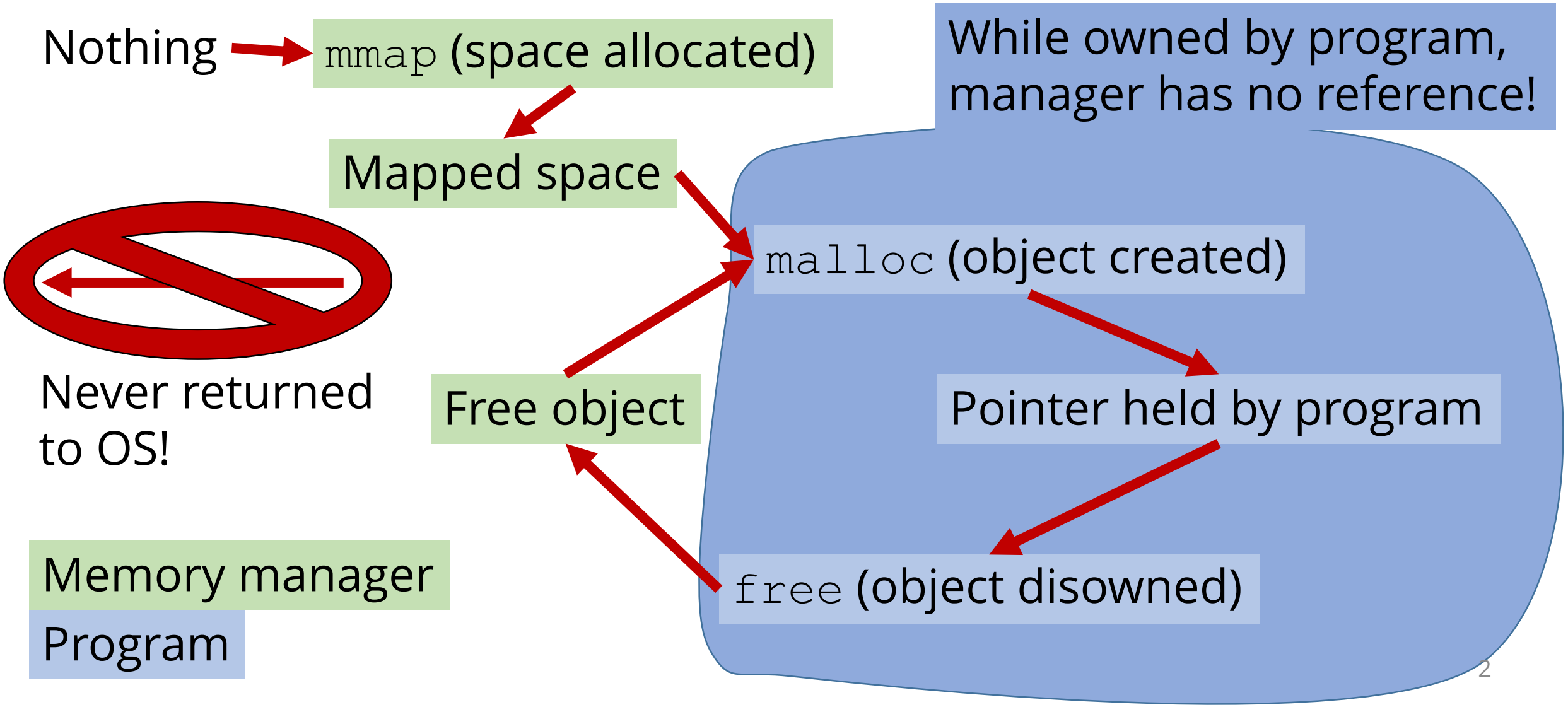
GC basics

Review


Nothing
mmap (space allocated)
Mapped space
While owned by program, manager has no reference!
malloc (object created)
Pointer held by program
free (object disowned)
Free object
Never returned to OS!
Memory manager
Program

Automatic memory management

- Defining principle: `free()` is automatic
- Common solution: Garbage collector
  - Part of the runtime does `free()` for you
- Other solutions exist (e.g. type-based)
- We focus on GC

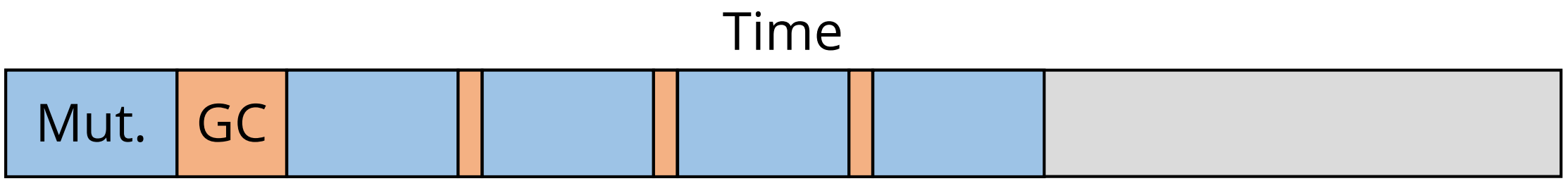
GC glossary

- Collector
- Mutator
- Heap vs. C heap
- Pool
- Root
- Reference
- Reachable
- Type information
- Stop-the-world
- Pause
- Parallel
- Concurrent

Performance

Many ways of measuring performance:

- Throughput
- Responsiveness
- Latency
- Resource utilization
- Fairness



Performance consideration

Manual memory management ain’t free!

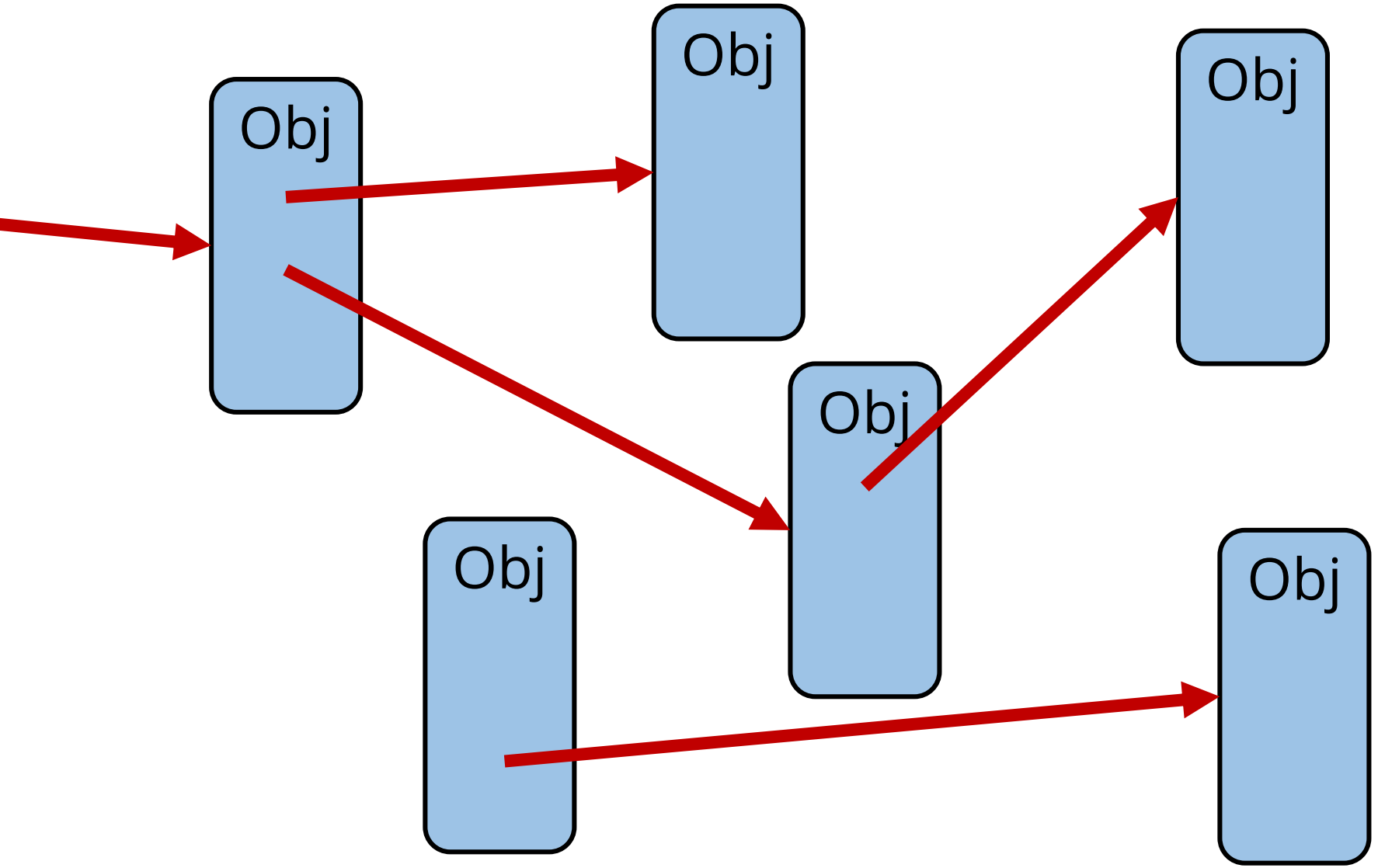
Performance

- GC technique affects performance
- Application affects performance
- Environment affects performance
- “5% improvement” is the GC mantra
  - (GC should take negative time by now)

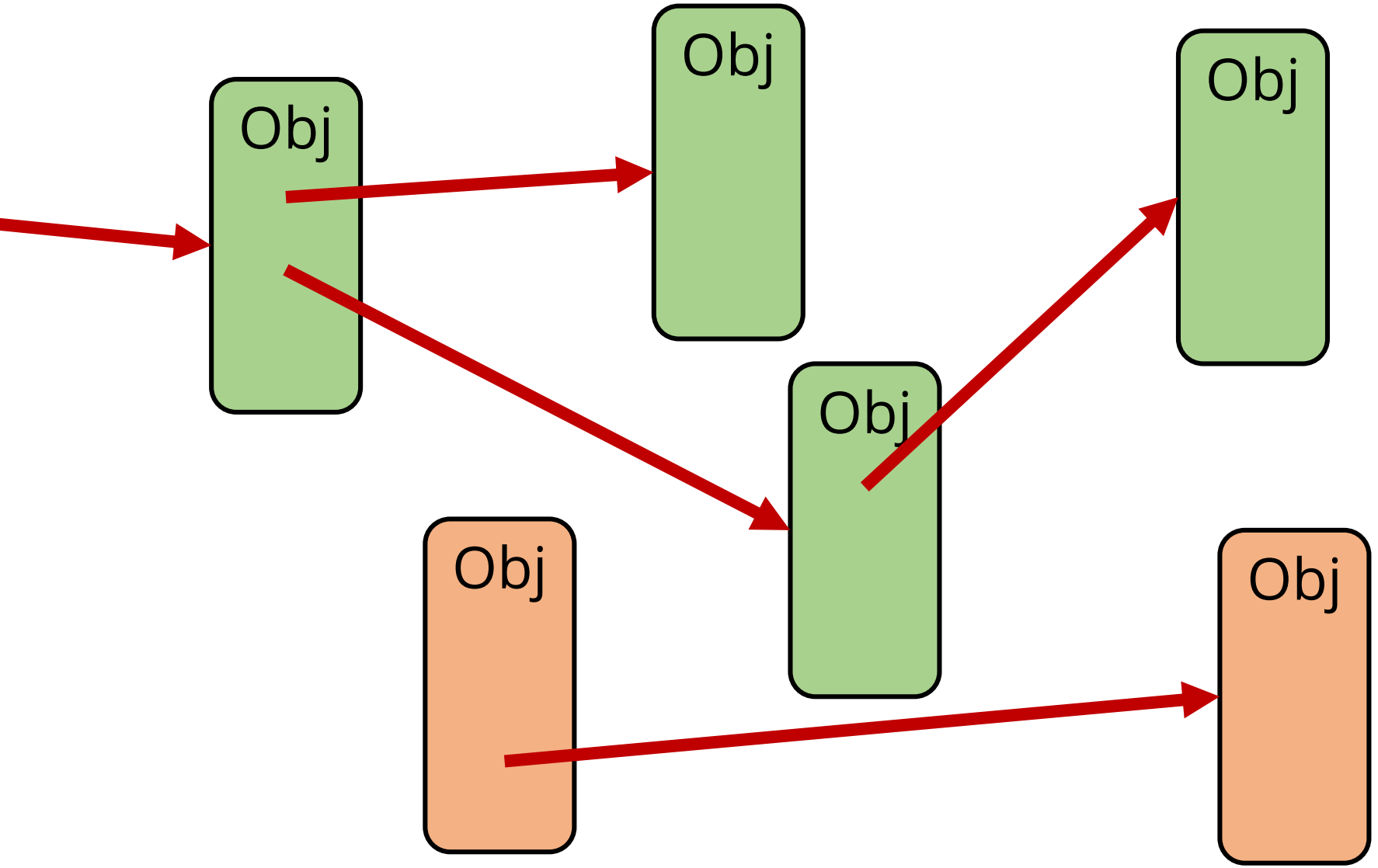
When to free?

- What we want:
  - Free an object when we’re done with it
- What does “done” mean?
- What we do:
  - Free an object when it’s unreachable

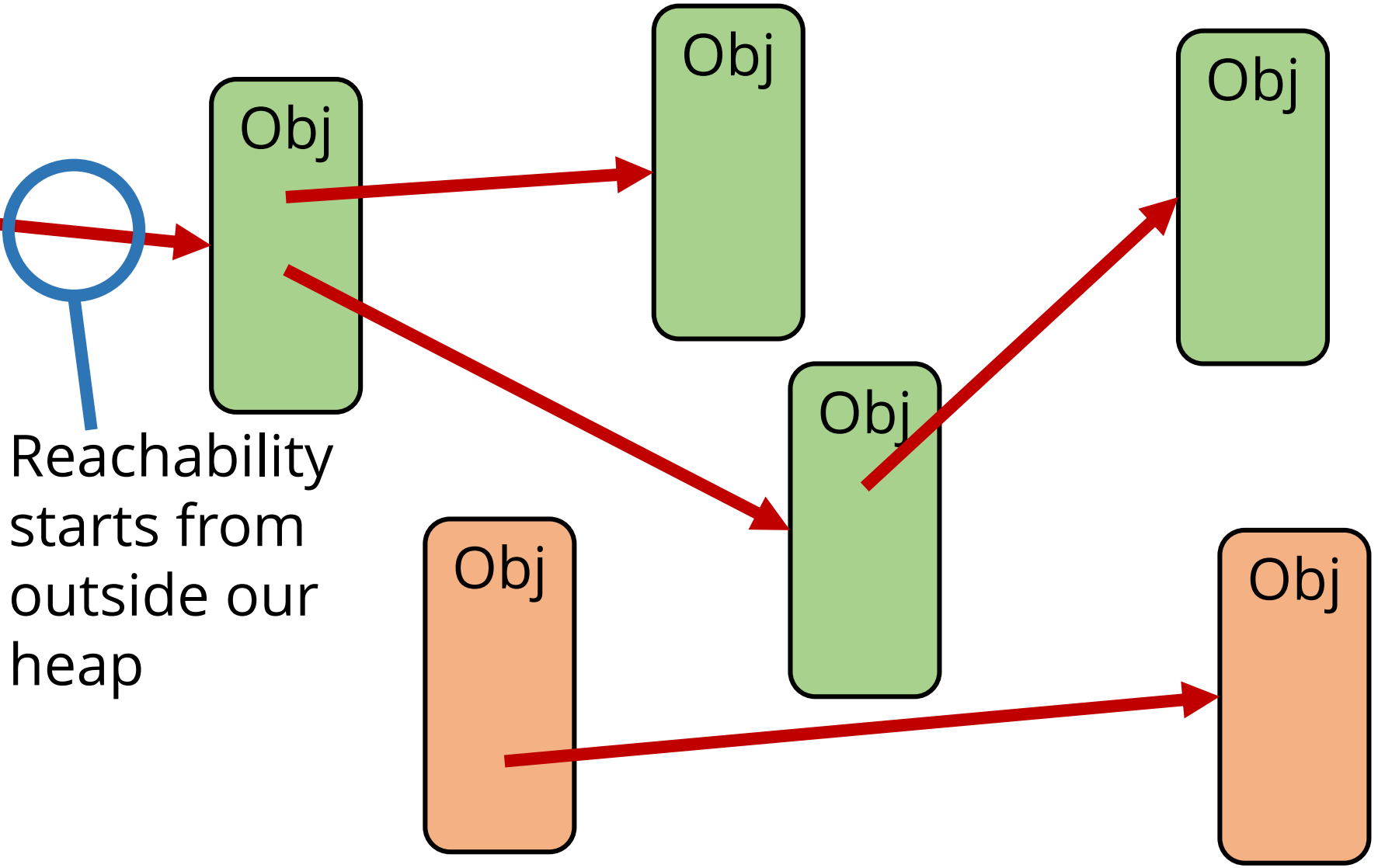
Reachable?



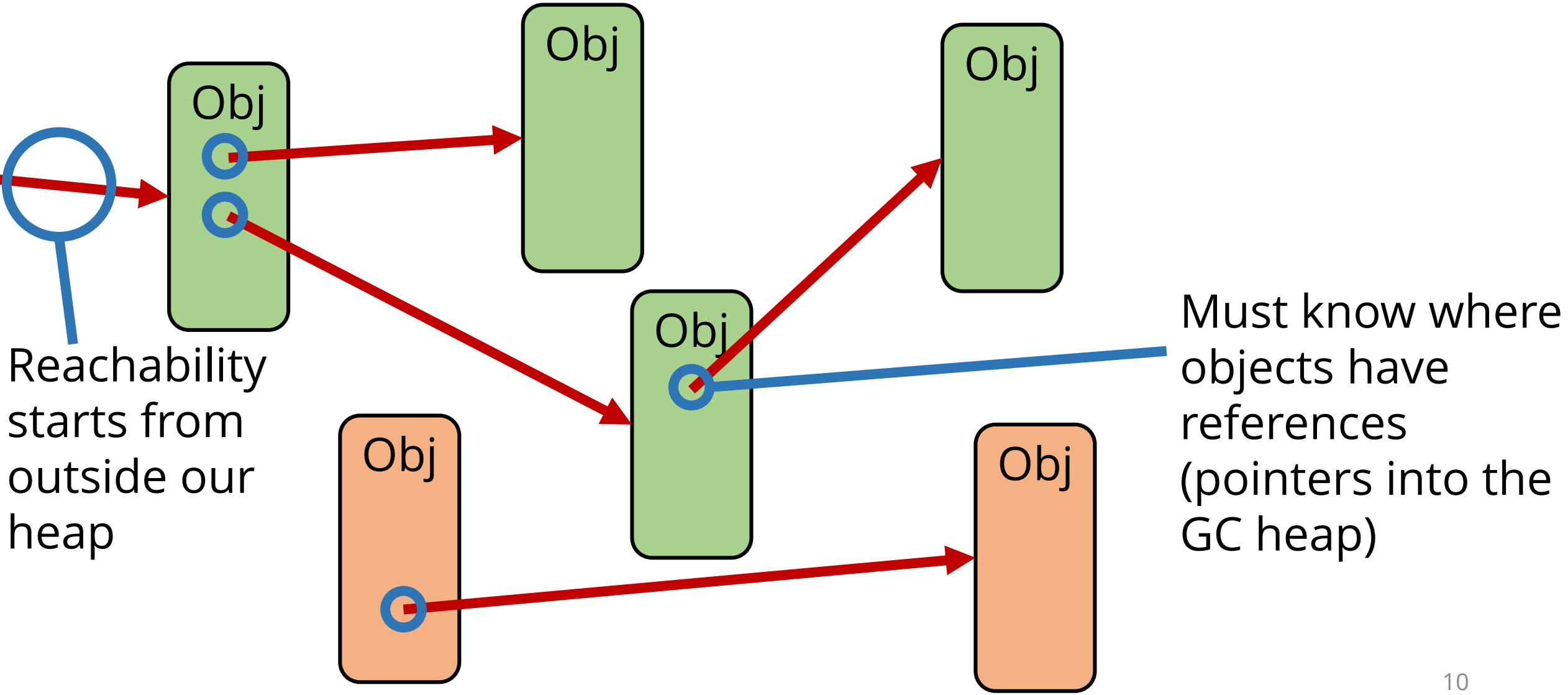
Reachable?



Reachable?


Obj
Obj
Obj
Obj
Obj
Obj
Reachability starts from outside our heap

Reachable?


Obj
Obj
Obj
Obj
Obj
Obj
Reachability starts from outside our heap
Must know where objects have references (pointers into the GC heap)

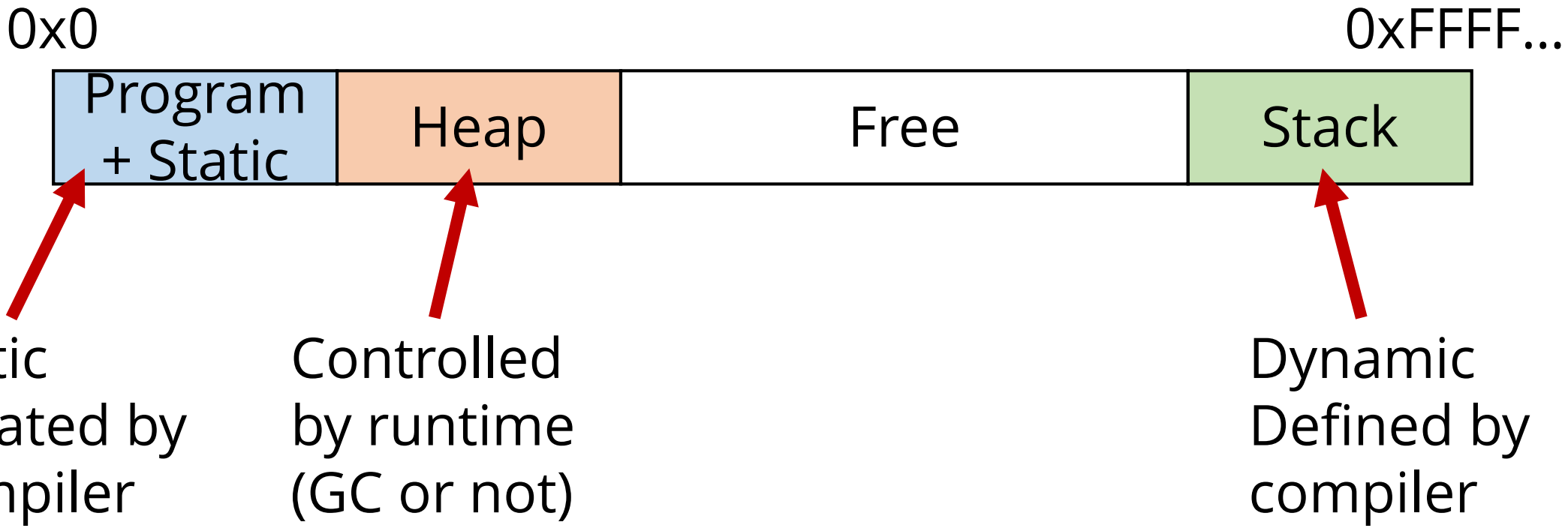
Aside: Can we do better?

- Easy to leak memory:

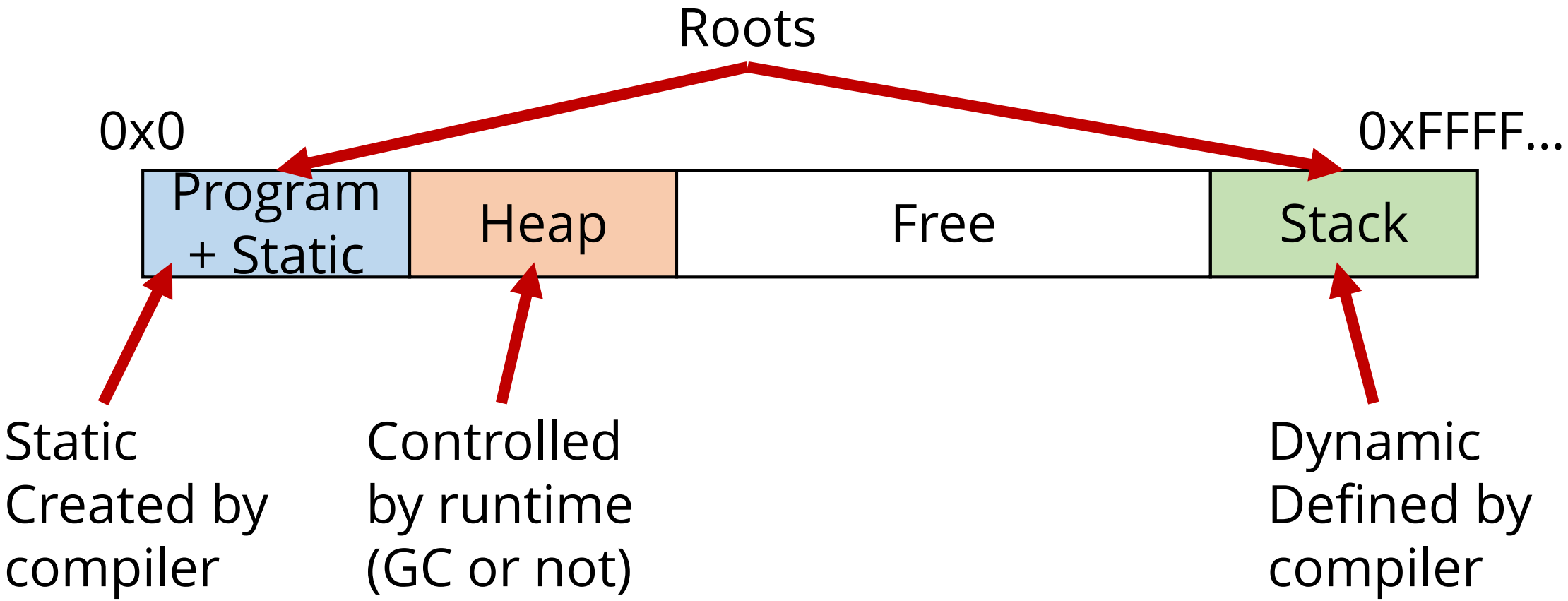
```
static List<Object> everyObjectIveEverAllocated;
```

- “No longer in use” → halting problem
- Fear not the halting problem:
  Liveness analysis is very real! But not general.

Roots


0x0
0xFFFF...
Program + Static
Heap
Free
Stack
Static
Created by compiler
Controlled by runtime (GC or not)
Dynamic
Defined by compiler

Roots


Roots
0x0
0xFFFF…
Program + Static
Heap
Free
Stack
Static
Created by compiler
Controlled by runtime (GC or not)
Dynamic
Defined by compiler

The compiler conundrum

- Roots are full of stuff
  - (Program code, non-reference variables, unused space…)
- To test reachability, we need *references*!
- Compiler must cooperate with GC: Tell the GC where references exist in roots

Stack references

Split stack:

```
; need 8 bytes stack
sub $8, %rsp

; and 8 bytes
; "pointer stack"
sub $8, %rbp
```

Marked stack:

```
; need 16 bytes
sub $16, %rsp

; tell GC about
; pointers
mov $8, %rax
call pushGCPointers
```

Conservativism

- Some languages just won’t play nice (I’m looking at you, C)
- We can at least *guess* where there are pointers

Heap references

- `malloc(size)` isn't enough!
- We need *references*!
- Need no more type information than that
- Crucial runtime type info: Pointer bitmap
- Extend header with pointer to type info

Allocation w/ type info

```
struct ObjectHeader {
  struct GCTypeInfo *typeInfo;
};

void *allocate(struct GCTypeInfo *typeInfo) {
  size_t size = typeInfo->size;
  // allocate as usual
  retHeader->typeInfo = typeInfo;
  return ret;
}
```

Aside: Alignment

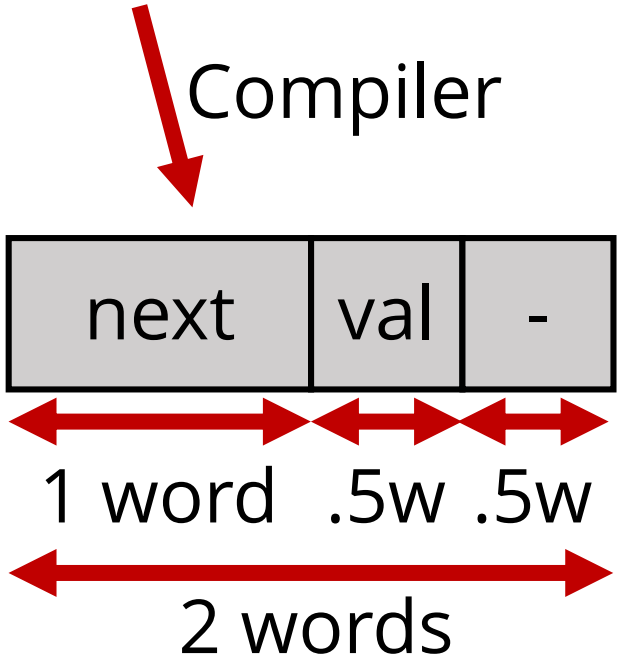
- Most systems require word-aligned pointers
- It therefore makes sense to word-align objects
- We only care about pointers, so only need one bit per word type info

Heap references

```
class IntList {
  IntList next;
  int val;
};
```

Heap references

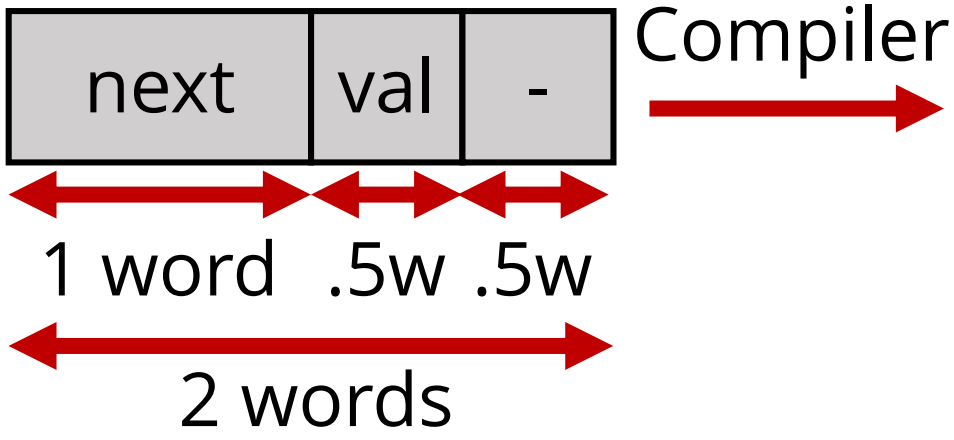
```
class IntList {
  IntList next;
  int val;
};
```



Heap references

```
class IntList {
  IntList next;
  int val;
};
```

Compiler



Compiler

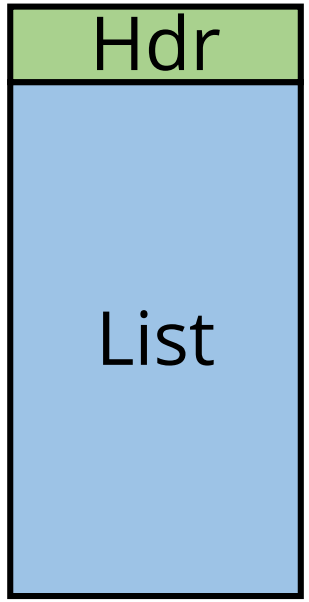
```
struct GCTypeInfo {
  size_t size;
  unsigned long pointerMap;
};
```

```
new GCTypeInfo(16, 0b1000…);
```

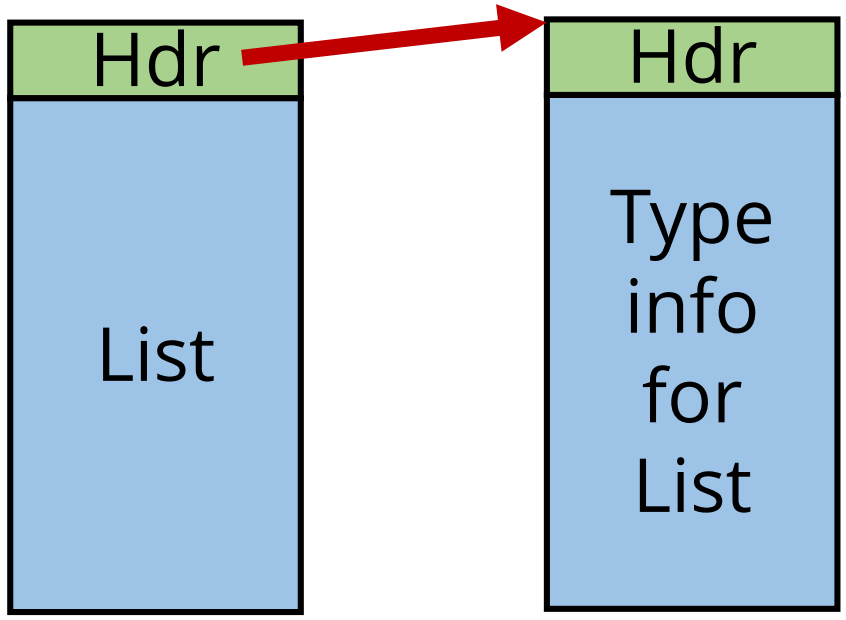
Aside: Dynamic type info

- All types known statically: Static type info
- What about types loaded at runtime?
- Type info object can be stored in the heap!

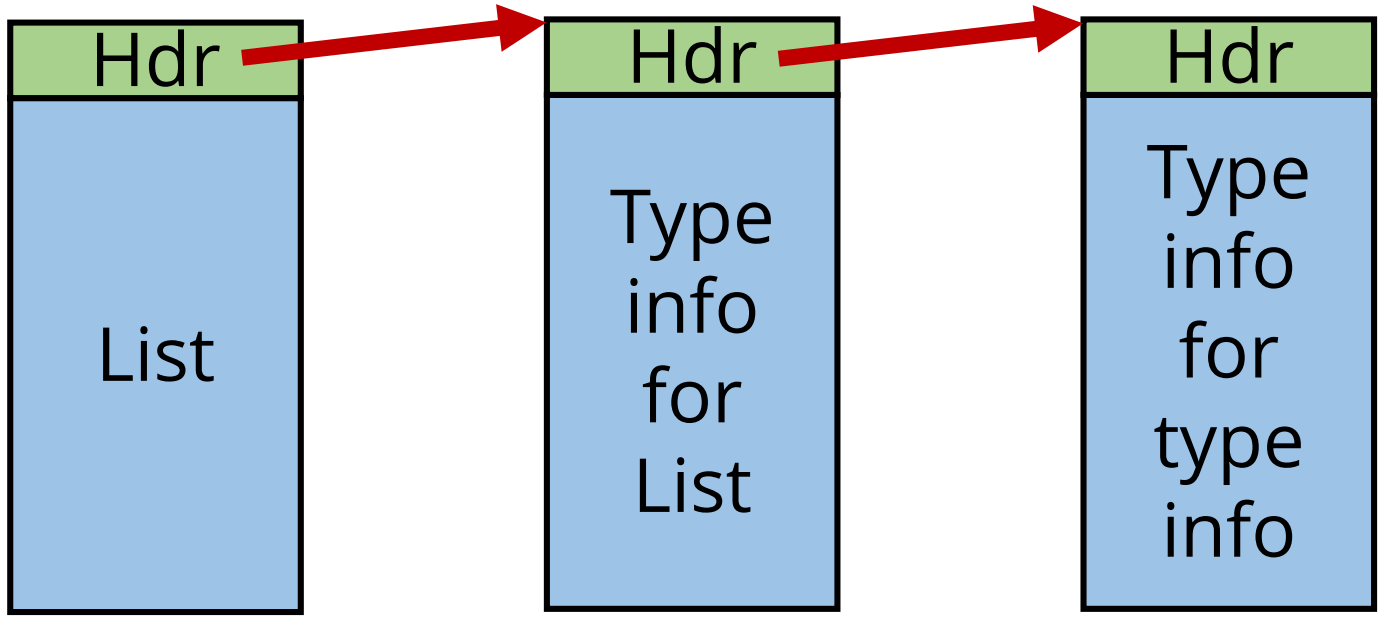
Aside: Dynamic type info



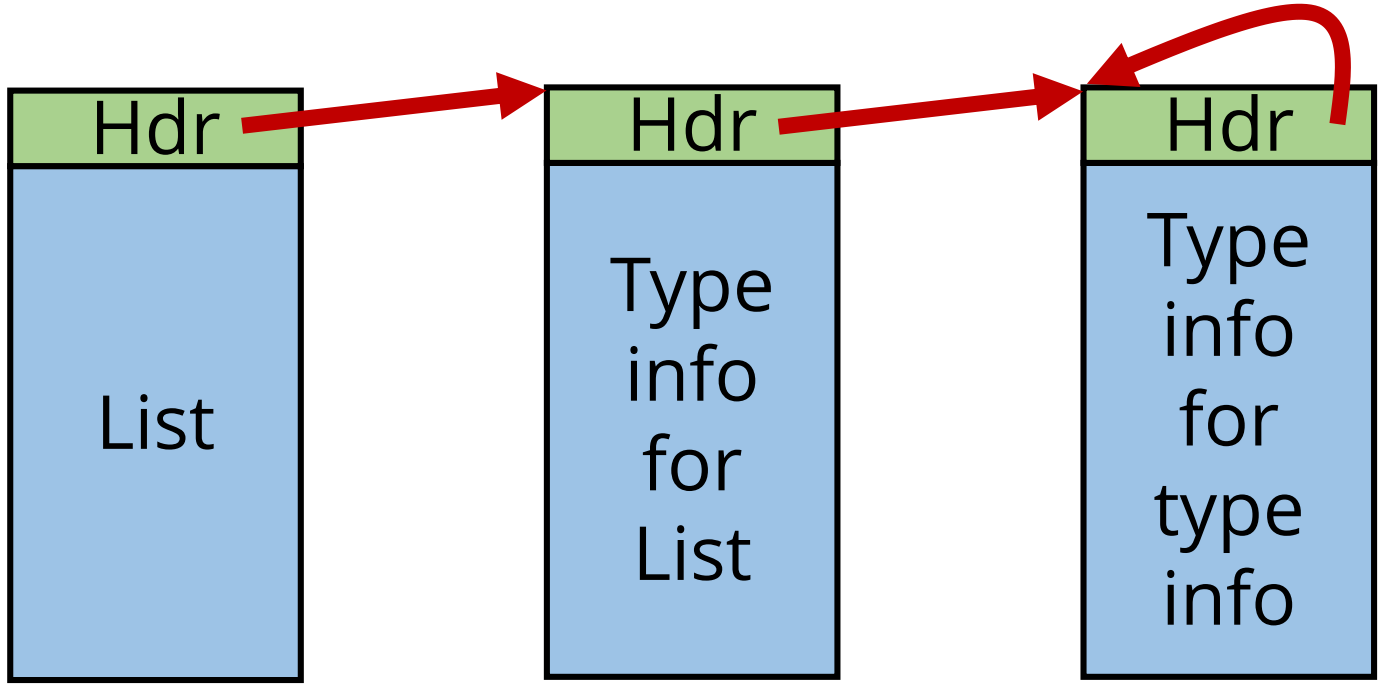
Aside: Dynamic type info



Aside: Dynamic type info



Aside: Dynamic type info



Breather

- Reachability
- Start with roots
- Compiler tells us references from roots
- Type info tells us references from objects

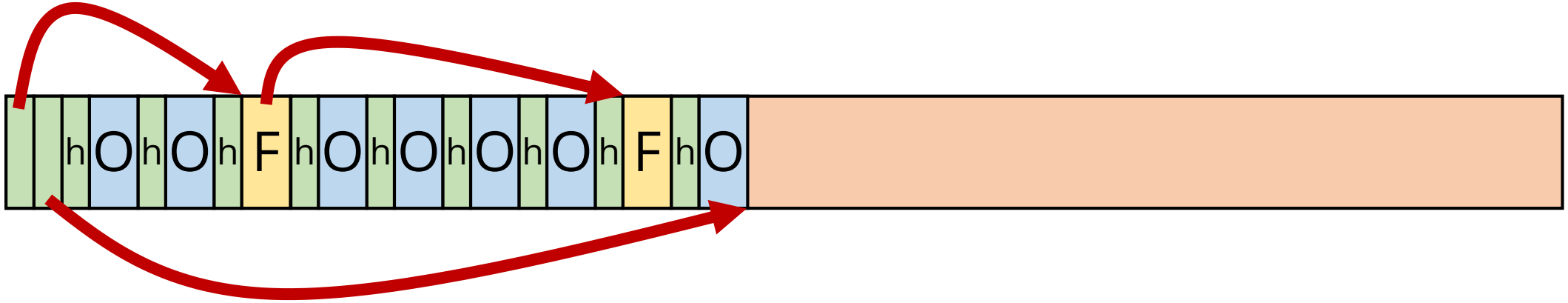
Garbage collection

- We've determined what *is* reachable…
- What *isn't* reachable is garbage!
- But… it's not reachable
- So how do we free it?

Parsable heap

- A heap is parsable if we can walk through all objects in it
- Parsability broken by gaps, lacking info or bugs

Parsable heap




```
struct Pool {
   struct FreeObject *freeObjects;
   void *freeSpace;
};
```

Collection

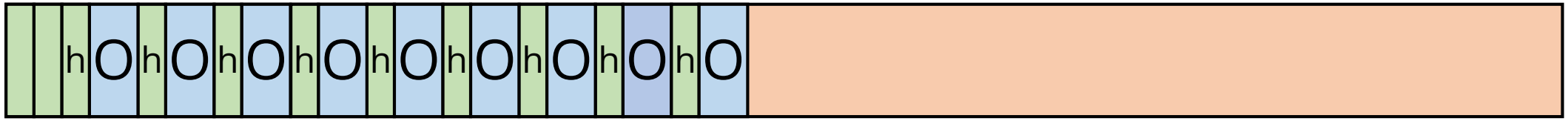
- While parsing the heap, how to tell which objects were unreachable?
- Add a “mark” field to headers, mark objects which are reached
- Unmarked objects are unreachable, so freed.

Mark and sweep

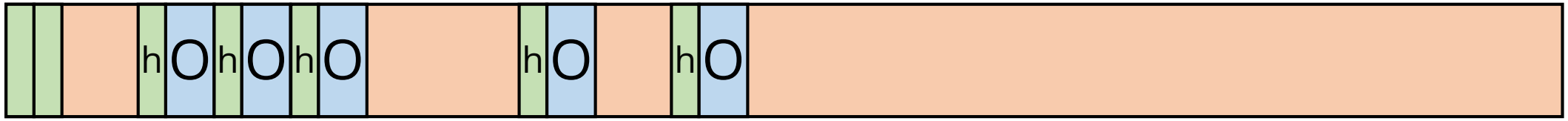
- That’s it! Now we have a mark and sweep garbage collector!

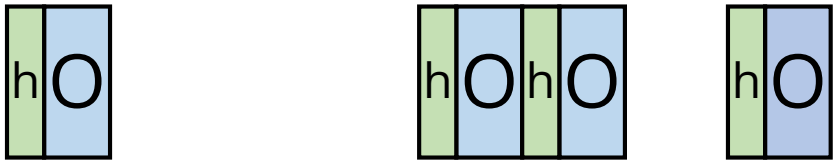
(it would be competitive in 1974)

Moving

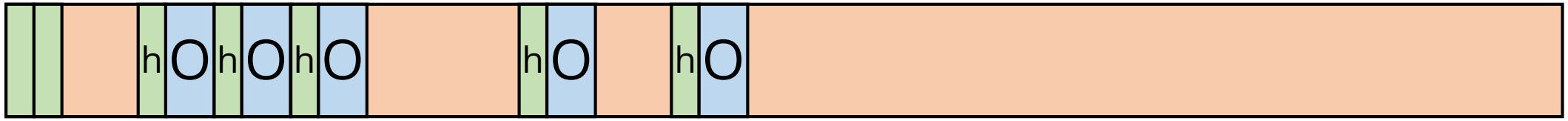


Moving

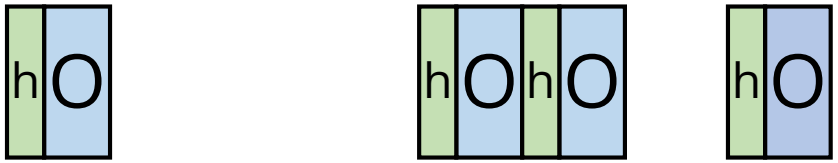




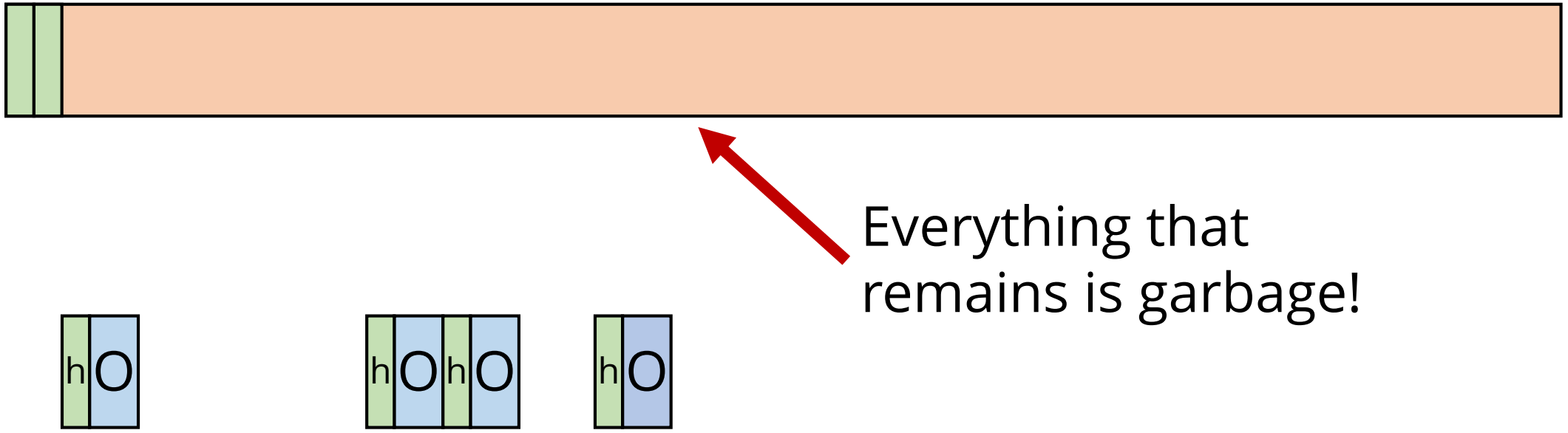
Moving



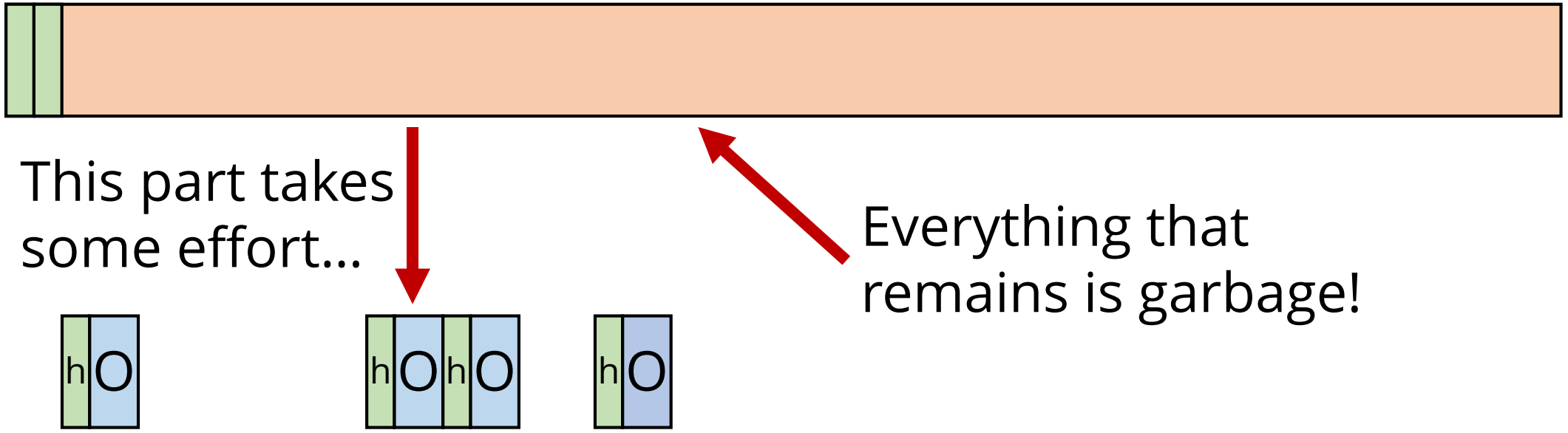
Everything that remains is garbage!



Moving



Moving


This part takes some effort...
Everything that remains is garbage!
h
O
h
O
h
O
h
O

Moving

- Must be able to update all references
- Compiler tells us about references anyway
- But… isn’t moving memory expensive?

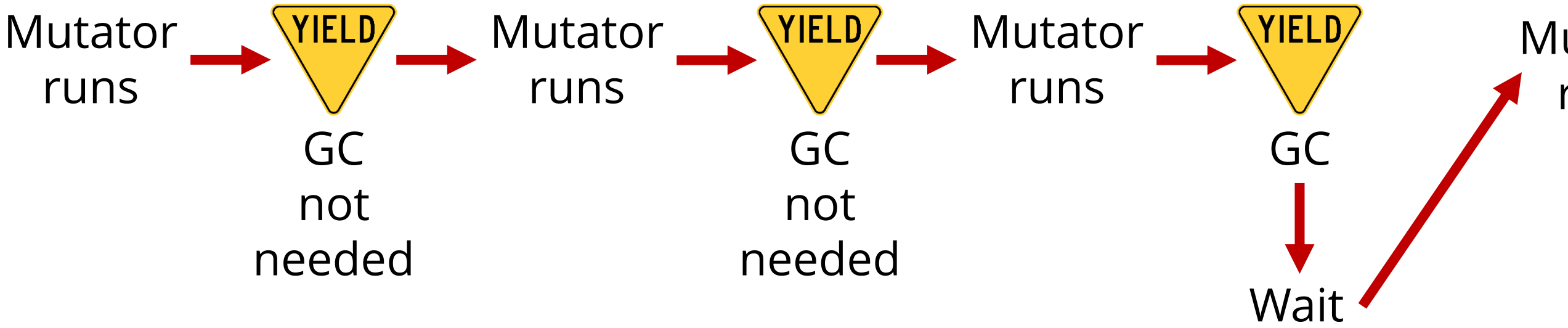
Lifetime principles

- Most objects die young
- Therefore, more valuable to save time on (plentiful) dead objects than (few) living ones
- Moving can be worth it

Stop-the-world

- When do we scan?
- If mutator still running, we can miss references
- So, stop the (mutator's) world before GC
- Compiler implication: Yieldpoints

Yieldpoint


Mutator runs
YIELD
GC not needed
Mutator runs
YIELD
GC not needed
Mutator runs
YIELD
GC
Wait

Less stopping

- Incremental: Do some GC work each time
  - Difficulty: Hard to do half of reachability
- Concurrent: Do GC simultaneous with mutator work
  - Difficulty: Communicating reference changes

Summary

- Compiler tells us roots
- Compiler tells us when we can collect
- Find reachable objects
- Discard unreachable objects