Object Oriented Programming
Dr Robert Harle

IA CST, PPS (CS) and NST (CS)

Lent 2010/11

The OOP Course

- Last term you studied **functional** programming (ML)
- This term you are looking at **imperative** programming (Java primarily).
  - You already have a few weeks of Java experience
  - This course is hopefully going to let you separate the fundamental software design principles from Java's quirks and specifics

- Four Parts
  - From Functional to Imperative
  - Object-Oriented Concepts
  - The Java Platform
  - Design Patterns and OOP design examples

Java Practicals

- This course is meant to *complement* your practicals in Java
  - Some material appears only here
  - Some material appears only in the practicals
  - Some material appears in both: deliberately*!

* Some material may be repeated unintentionally. If so I will claim it was deliberate.

Books and Resources I

- OOP Concepts
  - Look for books for those learning to first program in an OOP language (Java, C++, Python)
  - *Java: How to Program* by Deitel & Deitel (also C++)
  - *Thinking in Java* by Eckels
  - *Java in a Nutshell* (O' Reilly) if you already know another OOP language
  - Lots of good resources on the web

- Design Patterns
  - *Design Patterns* by Gamma et al.
  - Lots of good resources on the web

Books and Resources II

- Also check the course web page
  - Updated notes (with annotations where possible)
  - Code from the lectures
  - Sample tripos questions

http://www.cl.cam.ac.uk/teaching/1011/OOProg/

From Functional to Imperative Programming

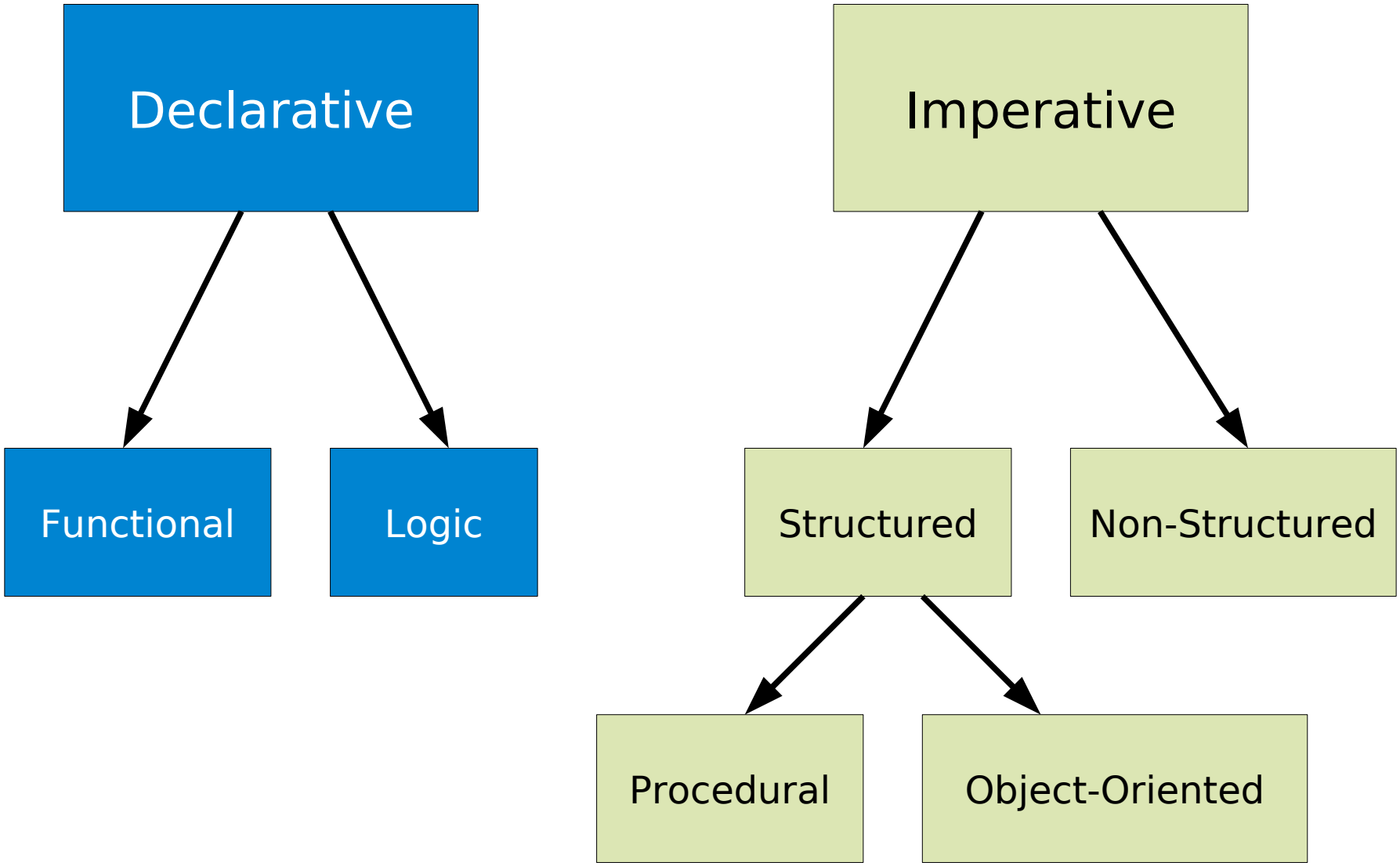
What can Computers Do? I

- The computability problem
  - Given infinite computing 'power' what can we do? How do we do it? What can't we do?
  - Option 1: Forget any notion of a physical machine and do it all in maths
    - Leads to an abstract mathematical programming approach that uses functions
    - Gets us Declarative/Functional Programming (e.g. ML)
  - Option 2: Build a computer and extrapolate what it can do from how it works
    - Not so abstract. Now the programming language links closely to the hardware
    - This leads naturally to imperative programming (and on to object-oriented)

What can Computers Do? II

- The computability problem
  - Both very different (and valid) approaches to understanding computer and computers
    - Turns out that they are equivalent
    - Useful for the functional programmers since if it didn't, you couldn't put functional programs on real machines...

Some Programming Paradigms



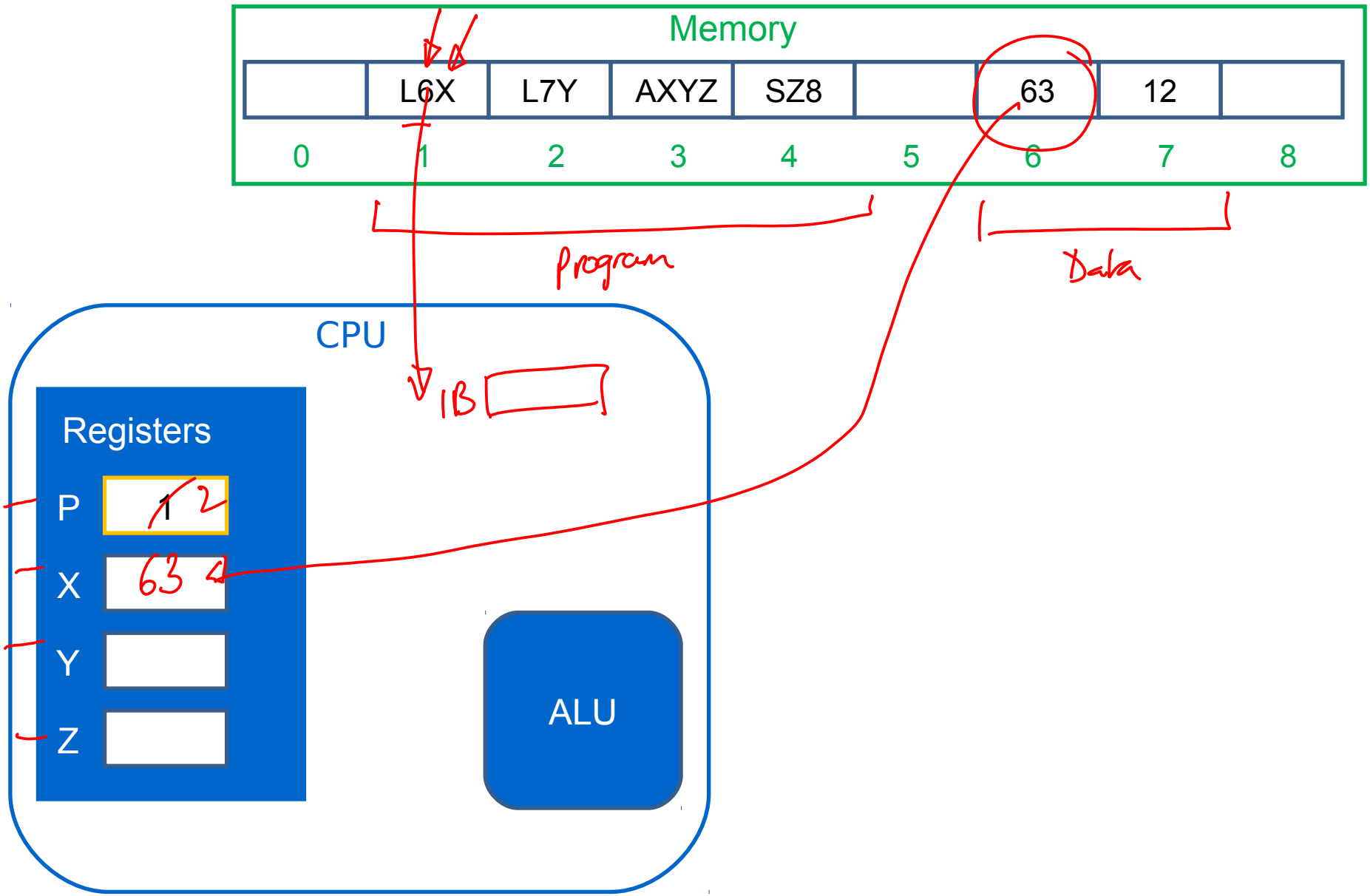
Key Declarative/Imperative Differences

- Declarative programs do not have state
- Declarative programs have functions whilst imperative programs have procedures
- Imperative programs require you to explicitly specify the type of every variable
- Declarative languages typically rely on recursion whilst imperative languages can also use control flow techniques such as while, for, etc.

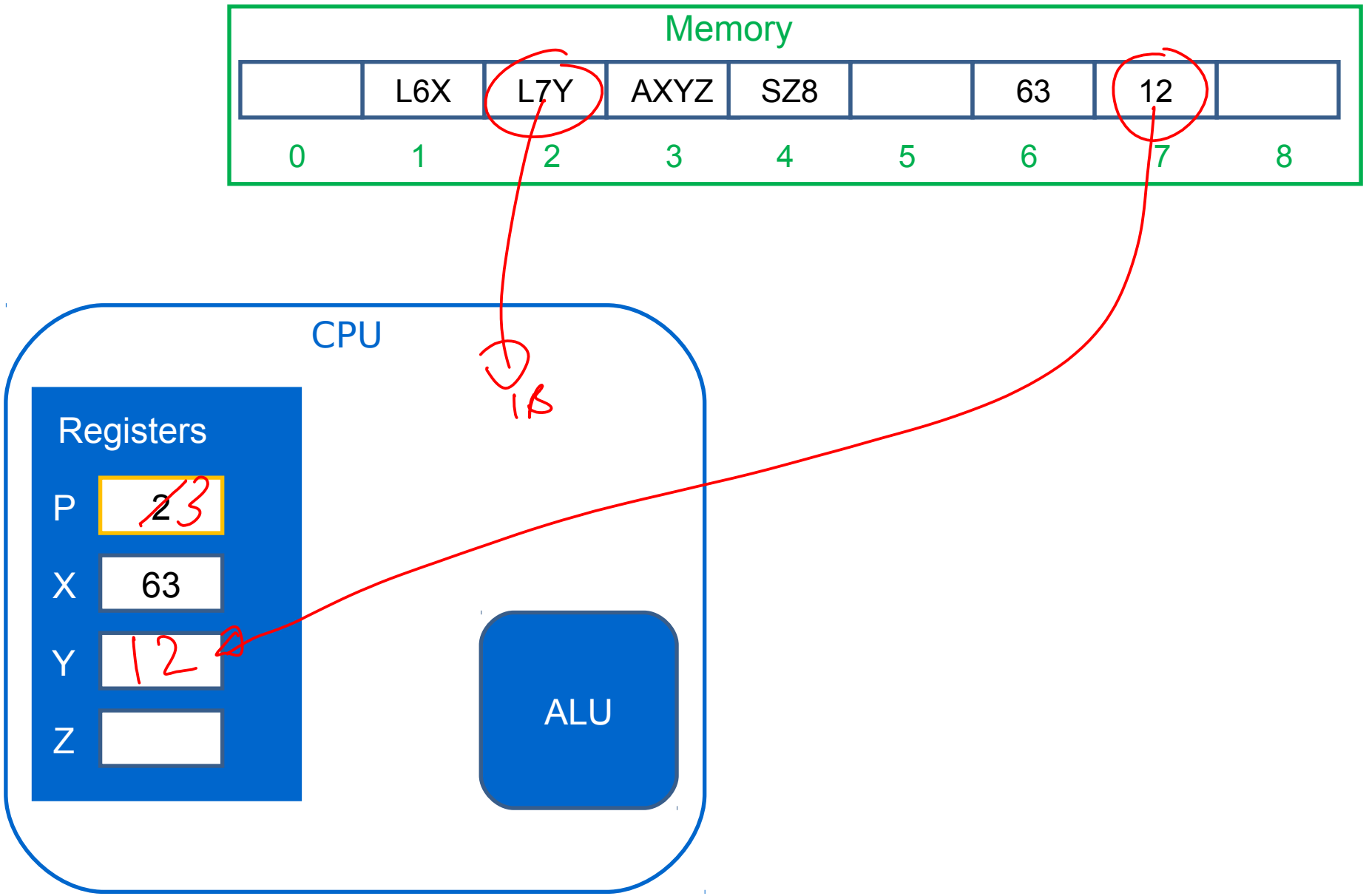
Thinking Imperatively

- Most people find imperative more natural, but each has its own strengths and weaknesses
- Because imperative is a bit closer to the hardware, it does help to have a good understanding of the basics of computers.
- It's worth reviewing a few points from the CF course last term to make sure we're all up to speed...

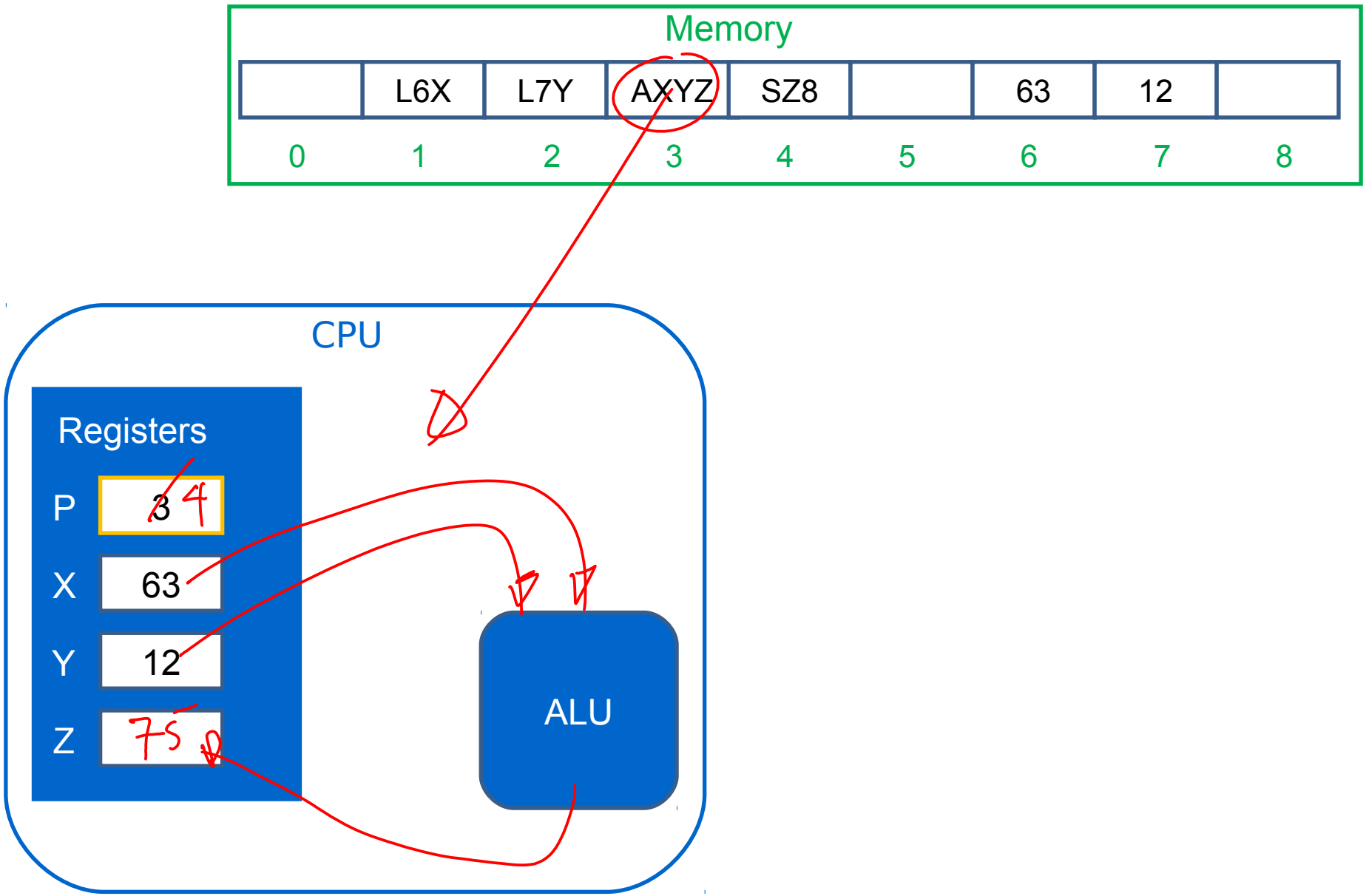
Dumb Model of a Computer I



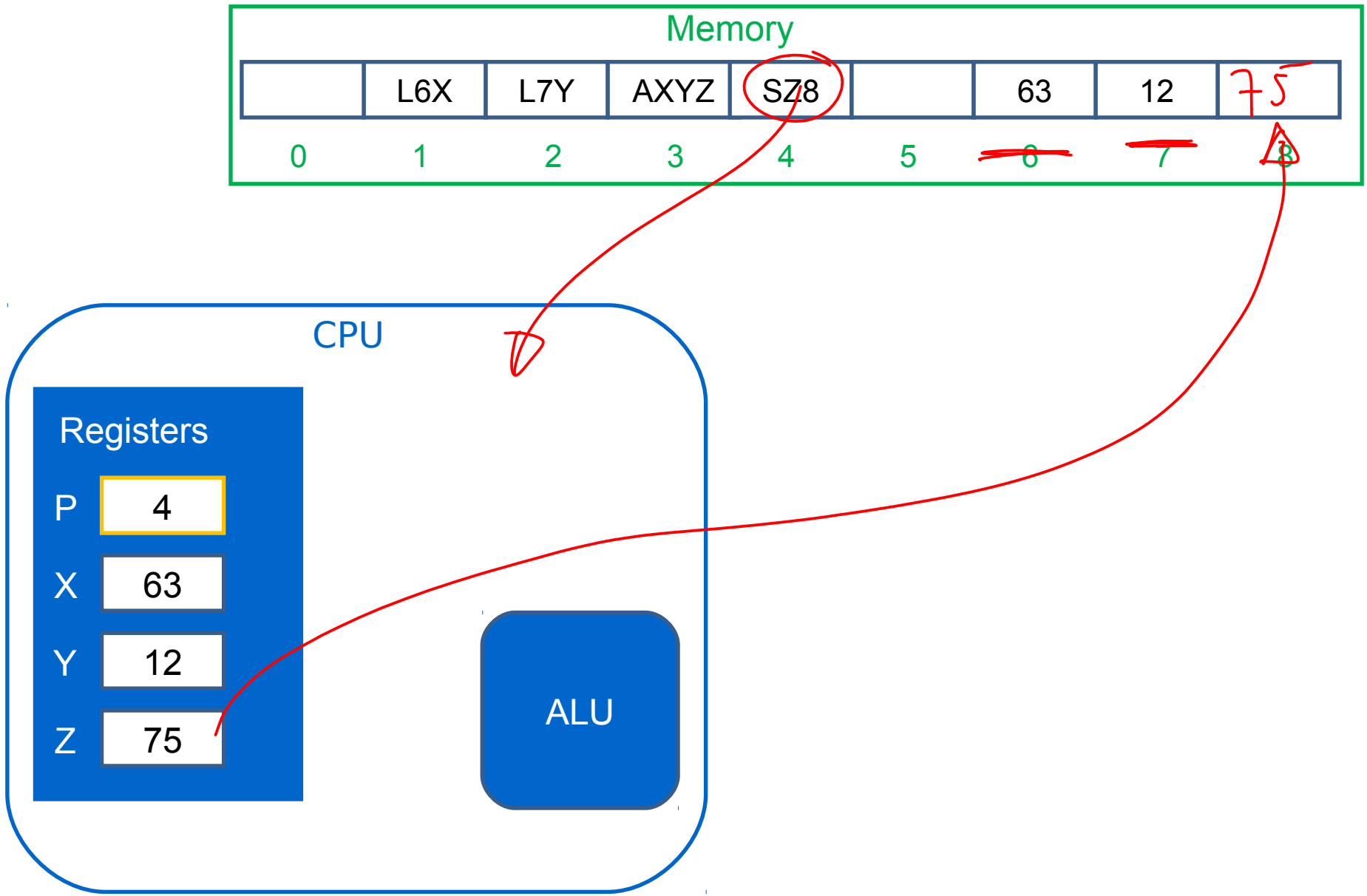
Dumb Model of a Computer II



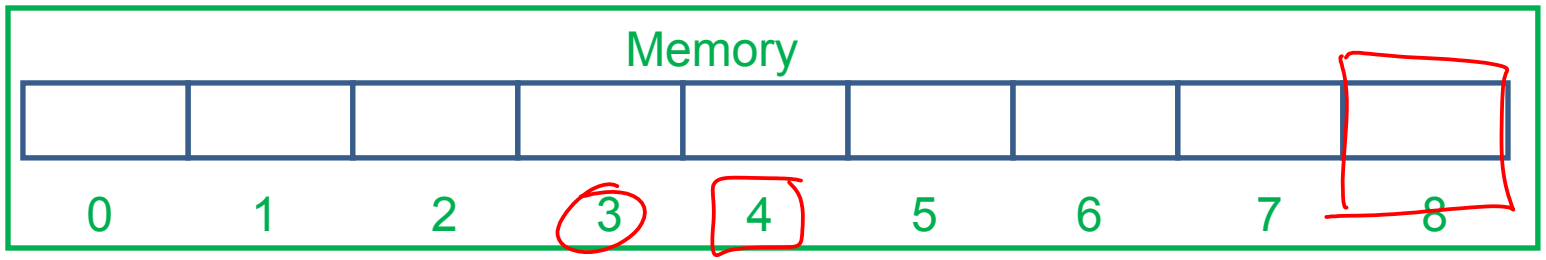
Dumb Model of a Computer III



Dumb Model of a Computer IV



System Memory

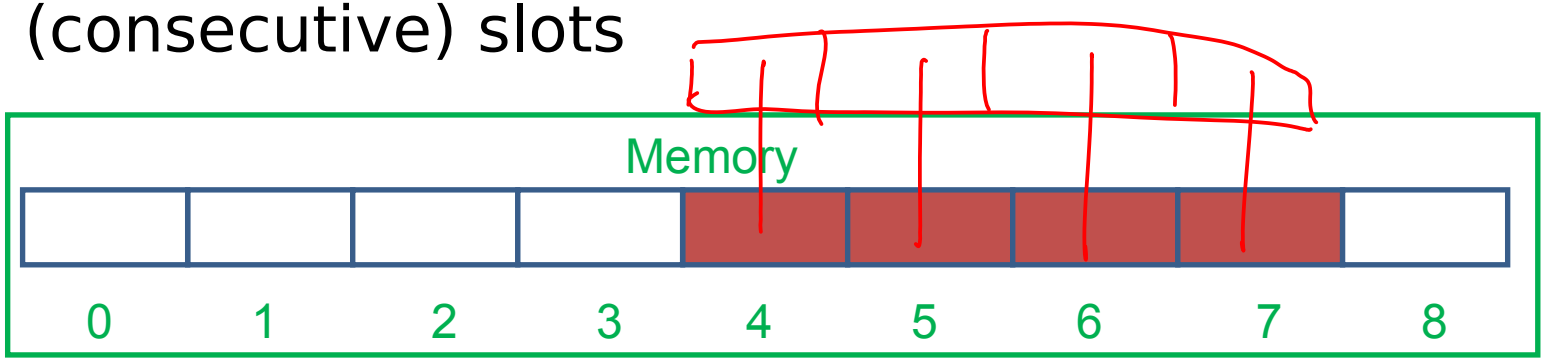


8 bits = 1 byte

- We model memory as a series of slots
  - Each slot has a set size (1 byte or 8 bits)
  - Each slot has a unique *address*
- Each address is a set length of n bits
  - Mostly n=32 or n=64 in today's world
  - Because of this there is obviously a maximum number of addresses available for any given system, which means a maximum amount of installable memory

Big Numbers

- So what happens if we can't fit the data into 8 bits e.g. the number 512?
- We end up distributing the data across (consecutive) slots

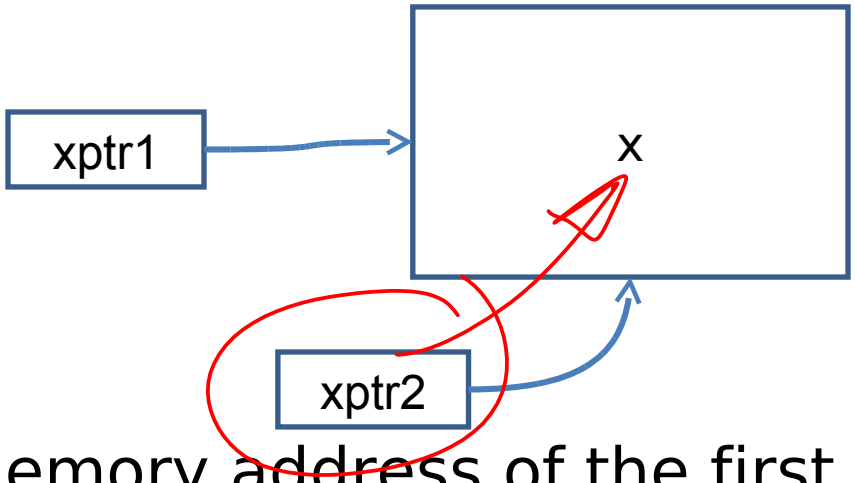


- Now, if we want to act on the number as a whole, we have to process each slot individually and then combine the result
- Perfectly possible, but who wants to do that every time you need an operation?

Pointers

- In many imperative languages we have variables that hold memory addresses.
- These are called *pointers*

```
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```

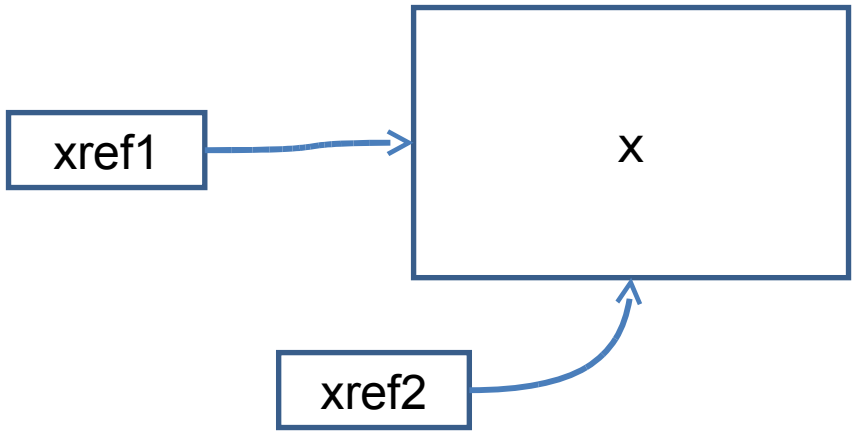


- A pointer is just the memory address of the first memory slot used by the object
- The pointer type tells the compiler how many slots the whole object uses

Example...

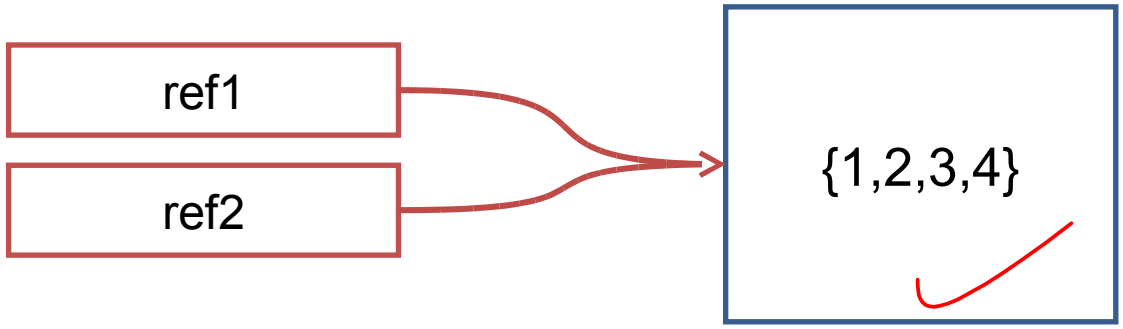
References

- The danger with pointers is that you can just randomly assign numbers to them (this can be very useful, but also dangerous as we've seen)
- Therefore many languages introduce a safer version of pointers: references
  - References **always** point to a valid place in memory or are explicitly NULL
  - You can't perform pointer arithmetic on them



References Example (Java)

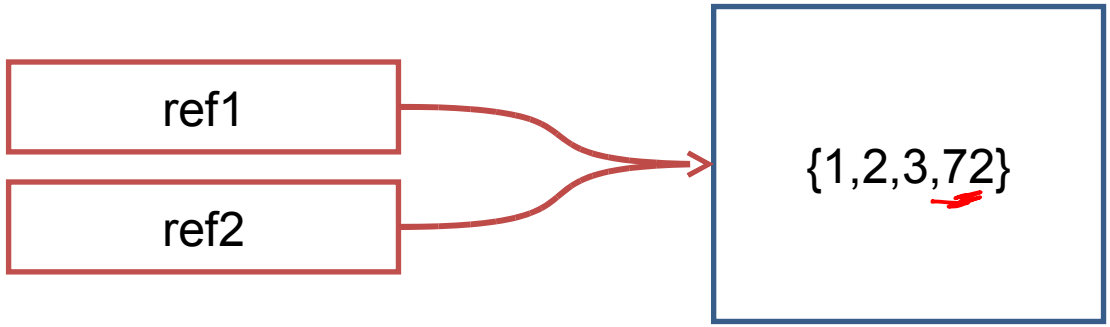
```
int[] ref1 = {1,2,3,4};
int[] ref2 = ref1;
```



int ar[] = _____

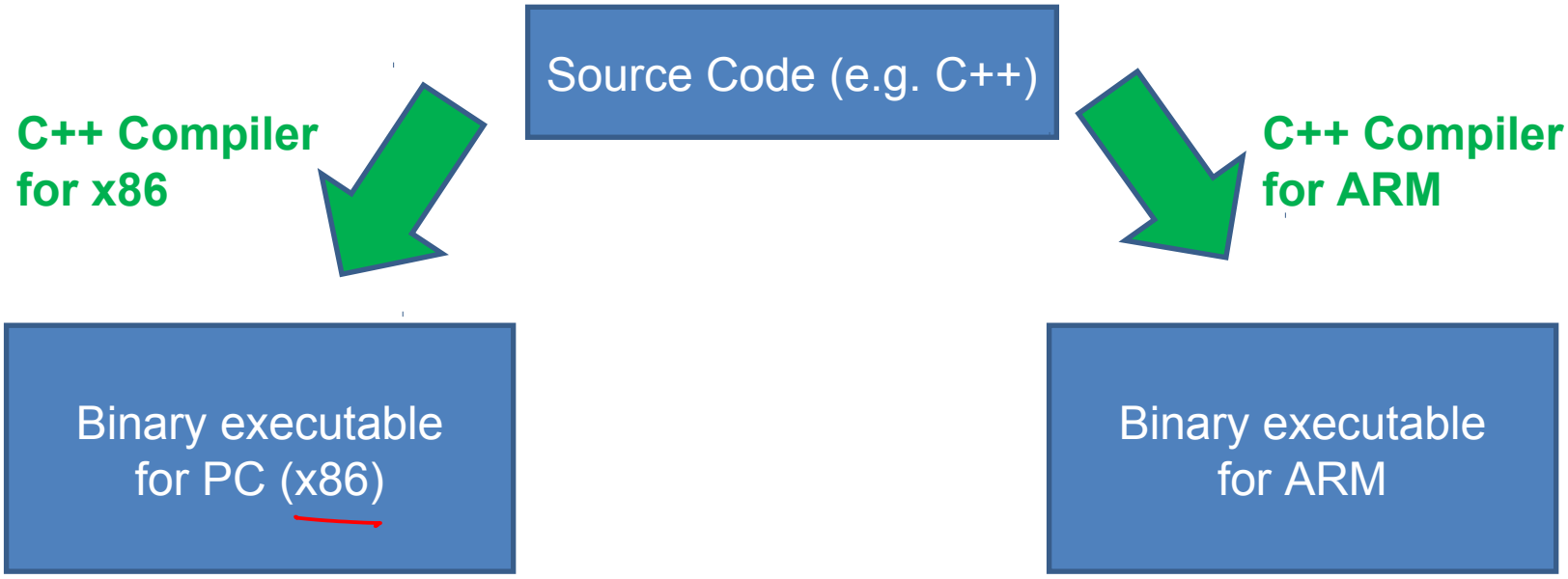
int[] ar = _____

```
ref2[3]=72;
```



Dealing with Machine Architectures

- Different CPUs have different instruction sets
- We write high level code
- We compile the code to a specific architecture



- Where was the compiled result when you were doing ML then?

Enter Java

- Sun Microcomputers came up with a different solution
  - They conceived of a Virtual Machine – a sort of idealised computer.
  - You compile Java source code into a set of instructions for this Virtual Machine (“bytecode”)
  - Your real computer runs a program (the “Virtual machine” or VM) that can efficiently translate from bytecode to local machine code.
- Java is also a *Platform*
  - So, for example, creating a window is the same on any platform
  - The VM makes sure that a Java window looks the same on a Windows machine as a Linux machine.
- Sun sells this as “Write Once, Run Anywhere”