CS558
Programming Languages

Fall 2023
Lecture 3b

Andrew Tolmach
Portland State University

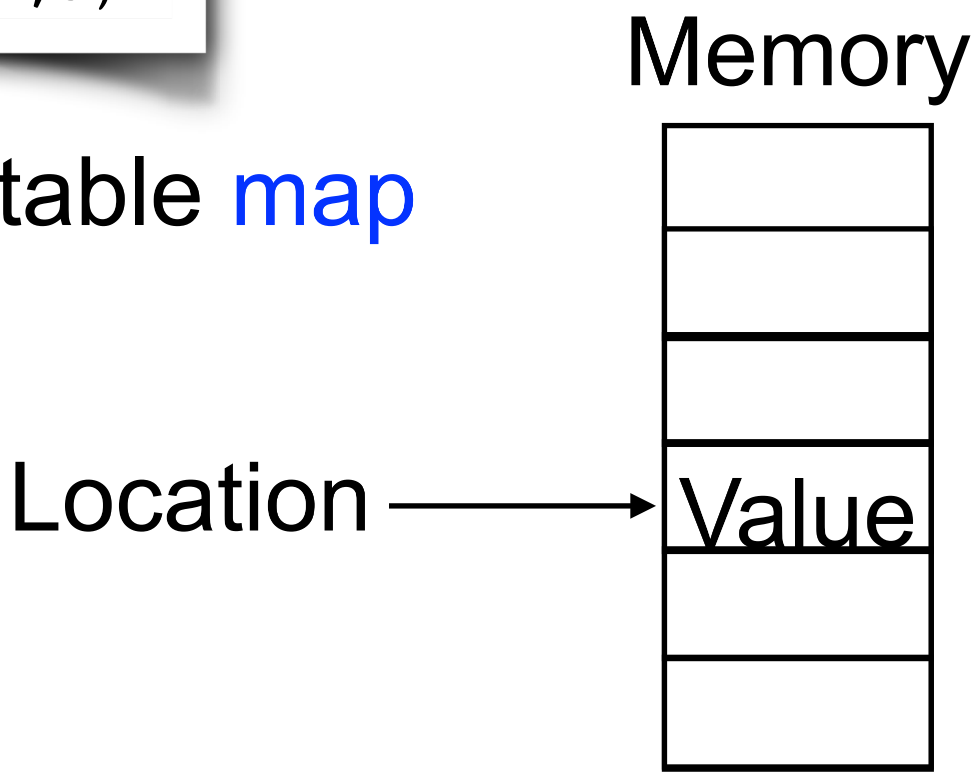
Describing the Store

```
int a = 42;
```

- Variable declarations often implicitly allocate storage
- In most languages, there are other ways to allocate storage too, such as explicit new operations or implicit boxing operations

```
new P(2,5)
```

- Simplistic store model: mutable map



Memory

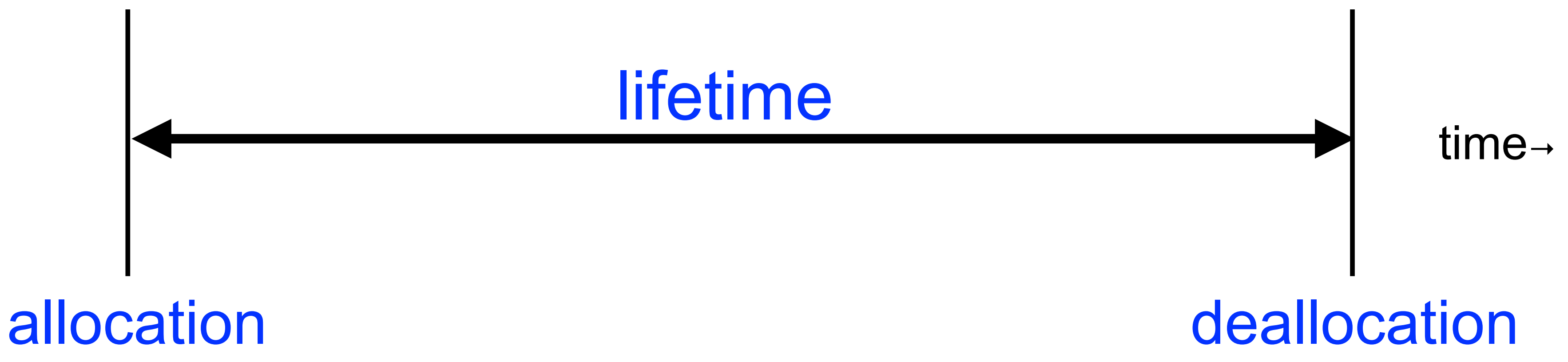
Location

Value

- Better models require distinguishing different classes of storage based on the lifetime of the data

Storage Lifetimes

- Typical computations use far more memory locations in total than they use at any one point
- So most language implementations support re-use of memory locations that are no longer needed



lifetime

time→

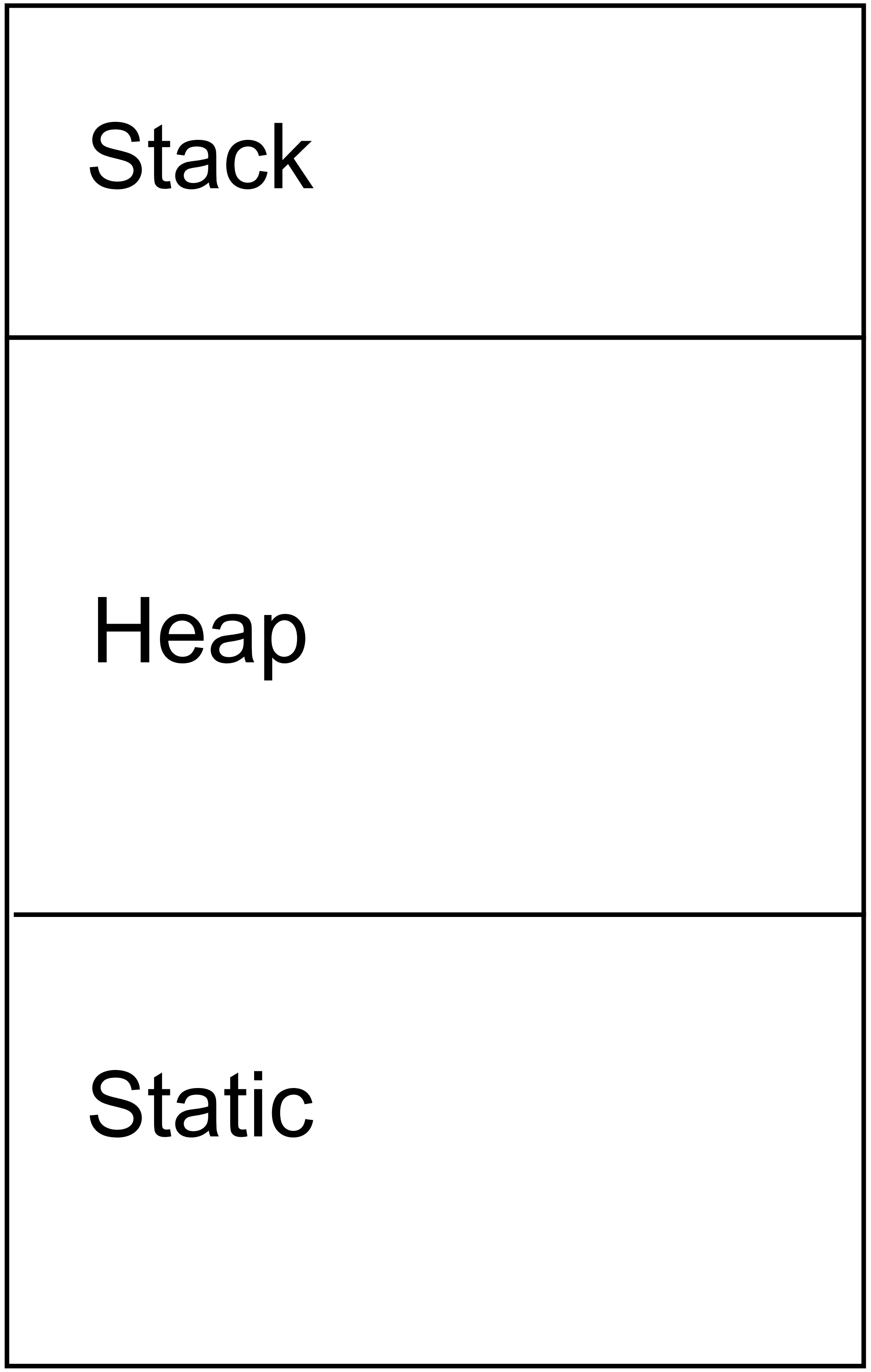
allocation

deallocation

- The lifetime of every object should cover all moments when the object is being used
  - Otherwise, we get a memory safety bug

Storage Classes

typical
process
memory
layout



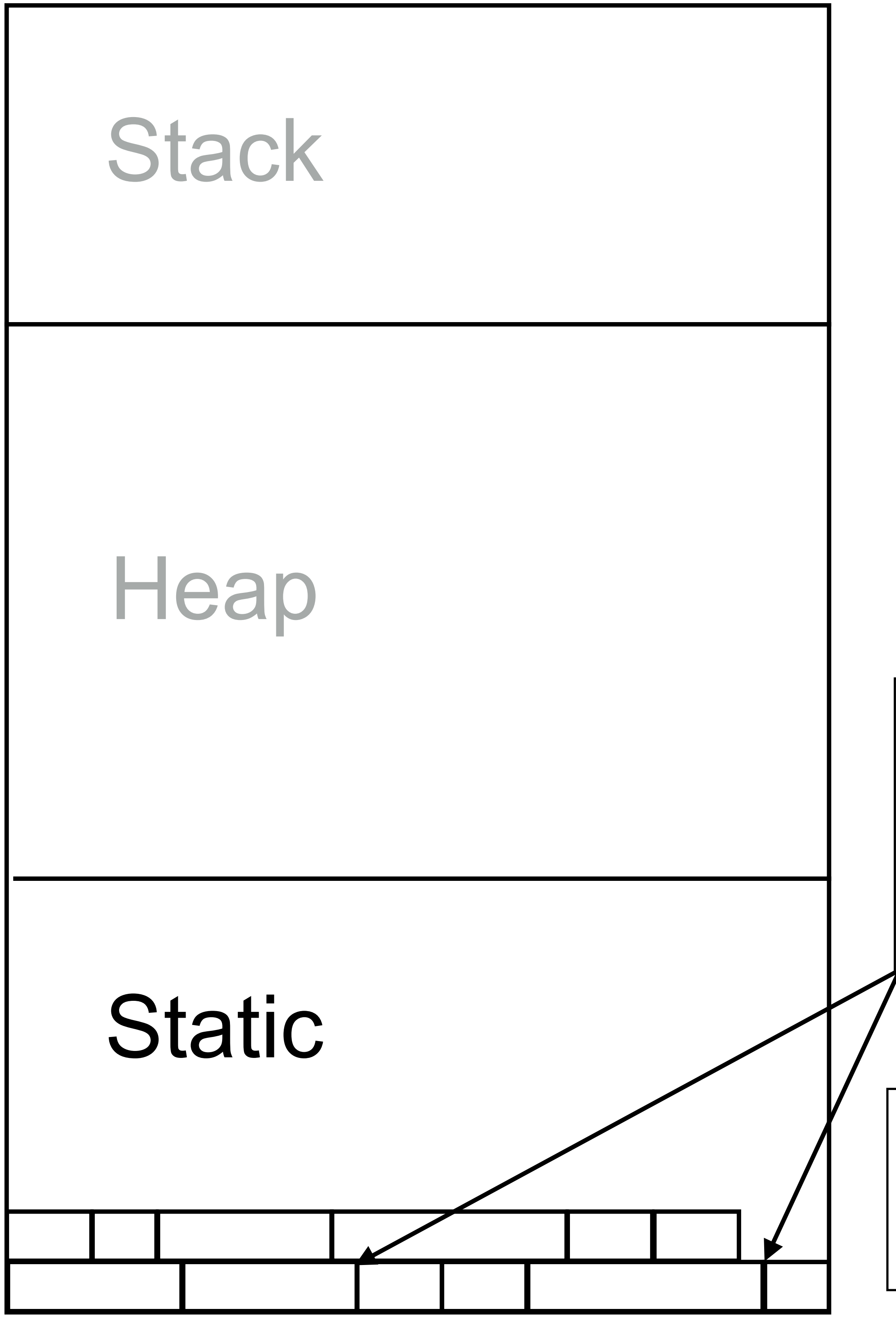
Stack

Heap

Static

Storage Classes: Static

typical
process
memory
layout



No runtime allocation/ deallocation cost

Usually holds global variables and constants

Fixed addresses known before execution starts

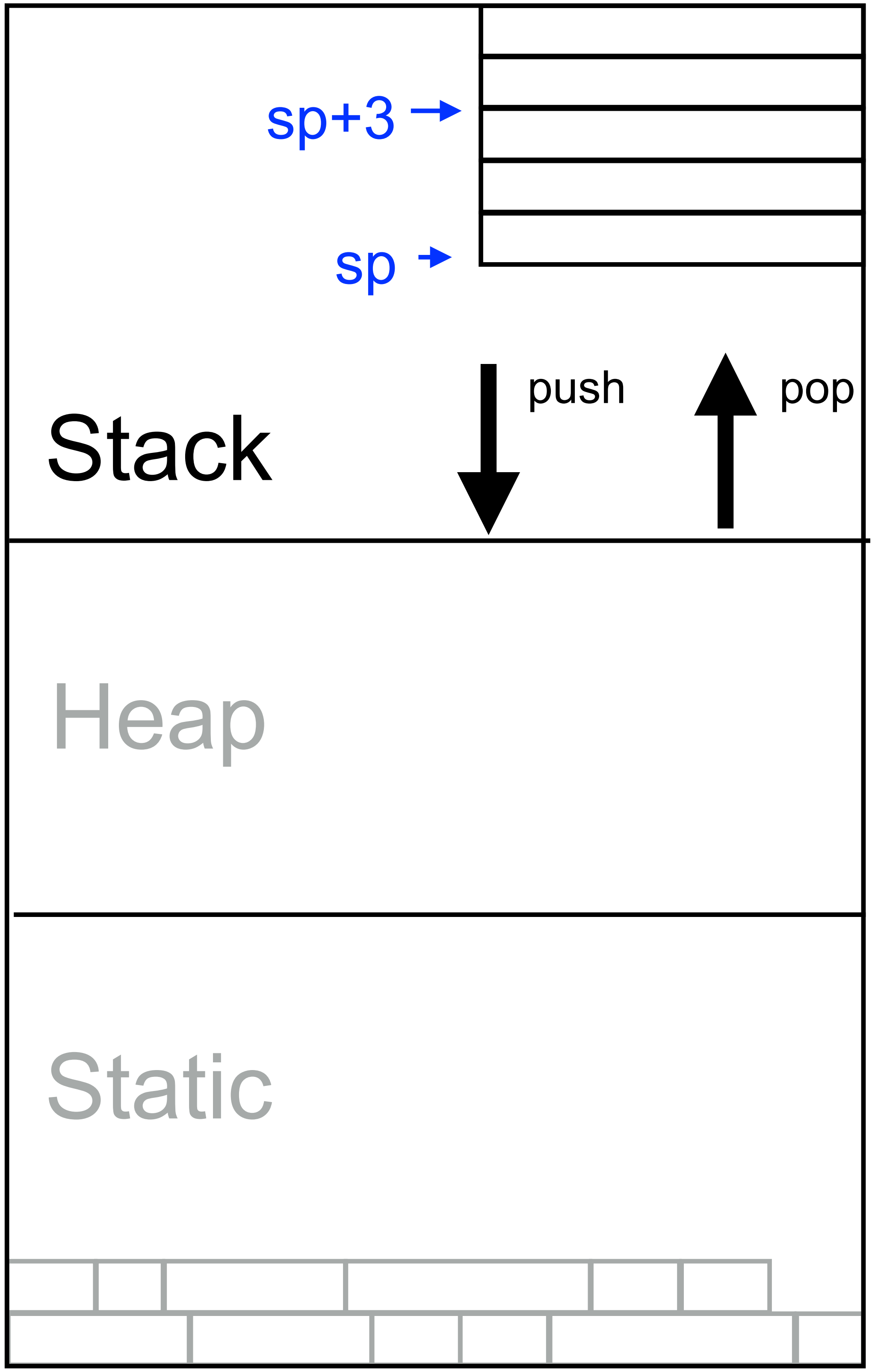
Lifetime = Entire Execution

Storage Classes: Stack

Usually holds function-local variables

(and internal control data, e.g. procedure return addresses)

Allocation/ deallocation is very cheap (just adjust sp)



typical process memory layout

Nested lifetimes: last allocated is first deallocated

Addresses are relative to top-of-stack pointer (sp)

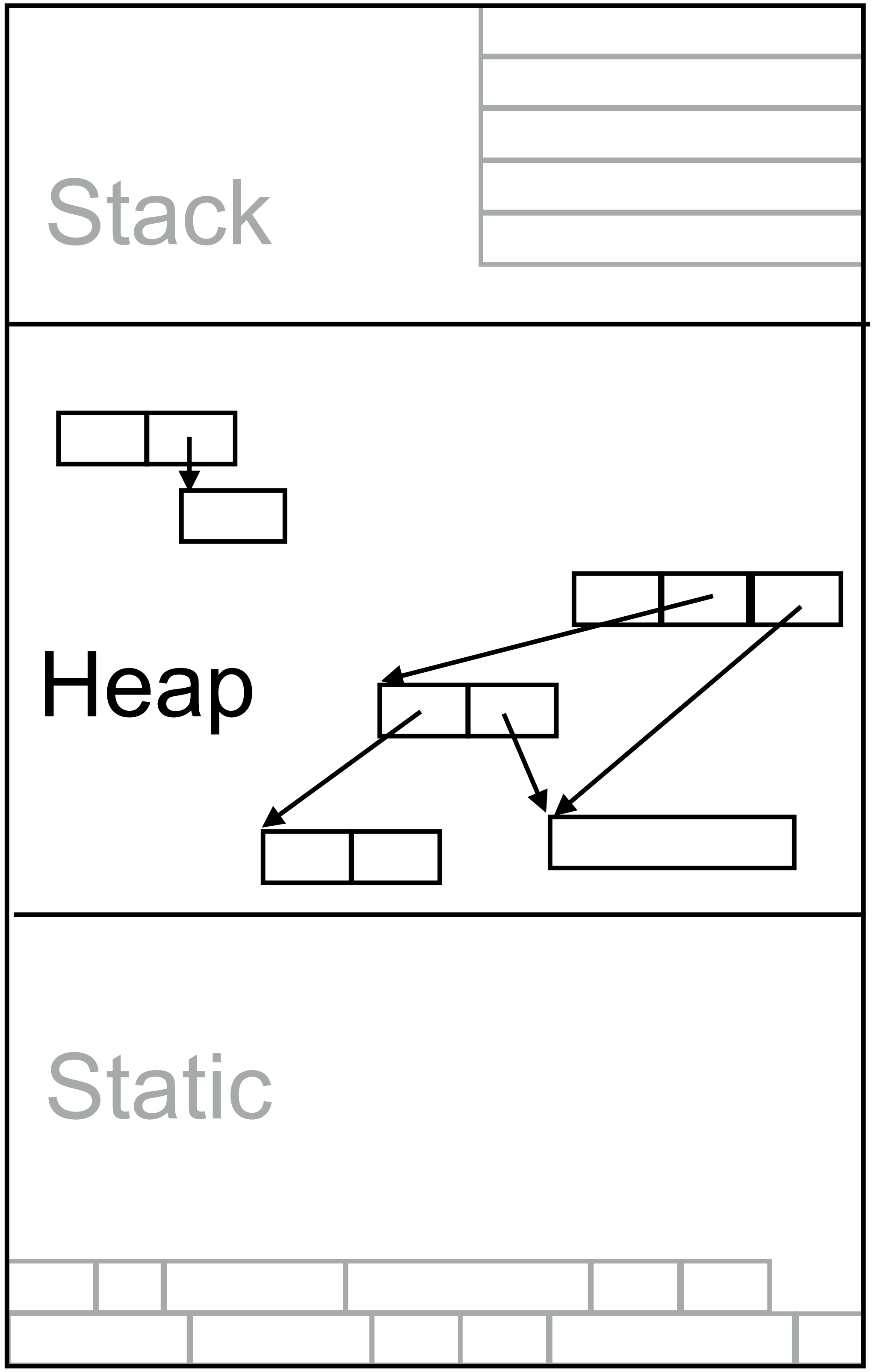
Good for cache and VM locality

Storage Classes: Heap

Arbitrary lifetimes

Used for explicitly allocated data

(and sometimes also implicitly allocated data, e.g. bignums, closures, etc.)



typical process memory layout

Allocation/ deallocation are relatively expensive

Done by runtime system code

Deallocation can be manual (risky) or done via garbage collection

Scope, Lifetime, Memory Safety

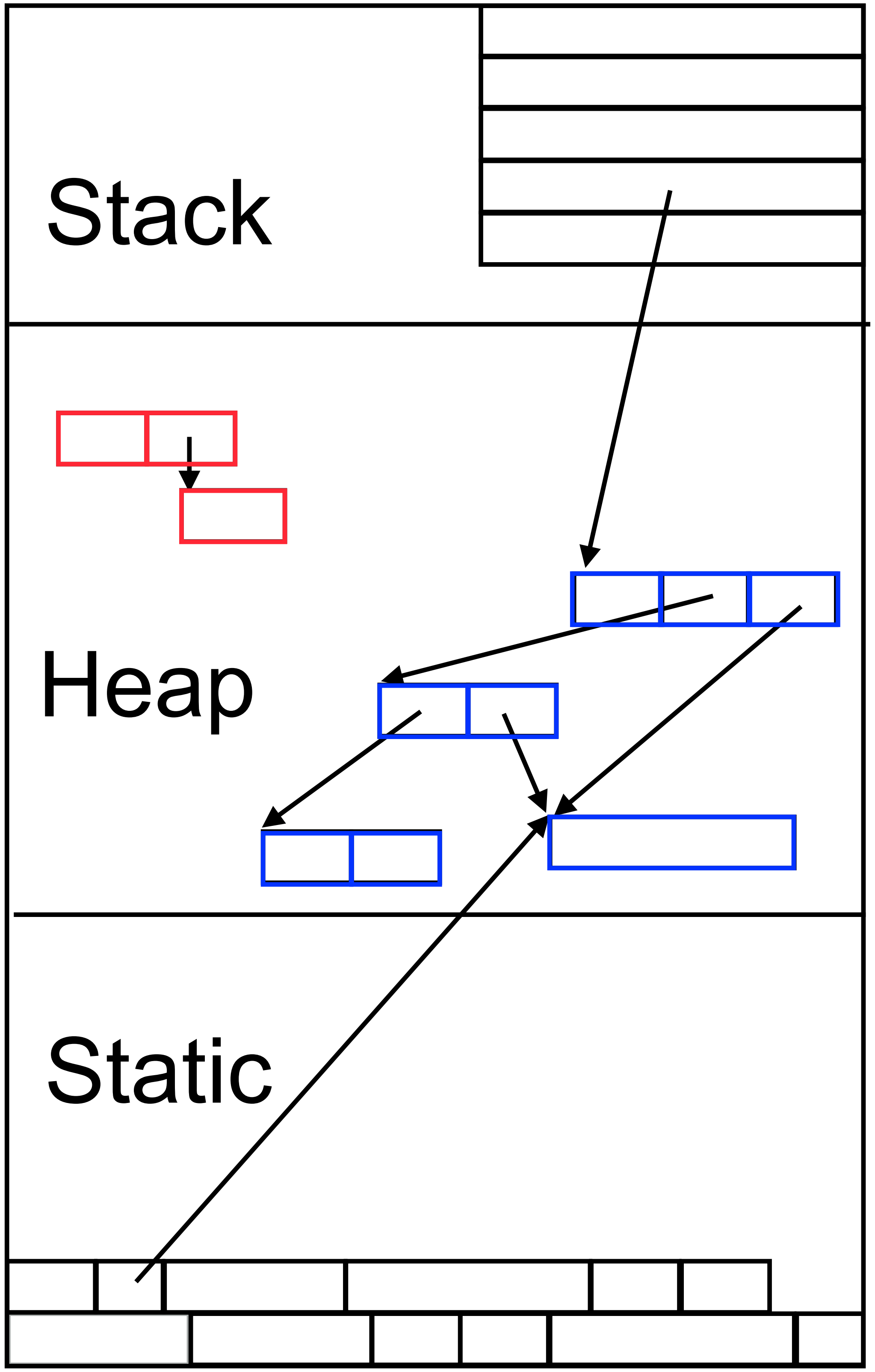
- Lifetime and scope are closely connected
- For a language to be memory safe, it suffices to make sure that in-scope identifiers never point (directly or indirectly) to deallocated objects
- For stack-allocated local variables, this happens naturally
  - Stack locations are deallocated only when function returns and its local variables go out of scope forever*

* Unless we have "first-class" functions...

Scope, Lifetime, Memory Safety

- For heap data, easiest to enforce safety using a garbage collector (GC)
  - GC typically works by recursively tracing all objects reachable from names that are currently in scope (or that might come back into scope later)
  - Only unreachable objects are deallocated, making their locations available for future re-allocation
  - (An alternative method is reference counting)
  - Of course, this takes time!

Tracing Garbage Collection



typical
process
memory
layout

Start by tracing pointers from roots in the stack and static areas

Any heap object reached by tracing is live

Recursively trace pointers between heap objects

When trace is done, any object that is not live is garbage

Its space can be reused for new allocations

Explicit Deallocation

- Many older languages (notably C/C++) support explicit deallocation of heap objects

- Somewhat more efficient than GC

```
char *foo() {
  char *p = malloc(100);
  free(p);
  return p;}
```

- But makes language unsafe: “dangling pointer” bug occurs if we deallocate an object that is still in use [unchecked runtime error]

- Converse problem: “space leak” bug occurs if we don’t deallocate an unneeded object.
  - Not a safety problem, but may unnecessarily make program run slower or crash with “out of memory” error

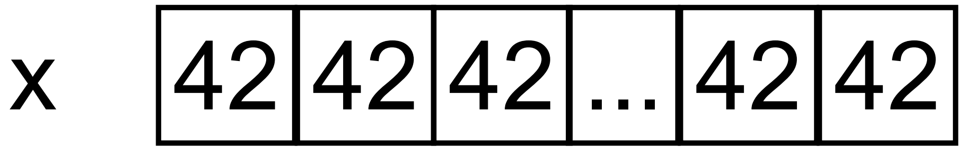
- Rust language supports safe explicit deallocation.

Pragmatics of Large Values

- Real machines are very efficient at handling word-size chunks of data (e.g. 16-64 bits depending on hardware). Things that fit easily in a word:
  - Numbers, characters, booleans, enumerations, class tags, etc.
  - Memory addresses (locations)
- Words are very easy to move, load, store, supply to operations, etc.
- But how can we manipulate larger chunks of data, such as records or arrays, which may occupy many words?

Boxing

- Two basic ways to represent large values
  - The unboxed representation holds the actual bits of the value, using as many machine words as needed

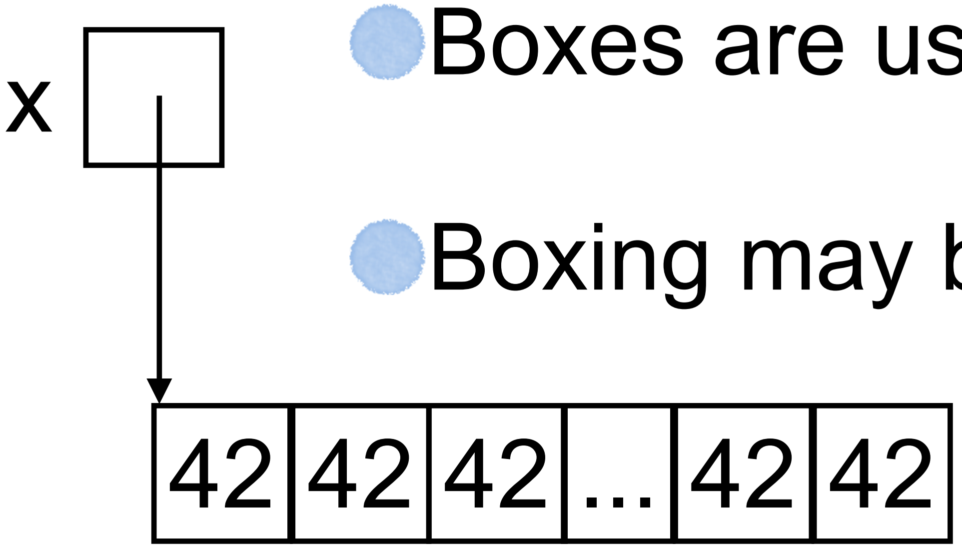


~textbook: “value” model

  - The boxed representation allocates separate storage (the “box”) for the actual bits, and then represents the value by the location of that storage

~textbook: “reference” model

    - Boxes are usually stored in the heap
    - Boxing may be performed implicitly or explicitly



Boxed vs. Unboxed

- Choice of representation can make a big difference to semantics on operations on the data
    - What does assignment mean?
    - What do equality comparisons mean?
    - How does parameter passing work?

Unboxed Assignment Semantics

- Early languages often used unboxed records and arrays

occupies
80x1 + 1x4
= 84 bytes

```
TYPE Employee =
RECORD
  name : ARRAY (1..80) OF CHAR;
  age : INTEGER;
END;
```

Pascal

- Semantics of assignment is to copy entire representation

```
VAR e1,e2 : Employee;
e1.age := 91;
e2 := e1;
e1.age := 19;
WRITE(e1.age, e2.age);
```

prints 19,91

Step-by-step

e1	fred	
e2	alice	

e1.age := 91

e1	fred	91
e2	alice	

e2 := e1

e1	fred	91
e2	fred	91

e1.age := 19

e1	fred	19
e2	fred	91

Unboxed representation issues

- This assignment semantics seems simple and appealing, but it has problems:
  - Assignment of a large value is expensive, since lots of words may need to be copied
  - Especially hard to generate efficient code if size of large value is not known statically

Boxed Assignment Semantics

- Most modern languages (e.g. Java, Python, Haskell) box all values (e.g. objects, records, constructions) that are larger than one word

- These languages naturally use reference semantics for assignment: just the pointer is copied, creating an alias

```
case class emp(var name:String, var Age:Int)
val e1 = emp("fred",91)
val e2 = e1
e1.age = 19
println(e1.age + " " + e2.age)
```

Scala

prints 19,19

Step-by-step

e2 = e1

e1.age = 19

Explicit Pointers

- Languages that use unboxed semantics may also have explicit pointer types to support reference-style operations

prints 19,19,91

```
struct Emp {
  char name[80];
  int age;
};
Emp *e1 = new Emp();
e1->age = 91;
Emp *e2 = e1;
Emp e3 = *e1;
e1->age = 19;
cout << e1->age << " " << e2->age
     << " " << e3.age << "\n";
```

C++

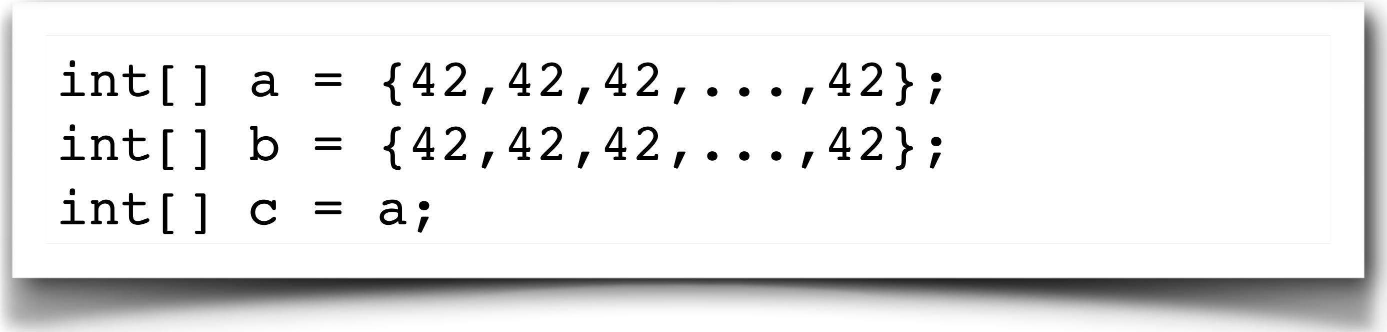
- In C/C++, `struct` and `class` instances are fundamentally unboxed, but programers usually box them explicitly (using `new` or `malloc`) and manipulate them via pointers

Varieties of Equality

- Languages typically provide some form of built-in equality testing on values. When are two (large) values equal?

- Under structural equality, values are equal when their contents are equal, bit for bit.

- Under reference equality, values are equal when their locations are identical.



```
int[] a = {42,42,42,...,42};
int[] b = {42,42,42,...,42};
int[] c = a;
```

Java

- Here `a,b,c` are all structurally equal, but only `a` and `c` are reference equal

- Reference equality ⇒ structural equality, but not vice-versa

Multiple kinds of equality

- Structural equality is only sane definition for unboxed values
- Reference equality may be cheaper to check than structural equality
- Some language provide both, under different names
  - They may also provide a standard way for programmer to define equality for a given type in an ad-hoc way
- E.g in Scala:
  - the `eq` operator gives reference equality
  - the == operator invokes a user-defined `equals` method
  - for case classes `equals` is pre-defined to be structural equality

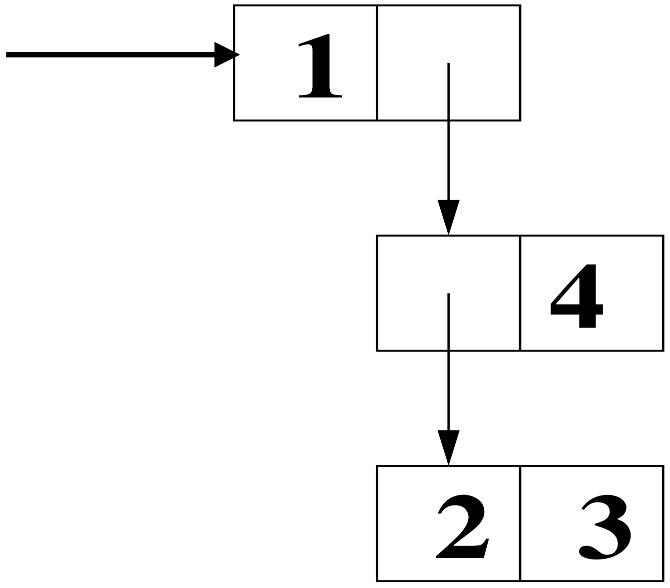
Pairs

- To study the essence of heap data structures, we can focus on a single new kind of value, the pair
  - Like a record with two fields, each containing another value
  - Written using “infix dot” notation

```
(1 . ((2 . 3) . 4))
```

- We can build larger records of a fixed size just by nesting pairs

corresponds to



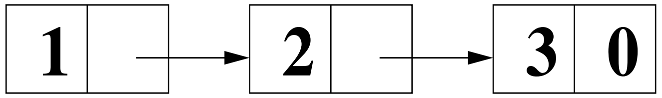
Lists

- We can also build all kinds of interesting arbitrary-sized recursive structures using pairs

- For example, to represent (singly-linked) lists we can use a pair for each node in the list.

    - First field contains an element; second field points to the next link, or is 0 to indicate end-of-list

    - Example: 1,2,3 `(1.(2.(3.0)))`



- Note that for programs to detect end-of-list, we need a test that distinguishes integers from pairs