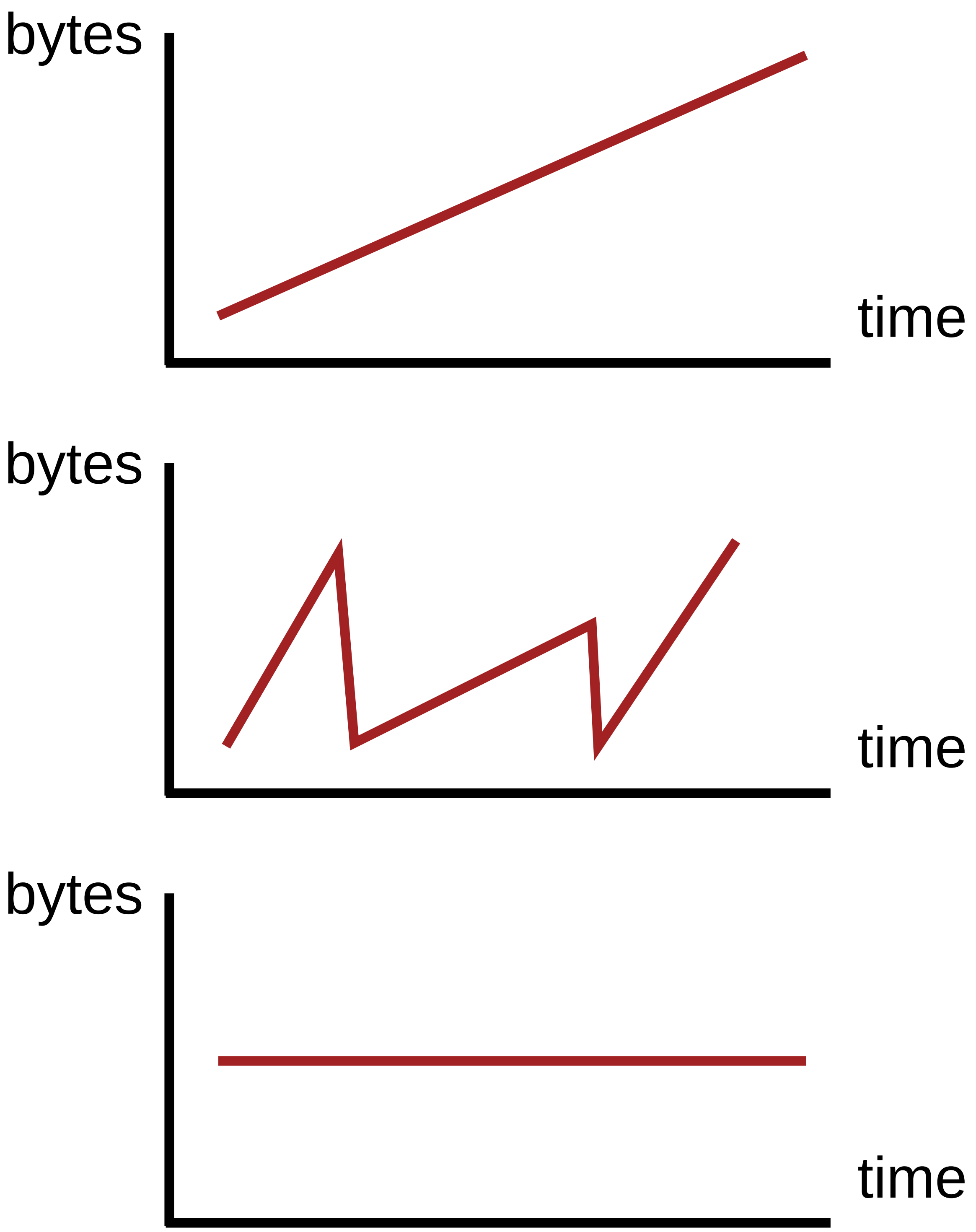
Embrace the Penguin

- Does `malloc()` fail? No (except in OS assignments). Still check!
- Why not? Demand Paging
- What happens on out of memory? Out of Memory (OoM) killer
- What is targeted by OoM killer? Applications with a lot of memory, e.g., the X-server, the desktop manager, or *your application...*
- What do you do then? End other applications or get more RAM

Allocation Strategies

`OS *copy = malloc(pasta)`: Allocation Patterns

- Ramp
  - Allocate more and more, does not free
  - Example: file system cache
- Peaks
  - Allocate in phases
  - Example: pipelined processes
- Plateau
  - Does not allocate, (only at beginning)
  - Example: long computations



Applications have some typical allocation pattern
adapted from source

Allocation Strategies

Please a Bit - Bitmap Allocator

- Pick a size: e.g., 64 byte (cache line!)
- Reserve a memory range for actual storage
- Reserve a memory space where you store one bit per byte
- Allocate by rounding up size, finding consecutive free block large enough
- Set bits for "allocated"
- On free? Need to find length
- Malloc-style allocator: size lookup somewhere else
- Other, great idea: use only for sizes ≤ blocksize, free becomes trivial

PERFORMANCE🚴

- Bit range: extremely hot ⇒ great
- size lookup for free() slow
- If single size (or range, e.g., 32-64 bytes): Extremely fast ⇒ *Slab Cache*
- Programs often allocate same size often

Allocation Strategies

Free List

- Linked List: [next, length]-segments
- Store in returned memory (no space overhead!)
- On alloc: iterate through list until length ≥ requested length
- Remove from list, return segment
- Possibly add new, smaller block to list
- Issue: alignment
- On free? Need to find length
- Simple idea: store before returned pointer (mind the extra space)
- Remember to merge segments if possible

PERFORMANCE🚴

- Implementation advantage: Simple to implement, ok-ish for all allocation sizes
- Performance issue: linked-list
- Expectation: Newly returned memory hot. Usually wrong, applications defer free()
- Remember? Programs often allocate same size often
- Usually, first try succeeds ⇒ actually decent!

Allocation Strategies

My Best Buddy - Buddy Allocator

- Allocatable memory size: $2^n, n \in \mathbb{N}$
- Allocation request: round up to next power of two
- Free-list of all blocks of required size. If empty: (recursively) allocate one power larger, split result and insert other block (= buddy) into free-list
- On free: insert now freed block into free-list. If buddy is also free, merge (recursively) to regain large blocks
- Issues:
  - Internal & external fragmentation
  - Space waste up to $\frac{1}{2} - \varepsilon$

PERFORMANCE🚴

- Possibly replace free-list with other tool.
- Advantage: Simple to implement
- Performance: binary split really fast
- Depending on free-list or replacement: merge slow(ish)
- Not (really) general purpose, but great building block!

Allocation Strategies

Welcome To the Arena

- Remember application phases? Wouldn't it be great to only cleanup once at the end of the stage?
- Idea: bin of memory (with parents)
- Allocate to bin that lives as long as needed, but not longer
- At the end of live: `deallocate_all(arena *)`
- Here: no real implementation advice, depends on your use case

PERFORMANCE🚴

- Hugely depends on correct usage, but then can be really fast
- No automatic allocation to correct arena possible, needs programmer input
- But does it? Machine learning to the rescue!
  - Learning-based Memory Allocation for C++ Server Workloads
  - Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond
  - Performance Test Selection Using Machine Learning and a Study of Binning Effect in Memory Allocators

Allocation Strategies

Composition is Key

- No allocator great alone
- But together: great building blocks
- General Pattern: Different behavior based on size (and alignment)

Example

- If size ≥ 2MB ⇒ call `mmap()` directly
- If 2MB > size ≥ 128kB ⇒ Use page-aligned free-list backed by `mmap()`
- All below: backed by buddy allocator, all buddy-managed memory initially 2MB aligned
- If 128kB > size ≥ 4kB ⇒ Use different page free-list
- If 4kB > size > 128 B ⇒ Use 64 B-aligned free-list
- If 128 B ≥ size > 64 B ⇒ Use 128 B slab cache
- If 64 B ≥ size > 48 B ⇒ Use 64 B slab cache
- If 48 B ≥ size > 32 B ⇒ Use 48 B slab cache
- If 32 B ≥ size > 16 B ⇒ Use 32 B slab cache
- If 16 B ≥ size ⇒ Use 16 B slab cache

Recap

What did we learn today?

- What is memory?
- What is a pointer?
- Big O is not everything
- Stack and Heap
- `malloc` and `free()`

And next lecture?

- Allocation Strategies & Performance Pitfalls
- Preprocessor
- Compiler
- Performance model: branch prediction and function calls
- `union`s