

[368] C++ Programming: Welcome!

Tyler Caraza-Harter

Outline

Welcome

Logistics

Background and Motivation

- Why C/C++: performance
- Why C++ (over C): language features

Demos

Introductions

Tyler Caraza-Harter

- Long time Badger
- Email: tharter@wisc.edu
- Just call me “Tyler” (he/him)



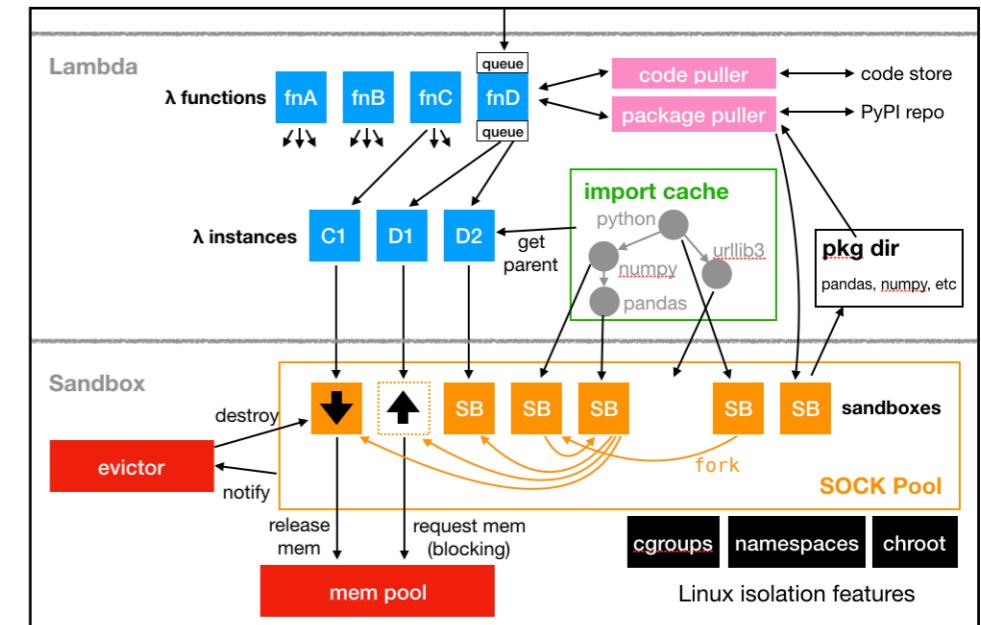
Industry experience

- Worked at Microsoft* on SQL Server and Cloud
- Other internships/collaborations:
Qualcomm, Google, Facebook*, Tintri*, Bauplan*

*used C++

Open source

- OpenLambda (serverless cloud platform)
- <https://github.com/open-lambda/open-lambda>



Who are You?

Year in school? Major?

What CS courses have people taken before?

- 300? 320? 354?

Please fill this form (**due today**):

<https://forms.gle/3BaLREBlupurZDky6>

Why?

- Help me get to know you
- Let me know your GitHub username (create an account if necessary) so I can grade your GitHub classroom submissions.

What will you learn in this 368?

Learning objectives

- Read and understand C++ code
- Write C++ programs making use of the abstractions provided by the language
- Understand the lower level details of memory management like pointers and references
- Organize and build multi-file projects using the make tool
- Solve real world programming problems using C++ as a tool

What will you learn today?

Learning objectives

- recall course logistics and policies
- describe reasons for using a language like C/C++
- describe reasons for using C++ over C

Outline

Welcome

Logistics

Background and Motivation

- Why C/C++: performance
- Why C++ (over C): language features

Demos

Main Websites

1

<https://tylercaraza-harter.com/cs368/s24/schedule.html>

- schedule, course content, how to get help
- links to all other resources/tools
- some lecture recordings (review only)

2

<https://github.com/cs368-wisc/s24>

- project specifications
- lecture demo code

3

Canvas

- announcements
- quizzes
- grade summaries

Other Tools

4

TopHat (me asking you questions during lecture)

- can earn points from this

5

Piazza (asking questions of **general interest**)

- goal: responses <1 business day
- don't post >5 lines of project code

6

Email (asking questions of **individual interest**)

- goal: responses <2 business days
- please keep related issues on the same thread

7

GitHub classroom

- you'll be given a **private** repo for your project

8

Anki Flash Cards

- memory terms, basic ideas using flash cards and spaced repetition

Lecture

Wednesday:

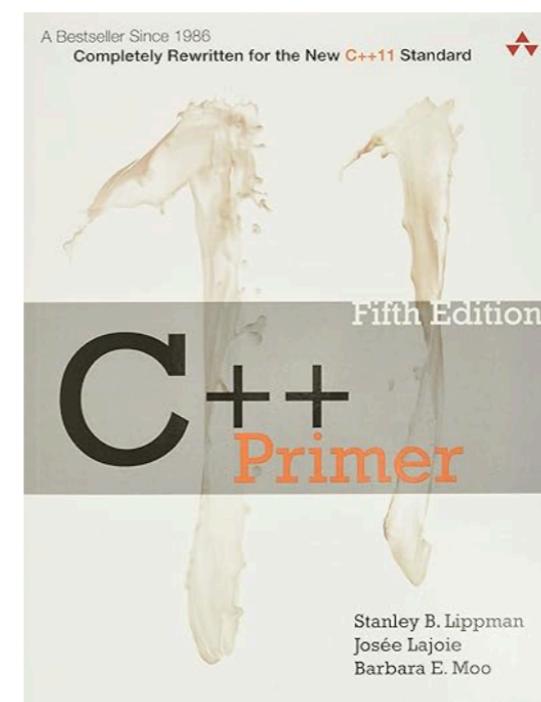
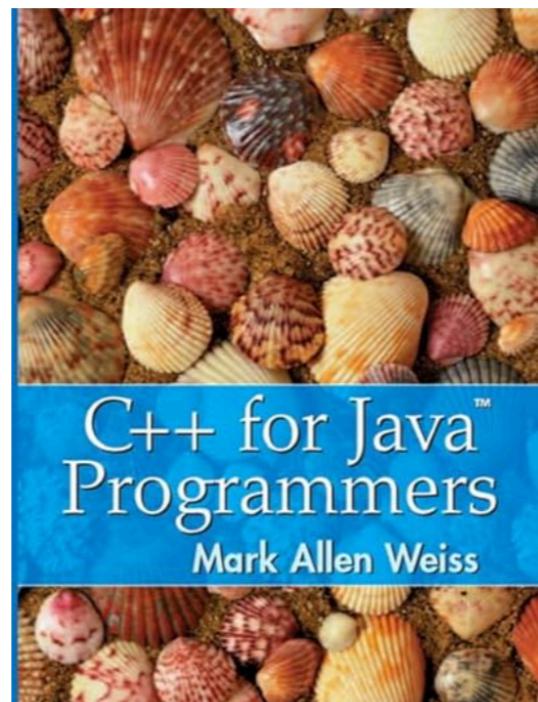
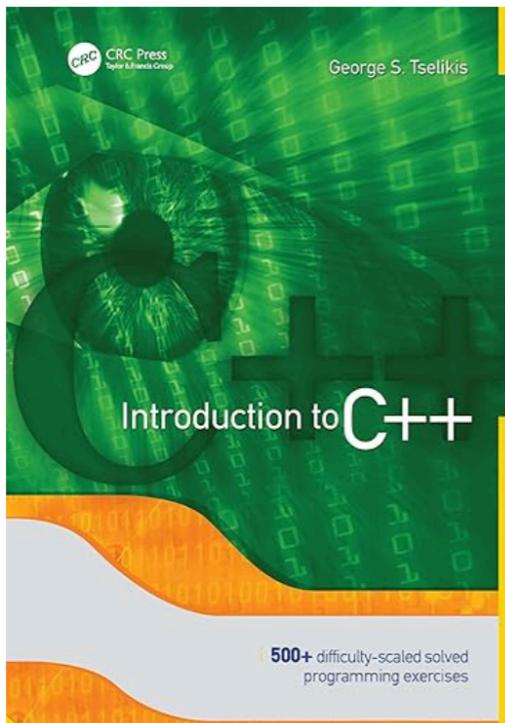
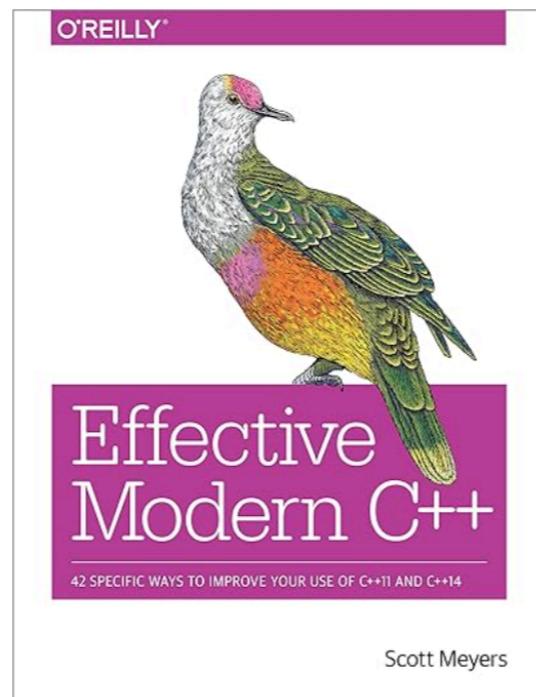
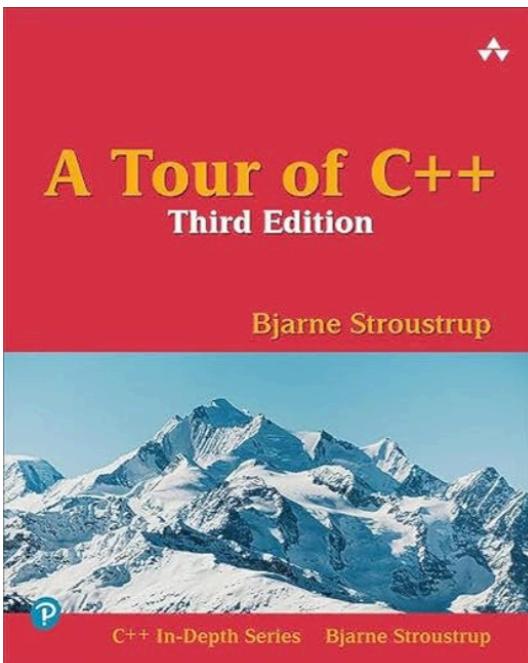
- in person (usually recorded too, barring technical difficulties)
- focus on concepts (lecture, worksheets, etc)
- TopHats

Friday:

- posted online, multiple short videos
- focus on programming demos
- watch before next in-person class!

A Few Books (Optional Supplement)

There is no assigned text (will cover everything needed in lecture). But here are a few books to consider if you want to supplement:



Note: the entire O'Reilly collection is free online through the Madison Public Library using a library card (also free)

Sparrow Project

Project:

- one big project with six project stages (P1 - P6)
- project name: Sparrow (simple prototype of Arrow)
- Arrow project (<https://arrow.apache.org/>) enables fast in-memory analytics on tables of data; the main implementation is in C++

Collaboration:

- done individually
- can help each other debug (with citation)
- sharing code is not allowed



Submission:

- you will push your code to a GitHub classroom repo (keep it private!)
- submit a form when a specific version (commit number) is ready for grading

Grading:

- autograded using tests I'll release
- I might manually modify grades if anybody tries to "game the tests"

Grading

This course is credit/no credit:

- so pass/fail, no letter grades
- to pass, you need to earn ≥ 100 points
- there will be > 150 points possible to earn, so there are many possible ways to pass
- 100 is a low bar; 140+ would be a score to really "feel good" about

Scoring:

- projects: 120 points possible (4 per passed test)
- quizzes: 30 points possible (1 per correct answer)
- TopHat: 1 point for correct answer, 0.5 for incorrect
- other: I might offer other opportunities for points as we go

Outline

Welcome

Logistics

Background and Motivation

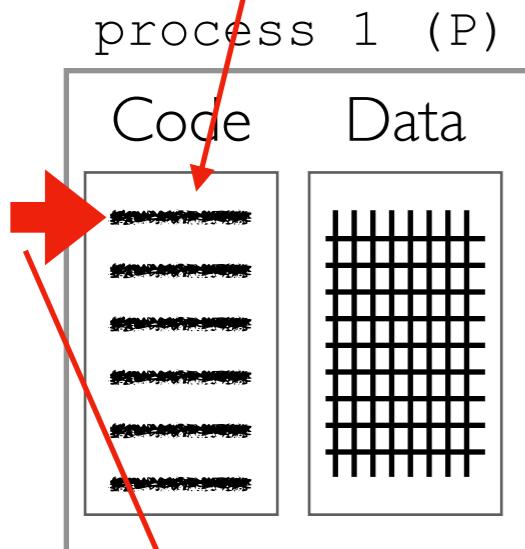
- Why C/C++: performance
 - how code runs
 - cachelines
 - garbage collection
 - safety checks
- Why C++ (over C): language features

Demos

Background: How to Code Runs on CPUs

these instructions are in "machine code"
that the CPU can understand

<https://pages.cs.wisc.edu/~deppeler/cs354/reference/x86-cheat-sheet.pdf>



these instructions are
represented as 1's and 0's
the CPU understands

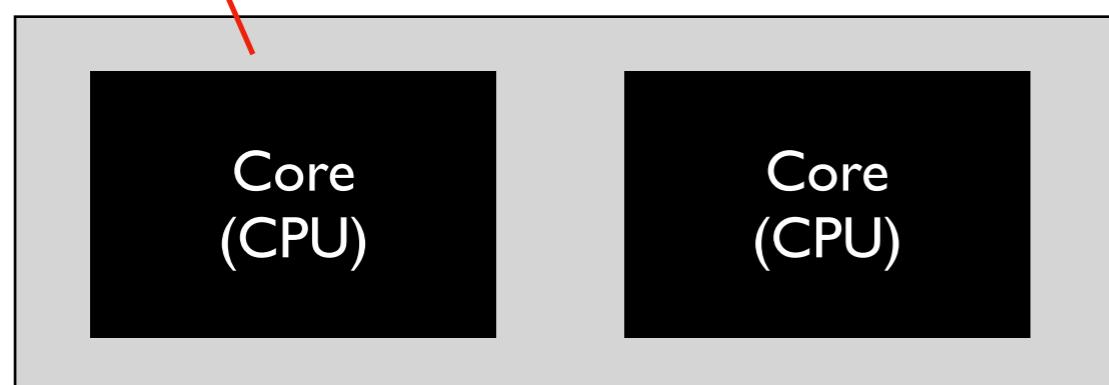
arithmetic
two operand instructions

```
addl src,dst    dst = dst + src
subl src,dst    dst = dst - src
imull src,dst   dst = dst * src
sall src,dst    dst = dst << src (aka shll)
sar1 src,dst    dst = dst >> src (arith)
shrl src,dst    dst = dst >> src (logical)
xorl src,dst    dst = dst ^ src
andl src,dst    dst = dst & src
orl src,dst     dst = dst | src
```

one operand instructions

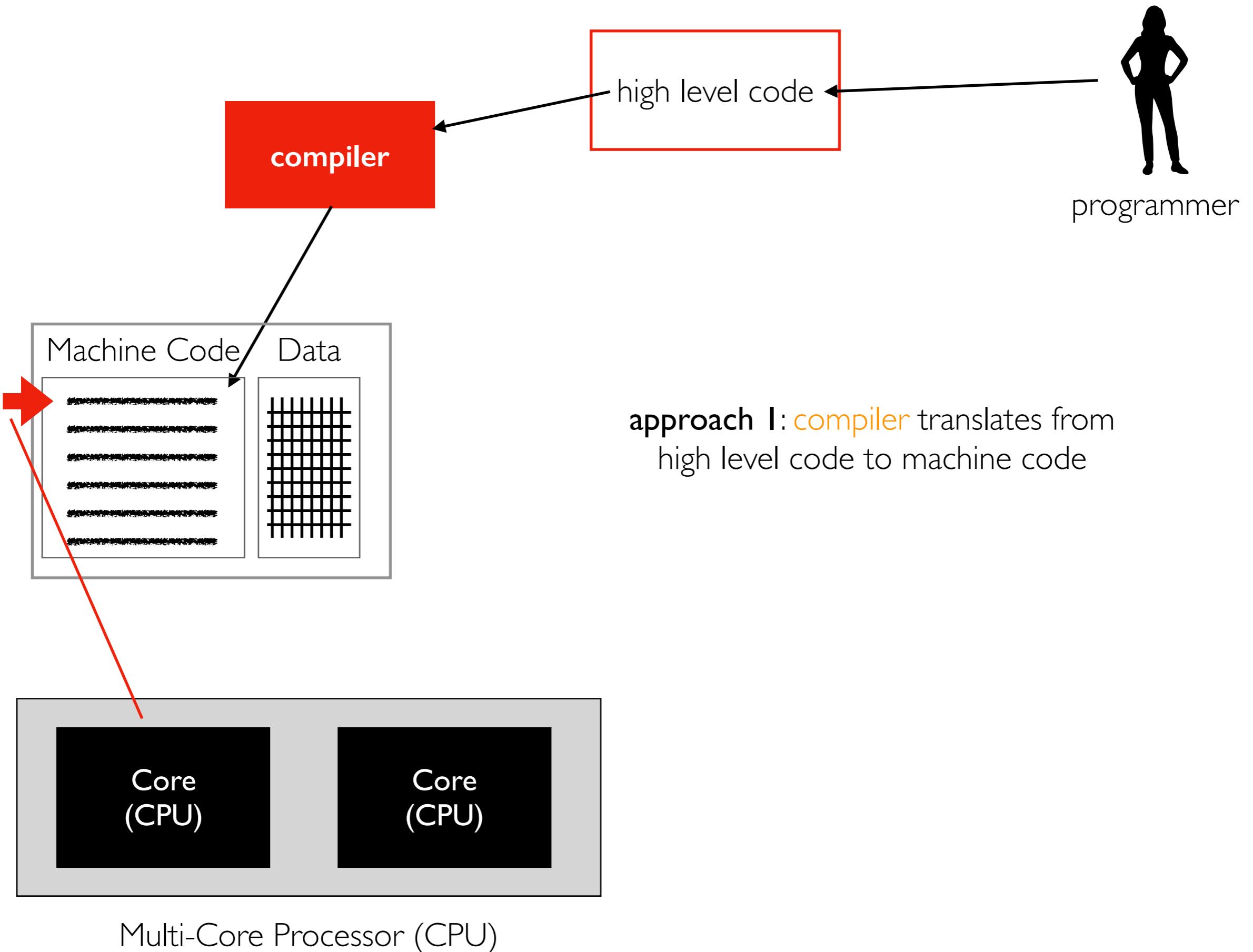
```
incl dst        dst = dst + 1
decl dst        dst = dst - 1
negl dst        dst = -dst
notl dst        dst = ~dst
```

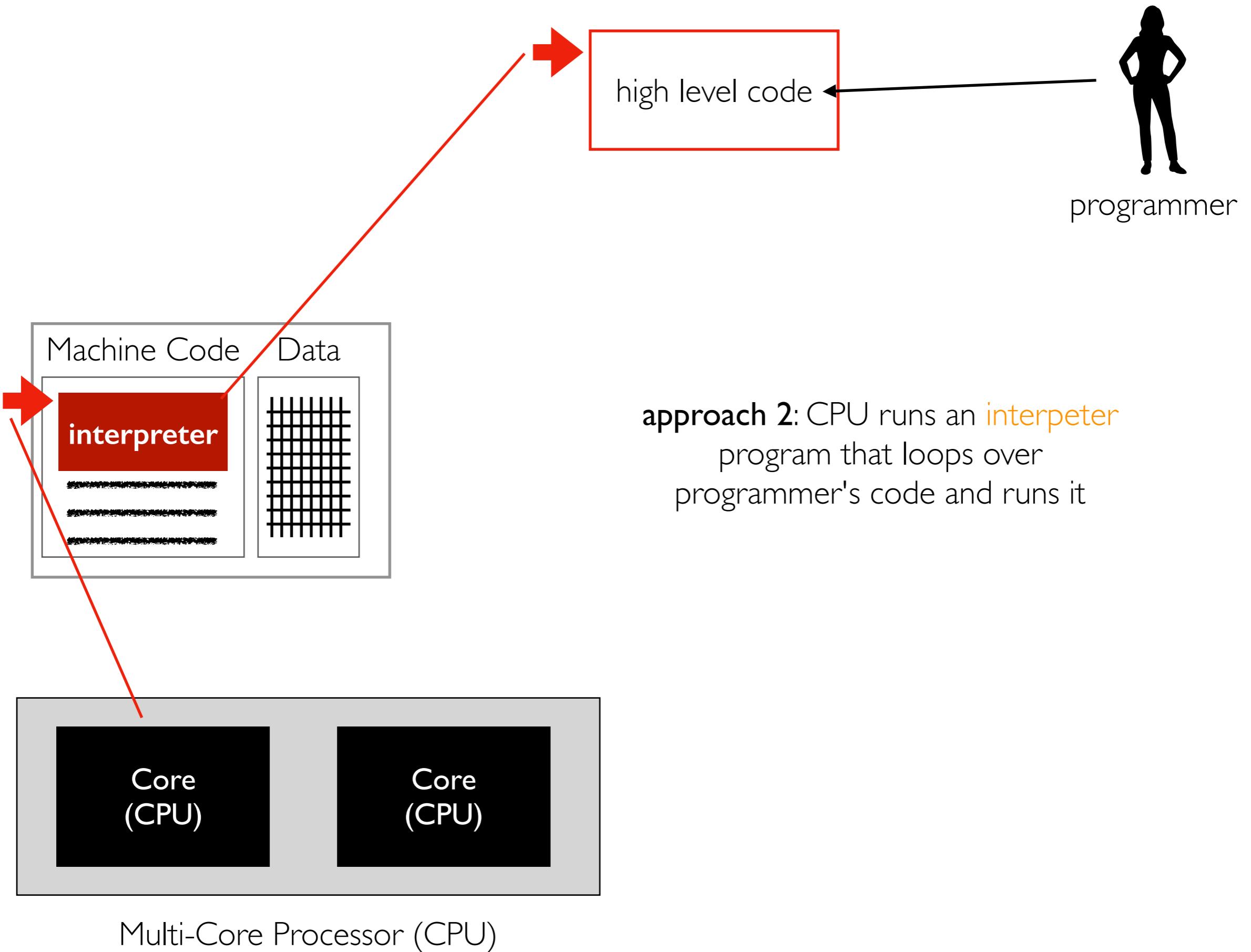
arithmetic ops set CCs implicitly
cf=1 if carry out from msb
zf=1 if dst==0,
sf=1 if dst < 0 (signed)
of=1 if two's complement
(signed) under/overflow

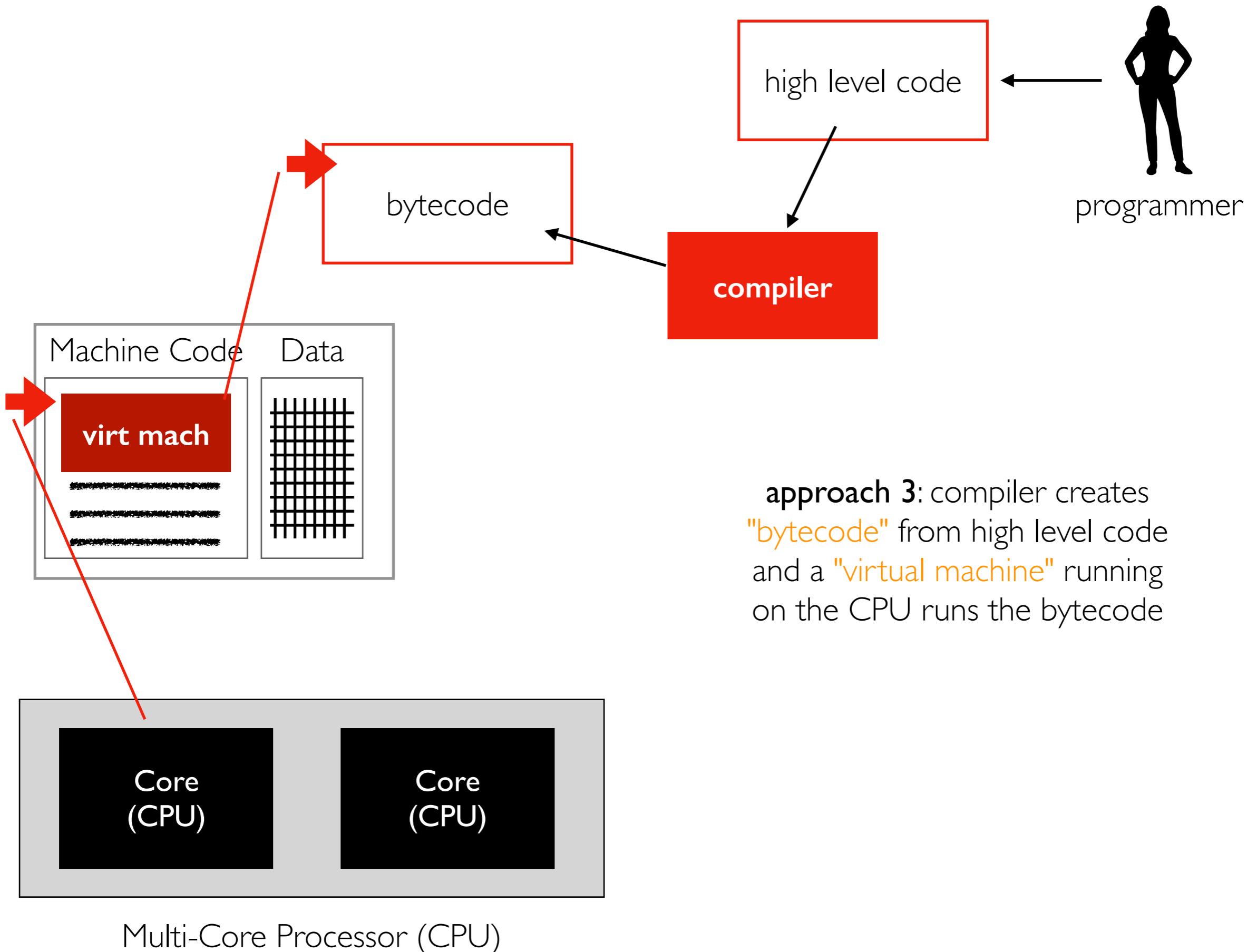


Multi-Core Processor (CPU)

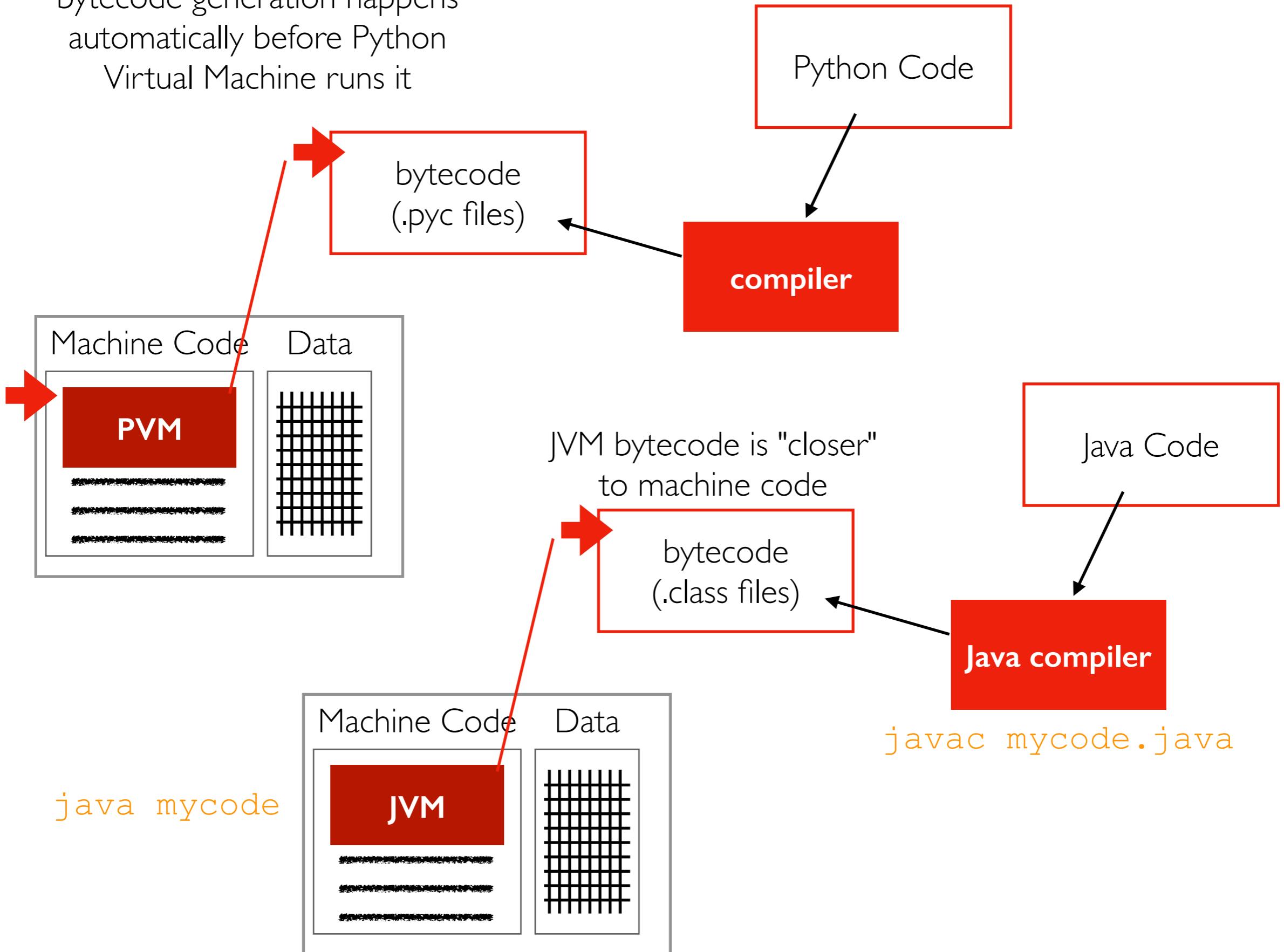
how do we bridge the gap between "high level" code (C++/Python/Java/etc) and machine code?







when you run "python3 ..."
bytecode generation happens
automatically before Python
Virtual Machine runs it



C/C++ Performance

Advantage I: compiled languages are *usually* faster at runtime

- no overhead due to interpreter or language virtual machine
- however, cannot dynamically profile+optimize

Outline

Welcome

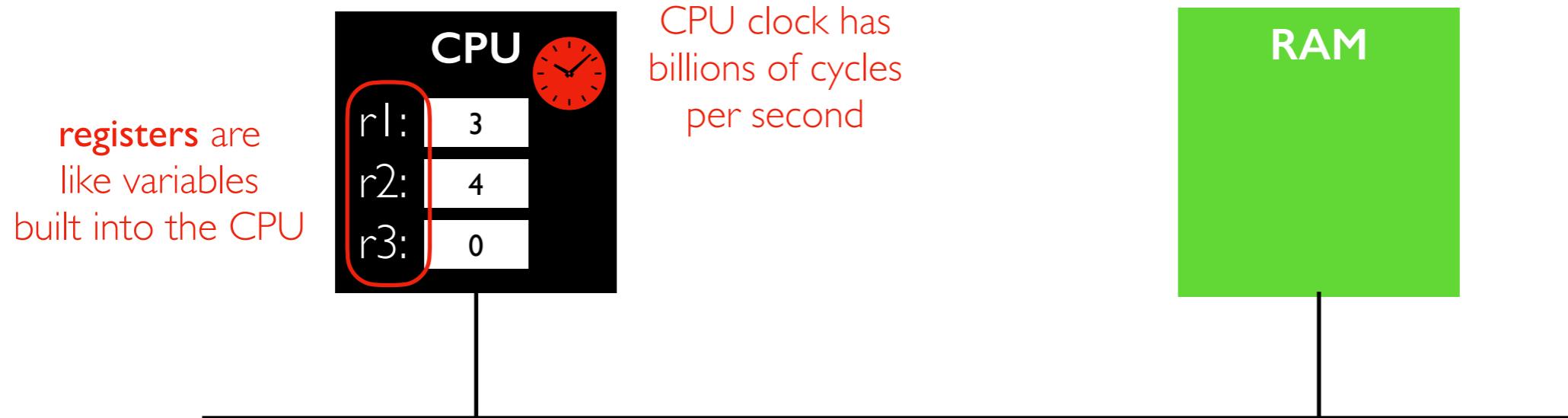
Logistics

Background and Motivation

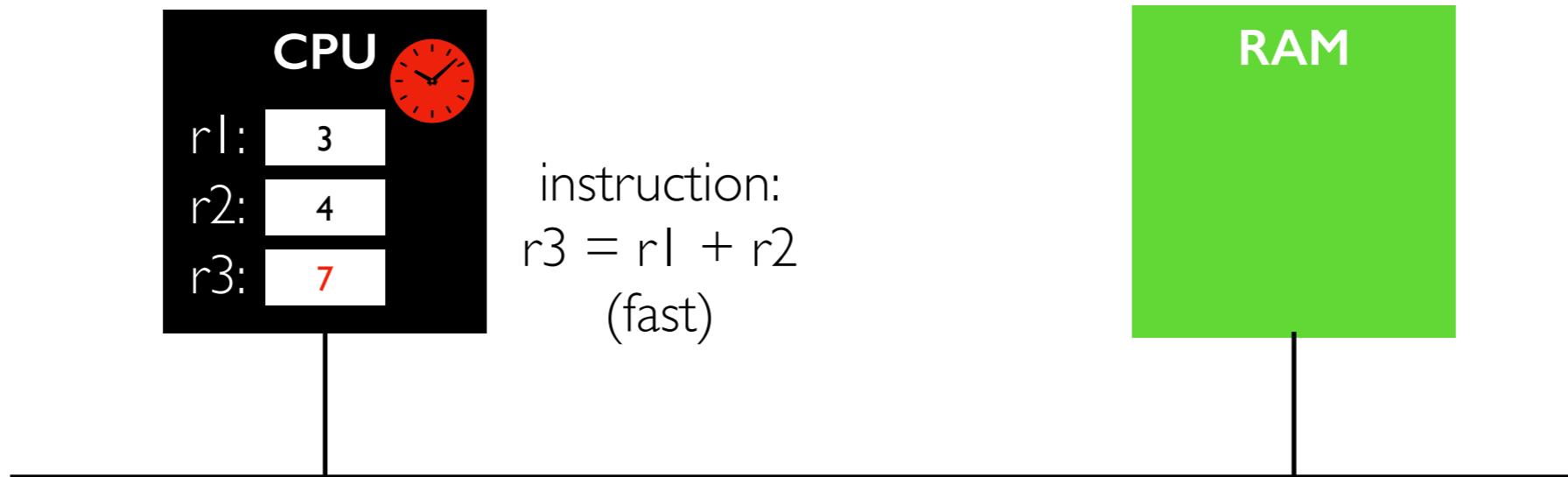
- Why C/C++: performance
 - how code runs
 - **cachelines**
 - garbage collection
 - safety checks
- Why C++ (over C): language features

Demos

Background: CPU and RAM



Background: CPU and RAM



Load and Store

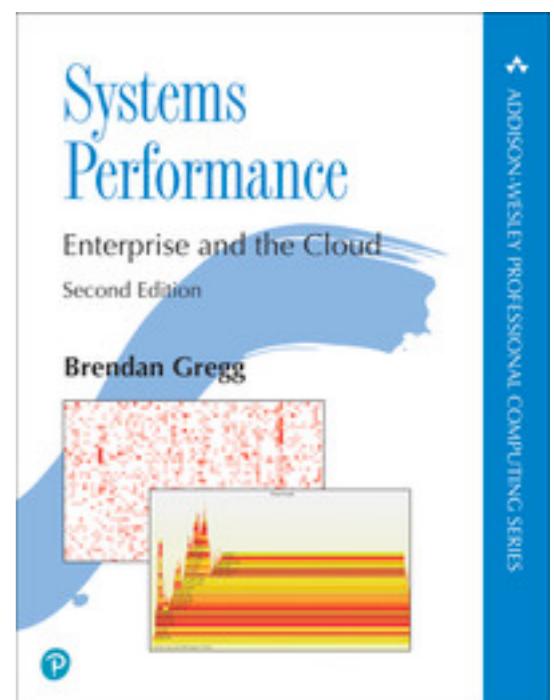


challenge: if we want to add some numbers stored in RAM, we need to **load** before adding and **store** after

Latency



very slow, but not long enough to switch to a different process...



source: visuals, estimates

Cache



What happens:

- the value needed (for example, a 4-byte integer) goes to the register
- a whole **cacheline** (often 64 bytes) containing the value goes to the cache
- future accesses to values in same cacheline will be relatively fast!

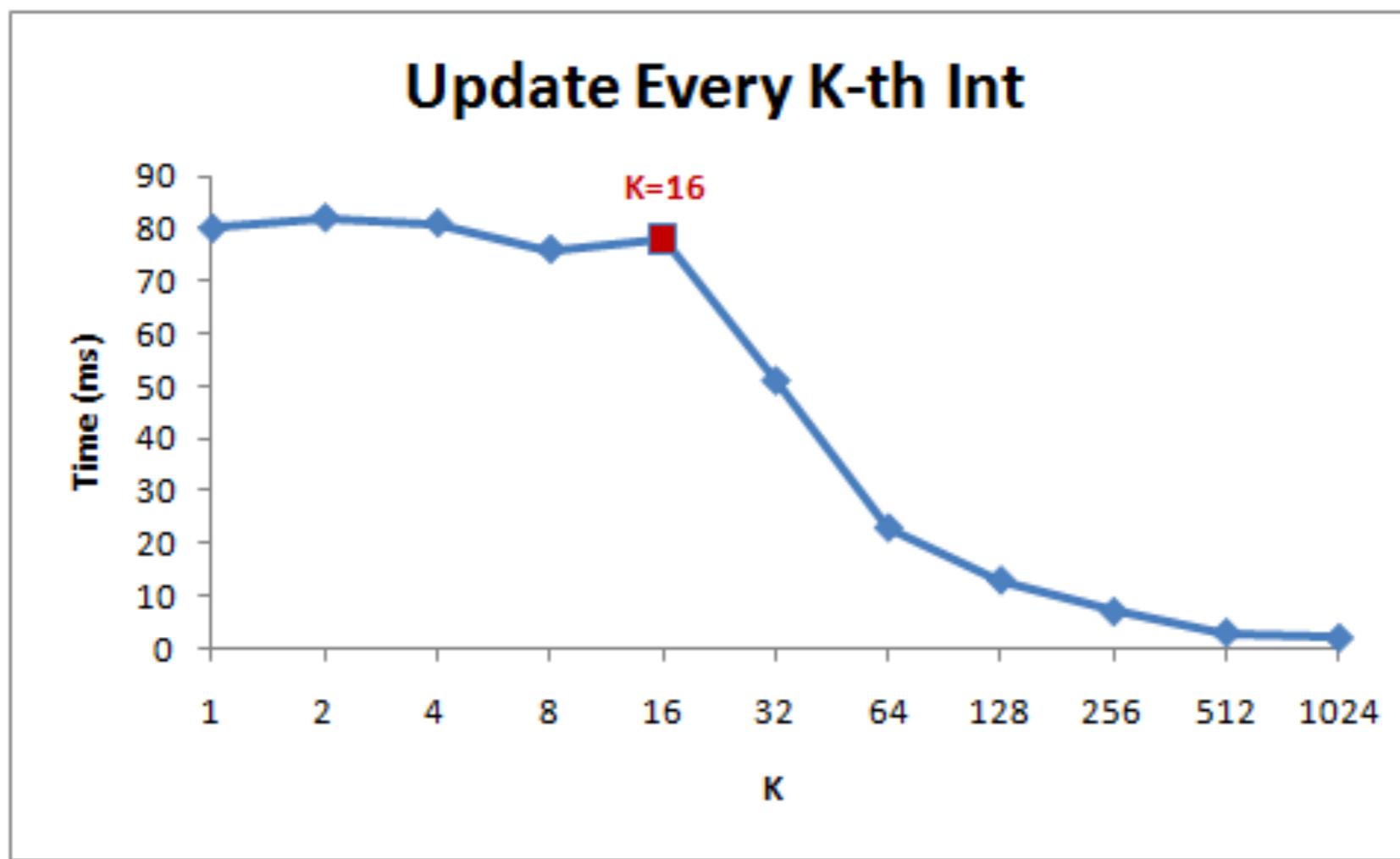
What matters for performance:

- how many cache misses there are (that is, how many times we need data that is not in the cache)
- how many values we access is less important

Example 1: Step and Multiply

as K gets bigger, we do fewer multiplications. But does it matter?

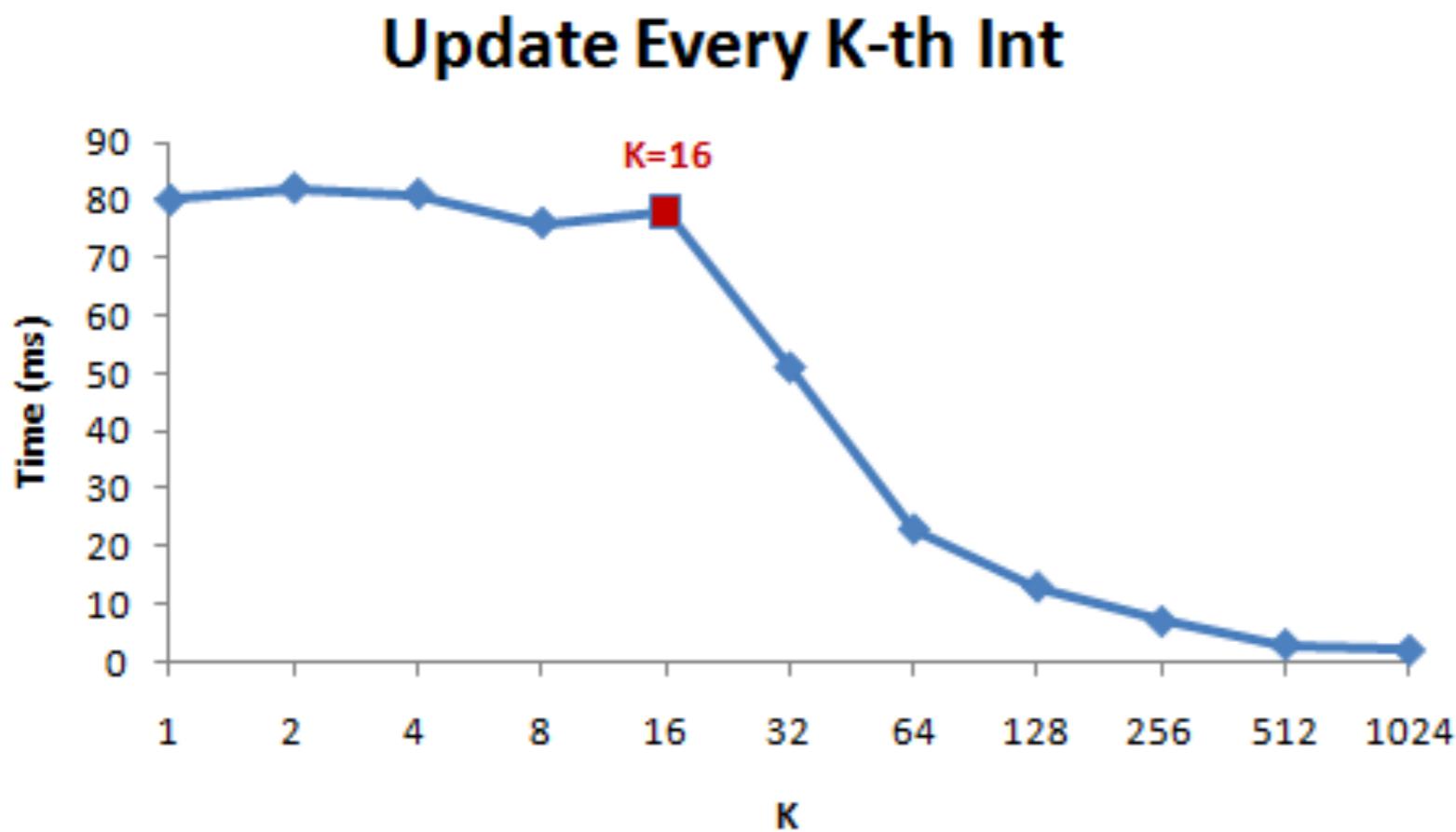
```
for (int i = 0; i < arr.Length; i += K) arr[i] *= 3;
```



[Gallery of Processor Cache Effects](#)

<http://igoro.com/archive/gallery-of-processor-cache-effects/>

Example I: Step and Multiply



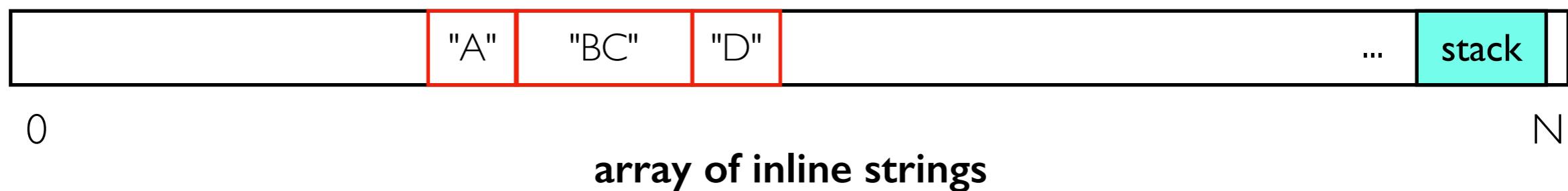
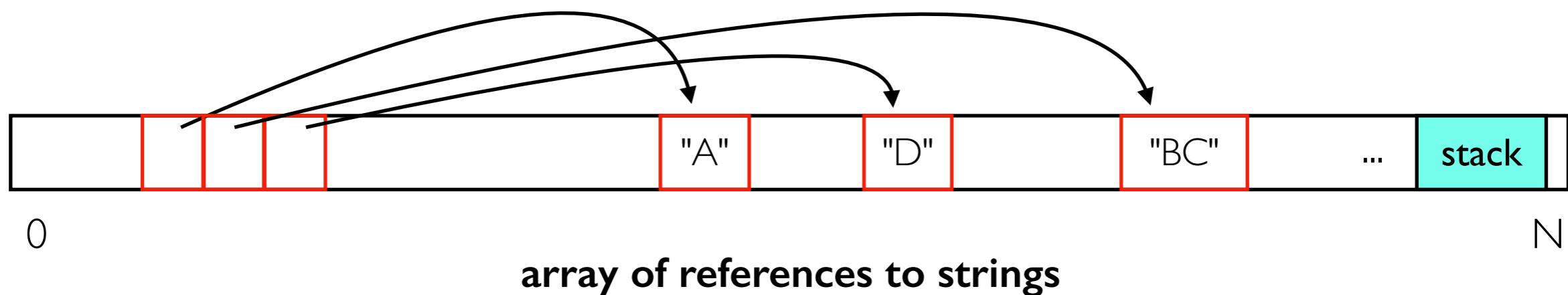
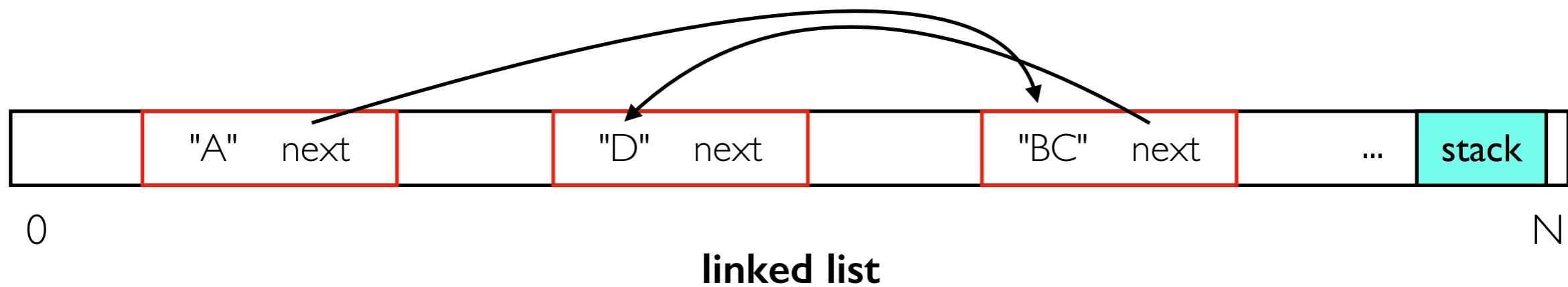
performance tip: think about how many cachelines you're touching, not just about how many values

k=1 loop: all the ints, all the cachelines

k=2 loop: **half** the ints, **all** the cachelines

Example 2: Series of Strings

which layout is most cache friendly?



C/C++ Performance

Advantage 1: compiled languages are *usually* faster at runtime

- no overhead due to interpreter or language virtual machine
- cannot dynamically profile+optimize

Advantage 2: C/C++ gives us more control over memory layout

- can design cache-friendly data structures

Outline

Welcome

Logistics

Background and Motivation

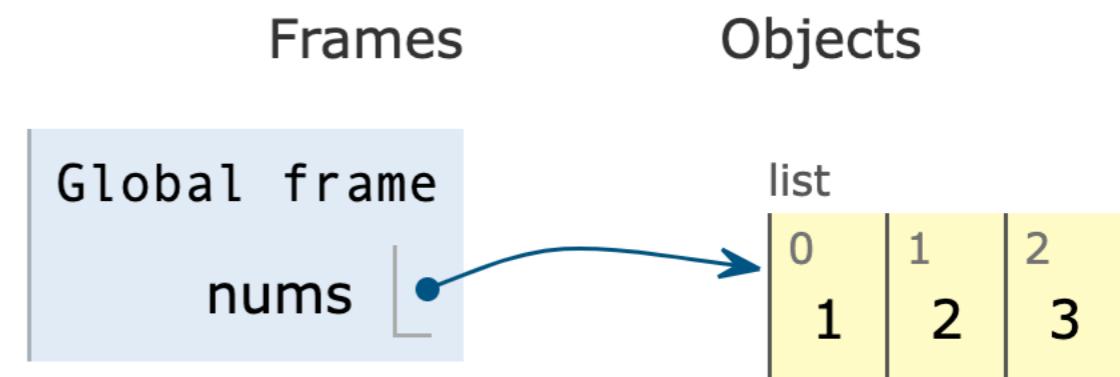
- Why C/C++: performance
 - how code runs
 - cachelines
 - garbage collection
 - safety checks
- Why C++ (over C): language features

Demos

Background: Memory Management

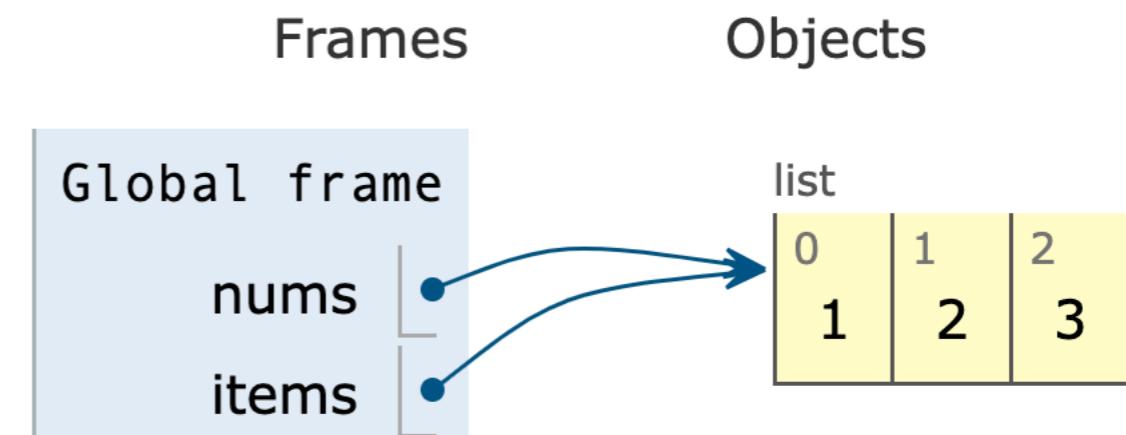
<https://pythontutor.com/>

```
→ 1 nums = [1,2,3]
→ 2 items = nums
  3 nums = [4,5,6]
  4 items = None
```



Background: Memory Management

```
1  nums  = [1,2,3]
→ 2  items = nums
→ 3  nums  = [4,5,6]
4  items = None
```



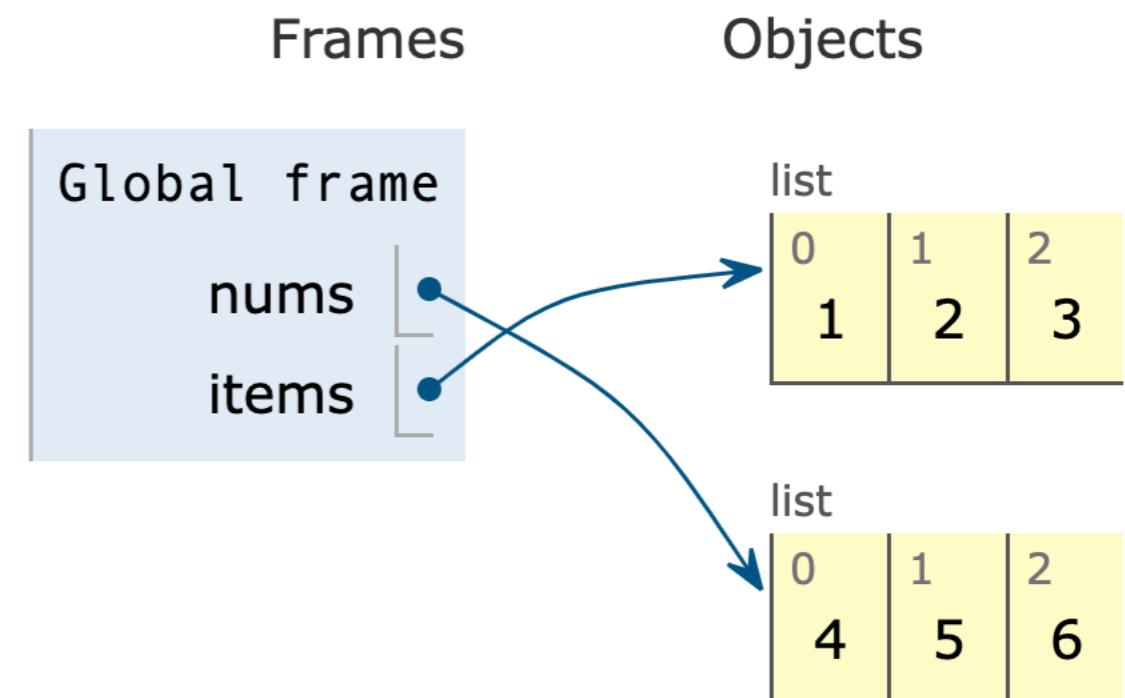
Background: Memory Management

```
1 nums = [1,2,3]
2 items = nums
3 → nums = [4,5,6]
4 → items = None
```

→ line that has just executed

→ next line to execute

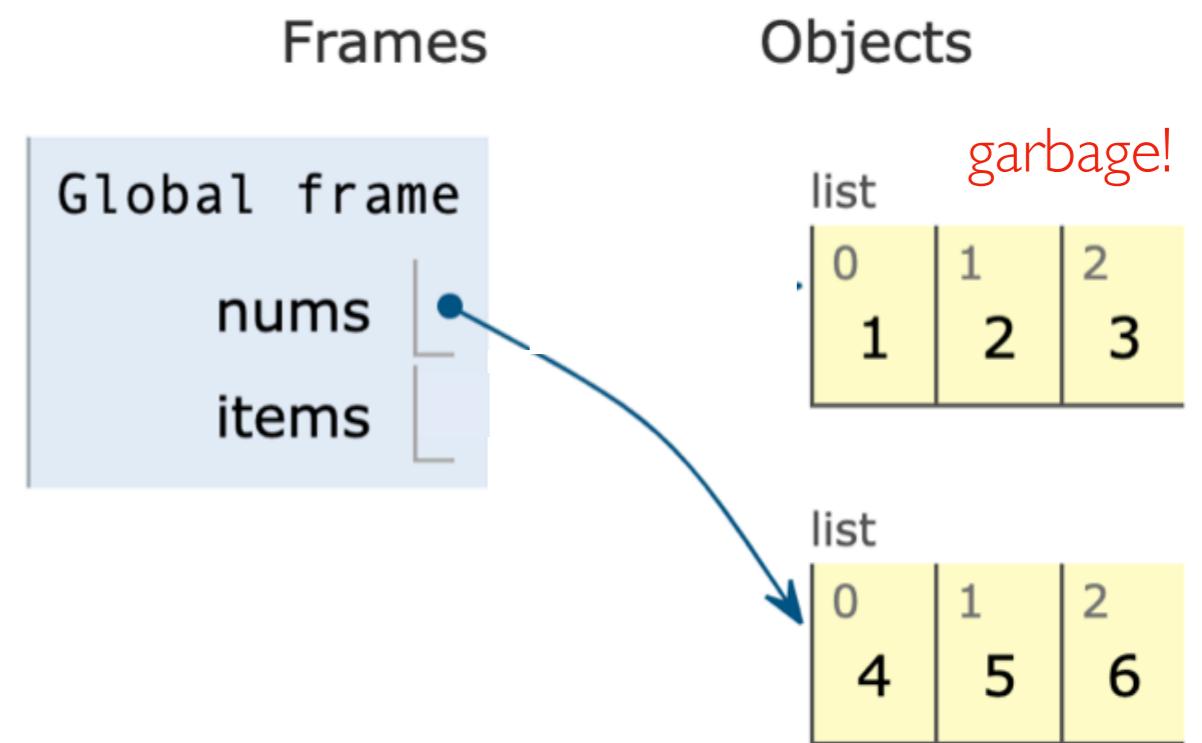
line that has just executed



Background: Memory Management

- data that can no longer be accessed in any way is "garbage"
- we can release garbage to free up memory
- in simple cases, the garbage objects might recognizable immediately
- in complicated cases (for example, circular references), a background garbage collection algorithm needs to run to identify garbage
- garbage collection is costly and generally involves pausing execution (perhaps for many seconds!)

```
1  nums = [1,2,3]
2  items = nums
3  nums = [4,5,6]
4  items = None
```



C/C++ Performance

Advantage 1: compiled languages are *usually* faster at runtime

- no overhead due to interpreter or language virtual machine
- cannot dynamically profile+optimize

Advantage 2: C/C++ gives us more control over memory layout

- can design cache-friendly data structures

Advantage 3: C/C++ lets us manage memory allocation/deallocation manually

- YOU (the programmer) write code to manually delete allocations
- memory is freed up sooner (don't need to wait for garbage collection)
- no overheads for GC; no long pauses during GC

Outline

Welcome

Logistics

Background and Motivation

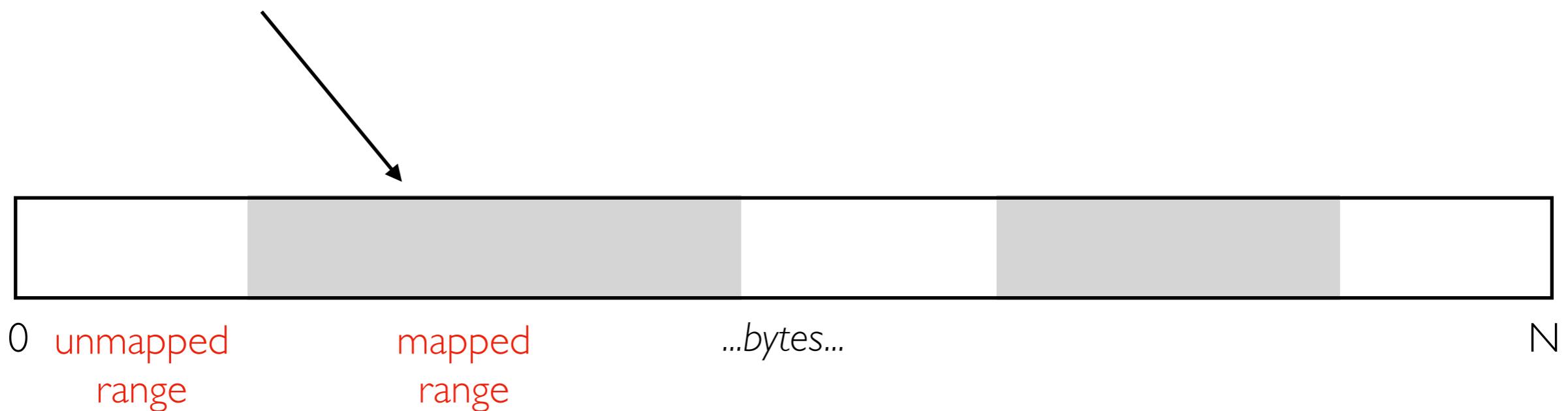
- Why C/C++: performance
 - how code runs
 - cachelines
 - garbage collection
 - safety checks
- Why C++ (over C): language features

Demos

Background: Safety Checks

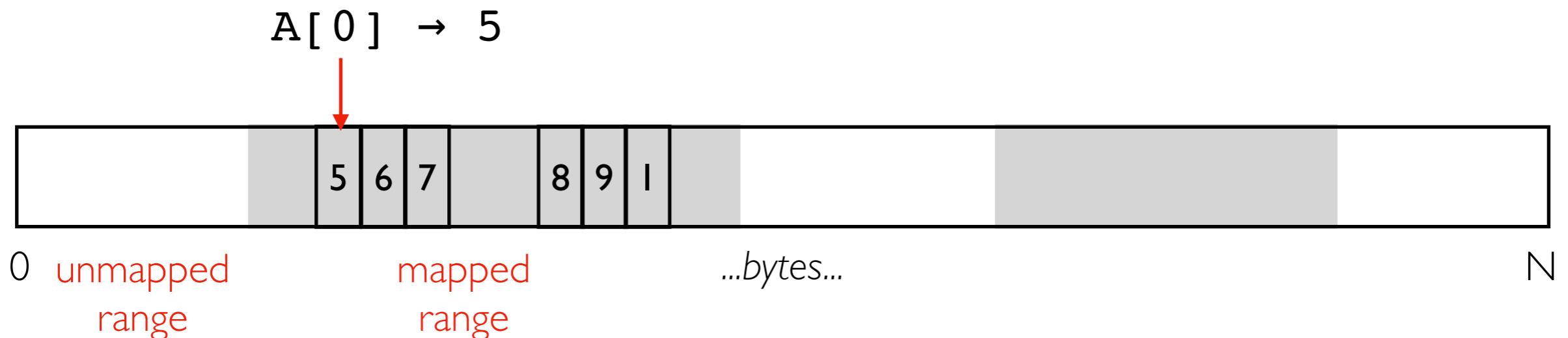
A = [5 , 6 , 7]
B = [8 , 9 , 1]

every process has an address space, which resembles a big array of bytes (indexes are called addresses). All the processes data lives somewhere in that address space.



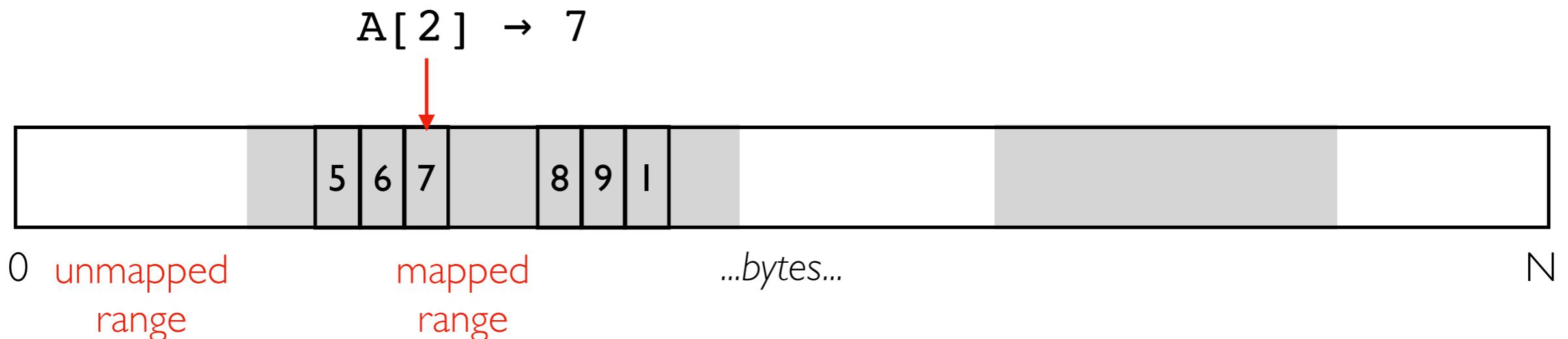
Background: Safety Checks

```
A = [ 5 , 6 , 7 ]  
B = [ 8 , 9 , 1 ]
```



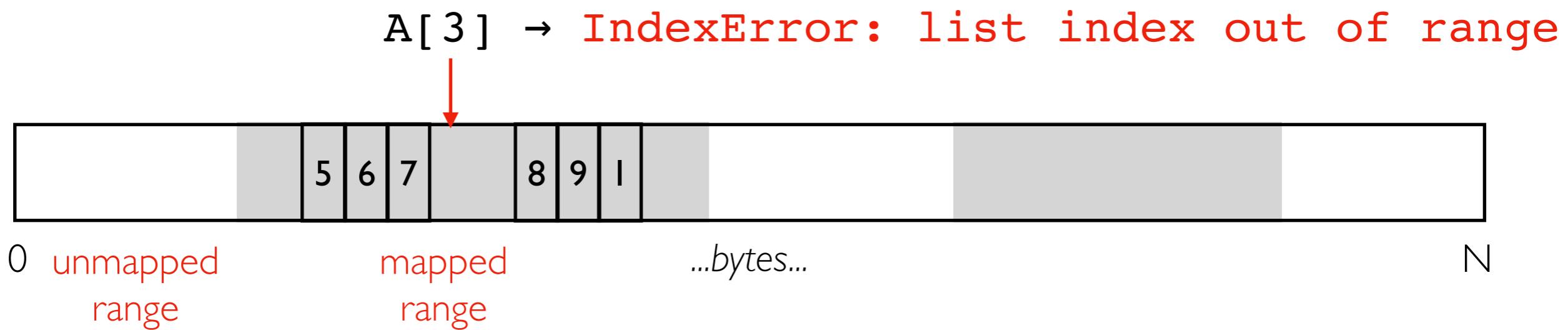
Background: Safety Checks

```
A = [ 5 , 6 , 7 ]  
B = [ 8 , 9 , 1 ]
```



Background: Safety Checks

```
A = [5, 6, 7]  
B = [8, 9, 1]
```



Many languages (Python, Java, etc) check bounds for you and raise an exception if you're outside. This checking has a performance cost, but is safer.

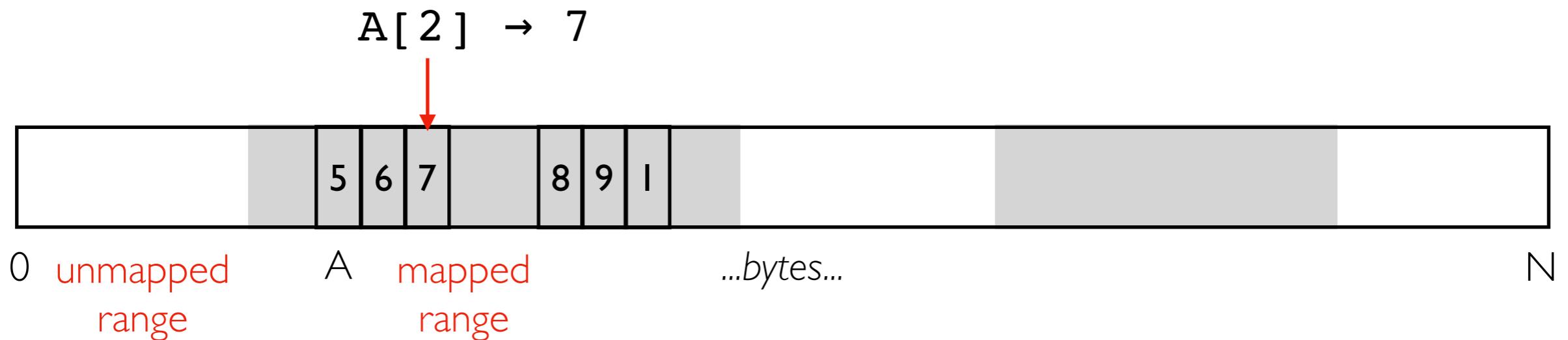
Bounds Checking

```
def print_at(items, idx):
    if 0 <= idx < len(items):
        print(items[idx]) ← Python checks that idx is in range, which is
    else:                                wasteful because your code already did that!
        print("bad index")
```

C/C++ Approach

Trust programmer to write code that checks bounds.

Generally don't spend time on double checking that!

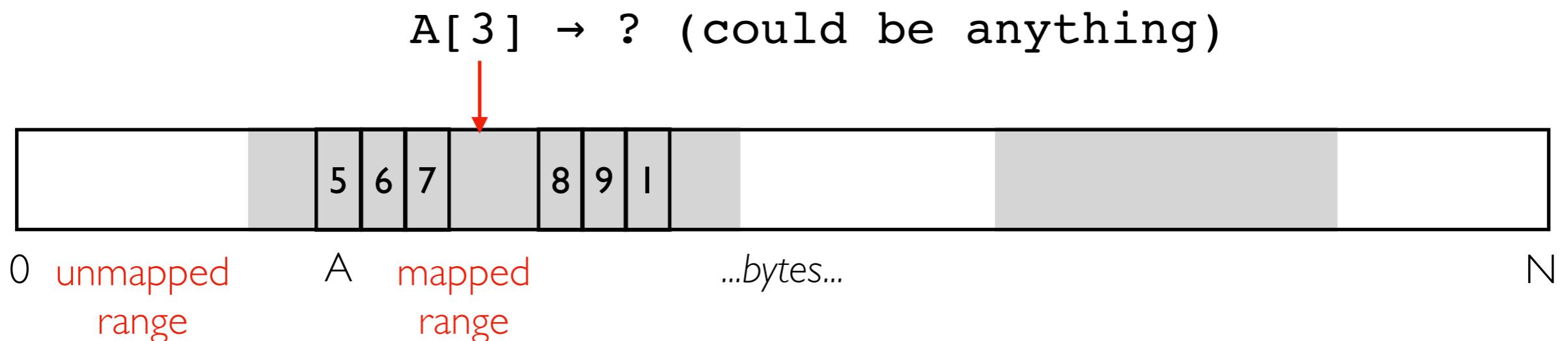


When programmer makes a mistake, however, there are a variety of strange things that could happen....

C/C++ Approach

Trust programmer to write code that checks bounds.

Generally don't spend time on double checking that!

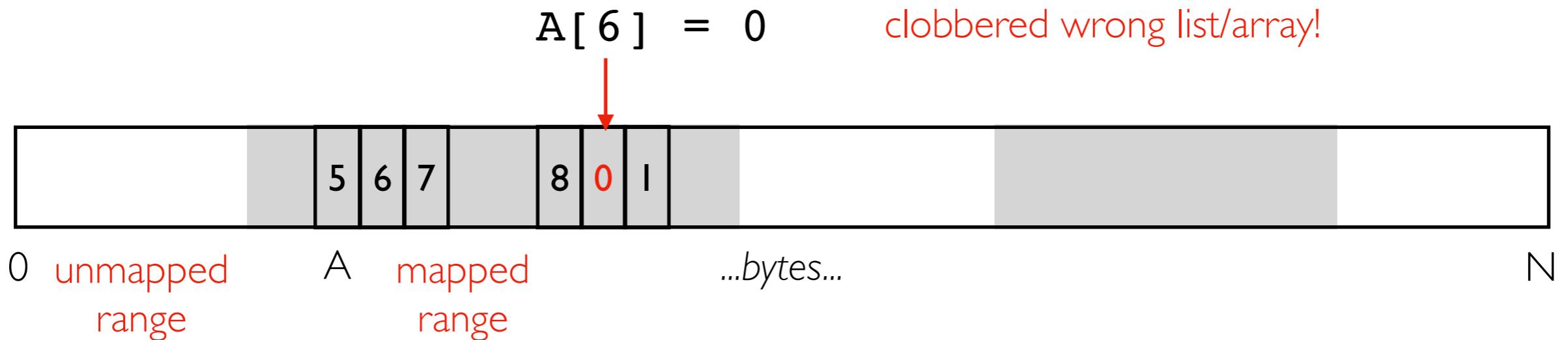


When programmer makes a mistake, however, are a variety of strange things that could happen...

C/C++ Approach

Trust programmer to write code that checks bounds.

Generally don't spend time on double checking that!

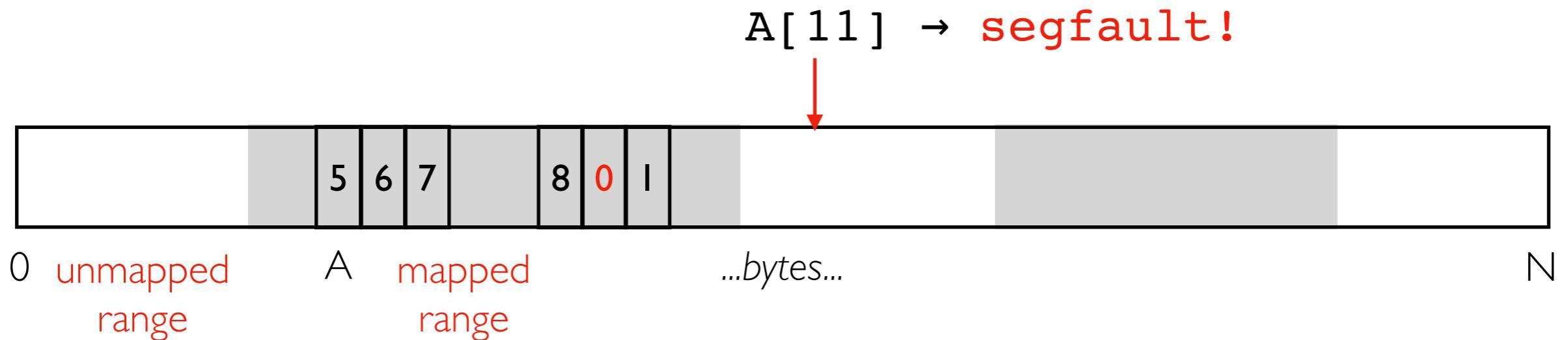


When programmer makes a mistake, however, are a variety of strange things that could happen...

C/C++ Approach

Trust programmer to write code that checks bounds.

Generally don't spend time on double checking that!



When programmer makes a mistake, however, are a variety of strange things that could happen...

C/C++ Performance

Advantage 1: compiled languages are *usually* faster at runtime

- no overhead due to interpreter or language virtual machine
- cannot dynamically profile+optimize

Advantage 2: C/C++ gives us more control over memory layout

- can design cache-friendly data structures

Advantage 3: C/C++ lets us manage memory allocation/deallocation manually

- YOU (the programmer) write code to manually delete allocations
- memory is freed up sooner (don't need to wait for garbage collection)
- no overheads for GC; no long pauses during GC

Advantage 4: C/C++ doesn't spend much compute time to catch programming mistakes

- avoids duplicated checking effort
- runs a little faster

C/C++ Performance

Advantage 1: compiled languages are *usually* faster at runtime

- no overhead due to interpreter or language virtual machine
- cannot dynamically profile+optimize

Advantage 2: C/C++ gives us more control over memory layout

- can design cache-friendly data structures

Advantage 3: C/C++ lets us manage memory allocation/deallocation manually

- YOU (the programmer) manage memory
- memory is more difficult and introduce new kinds for bugs (leaks, segfaults, etc).
- no overhead

Observation: almost all these performance features make programming more difficult and introduce new kinds for bugs (leaks, segfaults, etc).

Note: there are many tools for calling from one language to another (Python to C, Java to C++, etc).

Advantage 4: C/C++ is fast

- avoids duplication
- runs a little slower

Suggestion: if 80% of execution time is spent on 20% of your code, consider writing the critical 20% in a fast language (like C++) and the rest in an "easy" language (like Python)

Outline

Welcome

Logistics

Background and Motivation

- Why C/C++: performance
- Why C++ (over C): language features

Demos

A Few Language Features in C++ but not C

Function overloading

- multiple functions with the same name that accept different types

Type deduction

- use "auto" type (or other features) to let C++ decide what the type should be
- templating: don't need many different similar functions to handle different types

Alternatives to pointers

- references, smart pointers (for example, unique and shared)

OOP (Object Oriented Programming)

- classes, inheritance (multiple!), public/private/etc.

Resource management with RAII (Resource Acquisition is Initialization)

- use destructors to make sure resources are freed when necessary
- differentiate copy/move, manage ownership of objects over resources

Rich STL (Standard Library)

- containers, iterators, algorithms

Functional programming

- anonymous lambda functions
- many standard library functions that take function references

Outline

Welcome

Logistics

Background and Motivation

- Why C/C++: performance
- Why C++ (over C): language features

Demos