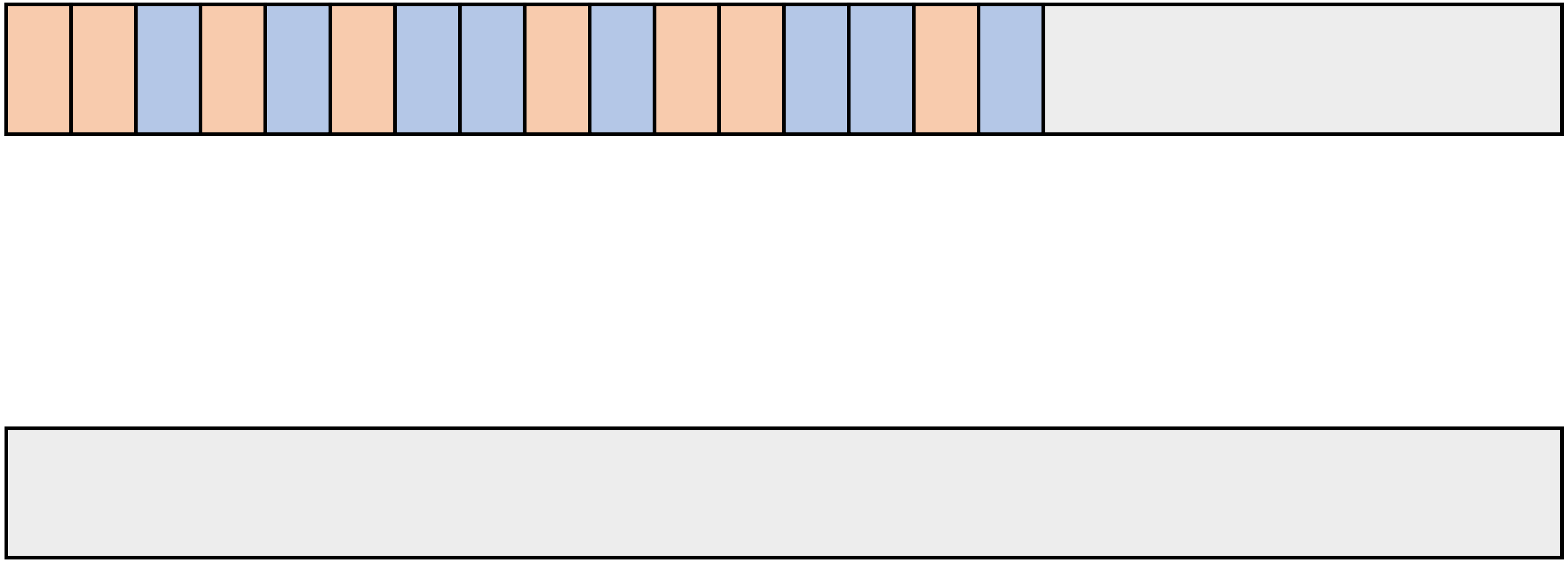
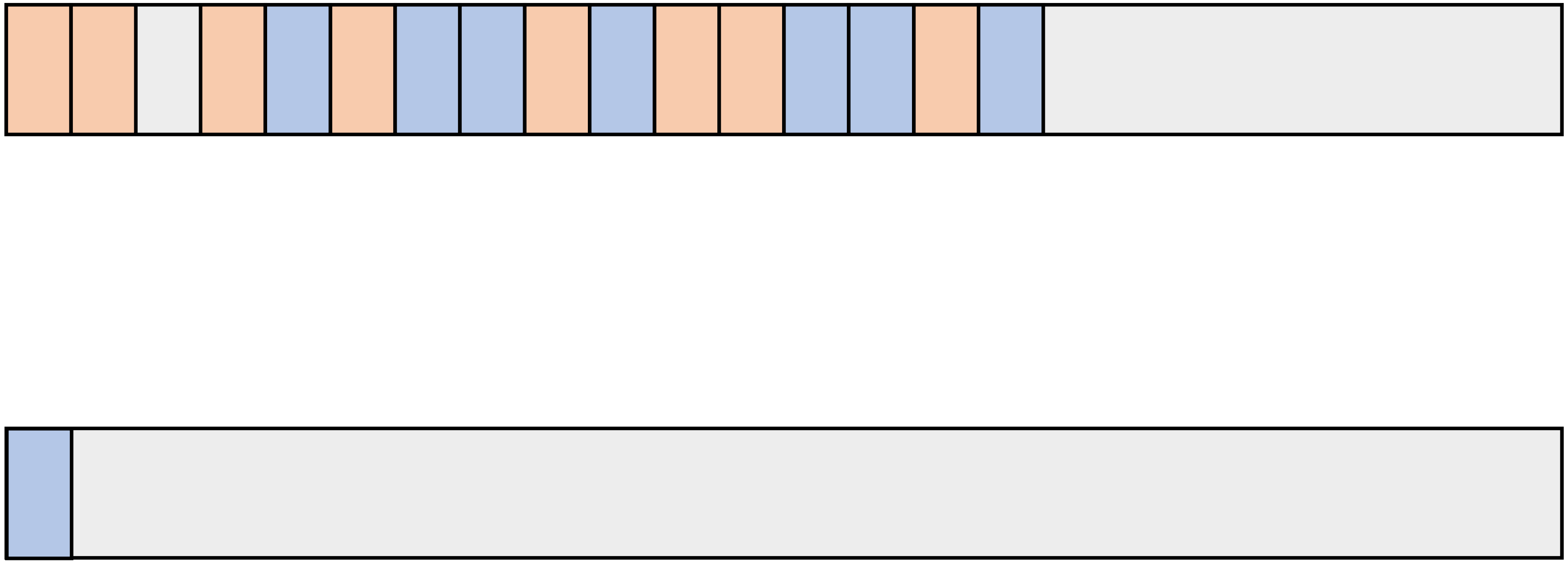
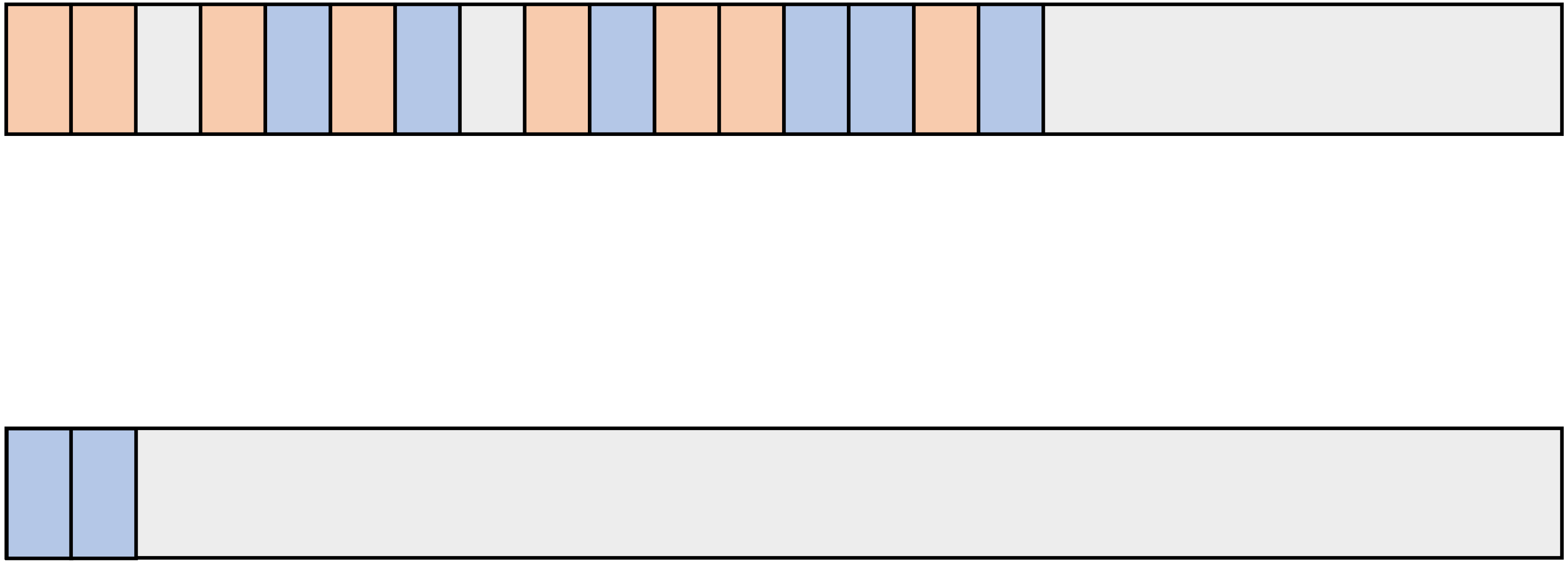
Semispace copying



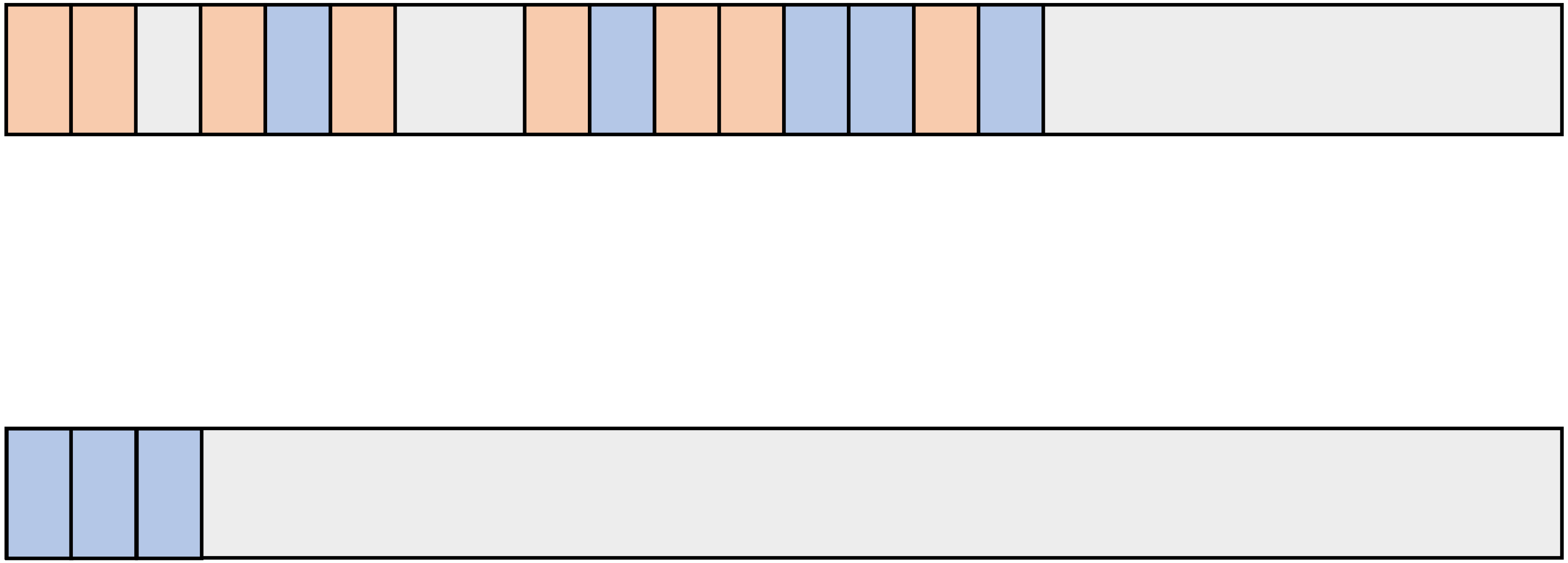
Semispace copying



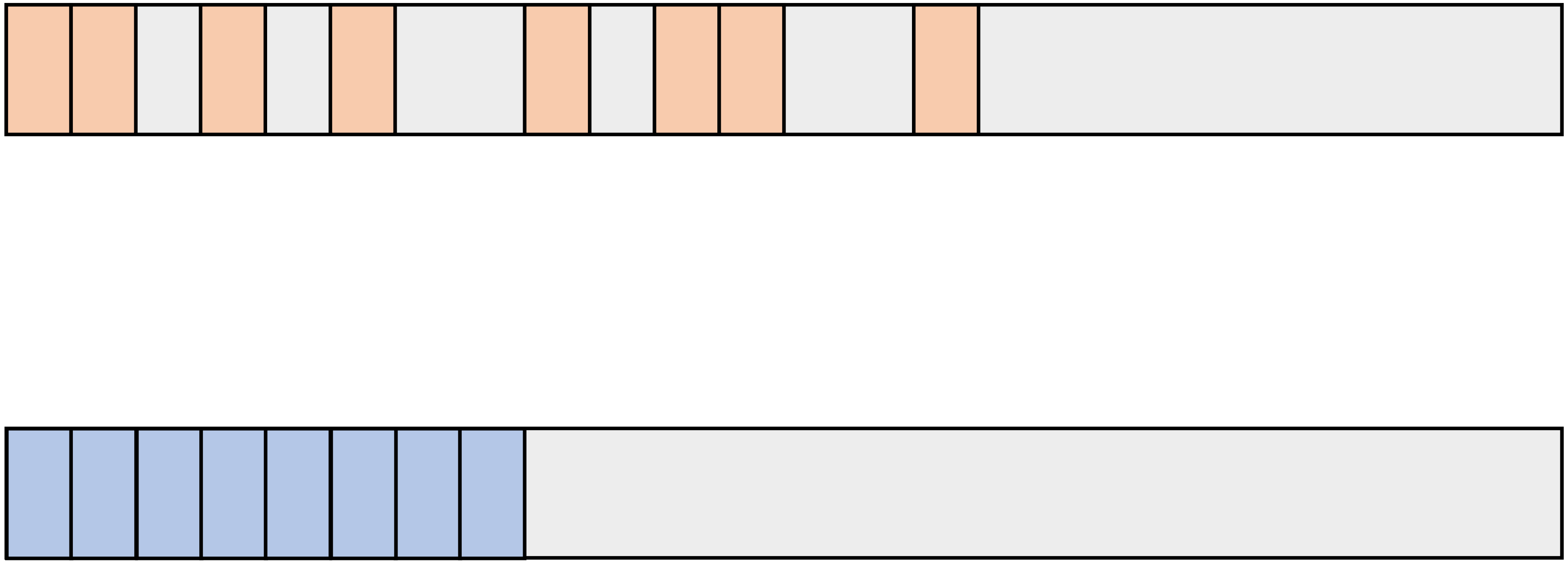
Semispace copying



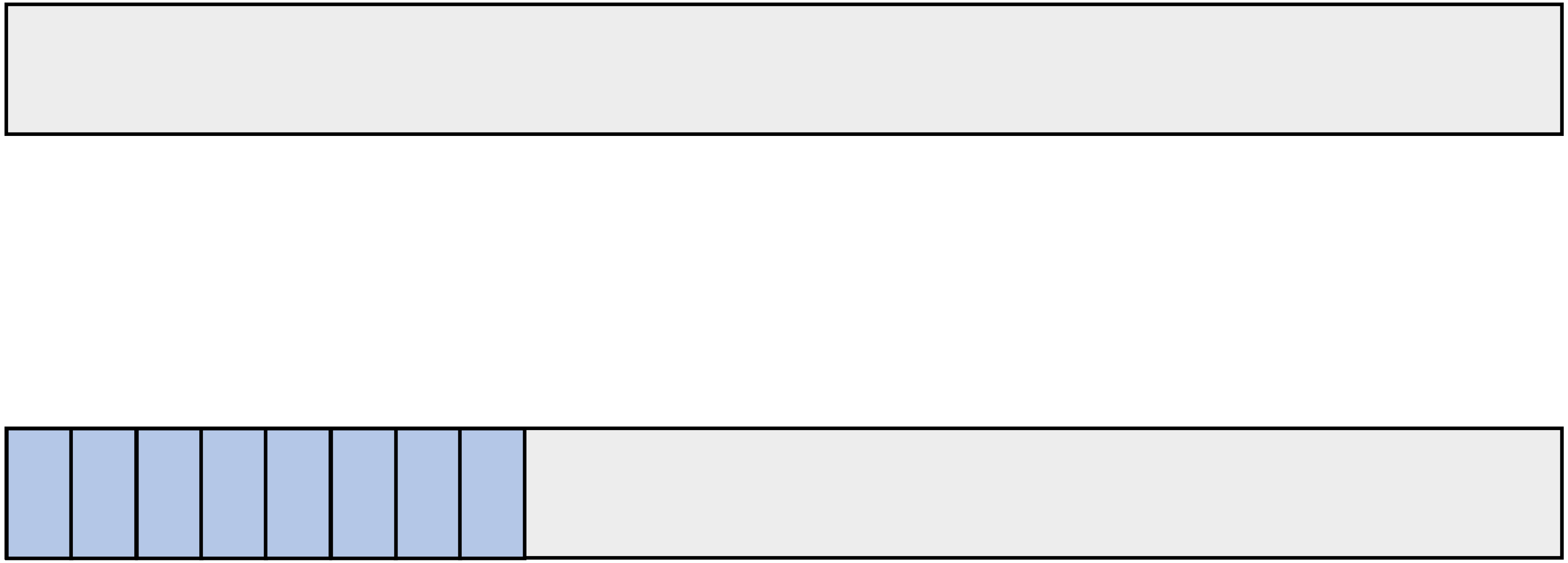
Semispace copying



Semispace copying



Semispace copying

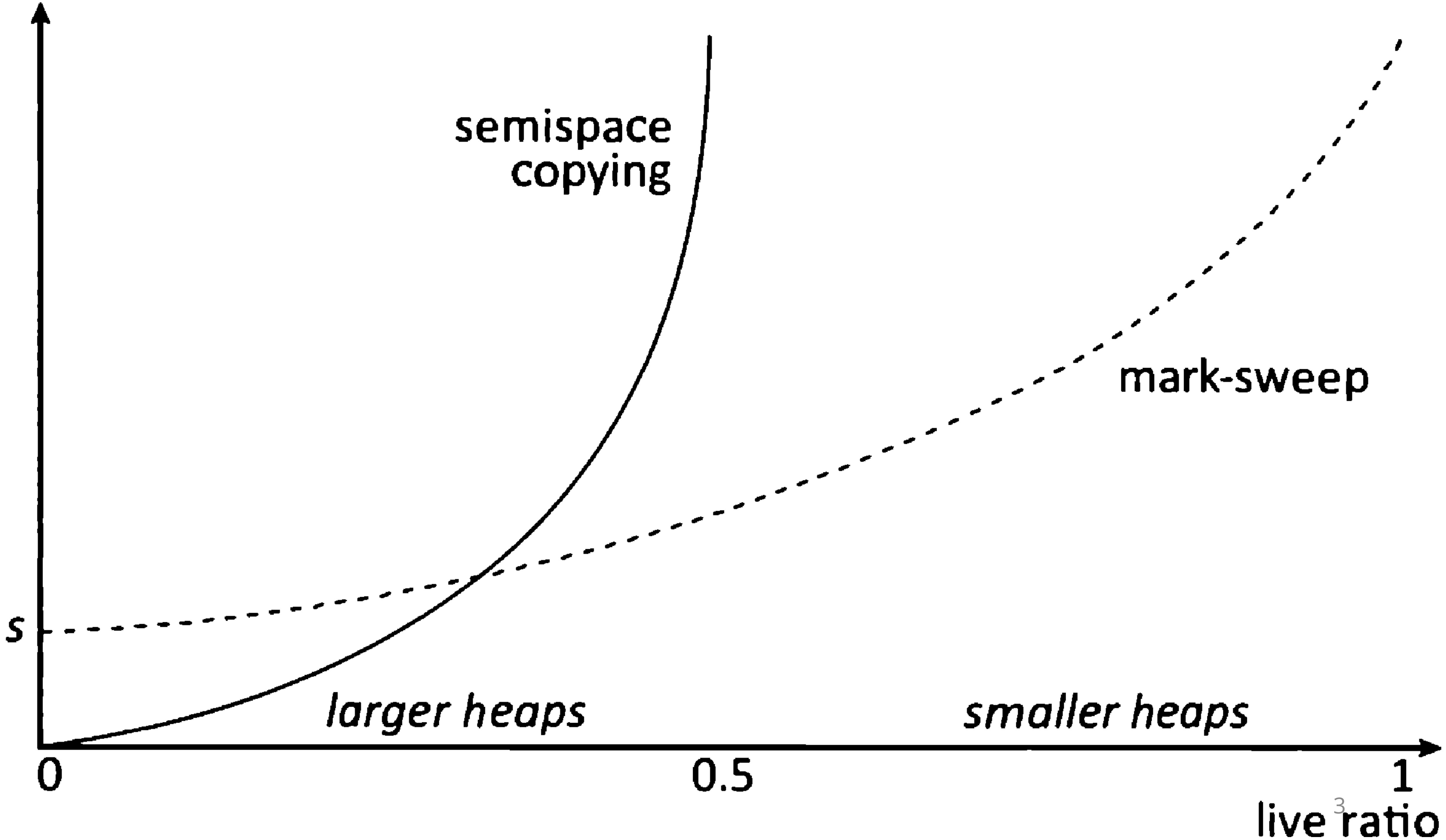


```
collect():
  fromspace, tospace := tospace, fromspace
  worklist := new Queue
  foreach loc in roots:
    process(loc)
  while (ref := worklist.pop()):
    scan(ref)

scan(ref):
  foreach loc in ref->header.descriptor->ptrs:
    process(ref+loc)

process(loc):
  fromRef := *loc
  if fromRef != NULL:
    *loc := forward(fromRef)

forward(fromRef):
  if alreadyMoved(fromRef):
    return forwardingAddress(fromRef)
  toRef := (allocate in tospace)
  memcpy(toRef, fromRef, fromRef->header.size)
  setForwardingAddress(fromRef, toRef)
  worklist.push(toRef)
  return toRef
```


semispace
copying
mark-sweep
s
larger heaps
smaller heaps
0
0.5
1
live ratio

When to GC?

- Typically: When tospace is full
- GC takes O(L)
- (L is a constant for most programs)

Allocating pools

- Must keep two sets of pools
- Always allocate in both!
- Tospace “mirrors” fromspace, but don’t need individual frompools and topools

When to allocate pools

- Need double the space of mark&sweep
- Performance consideration:
  - Throughput
  - Latency
- More pools *always* better throughput

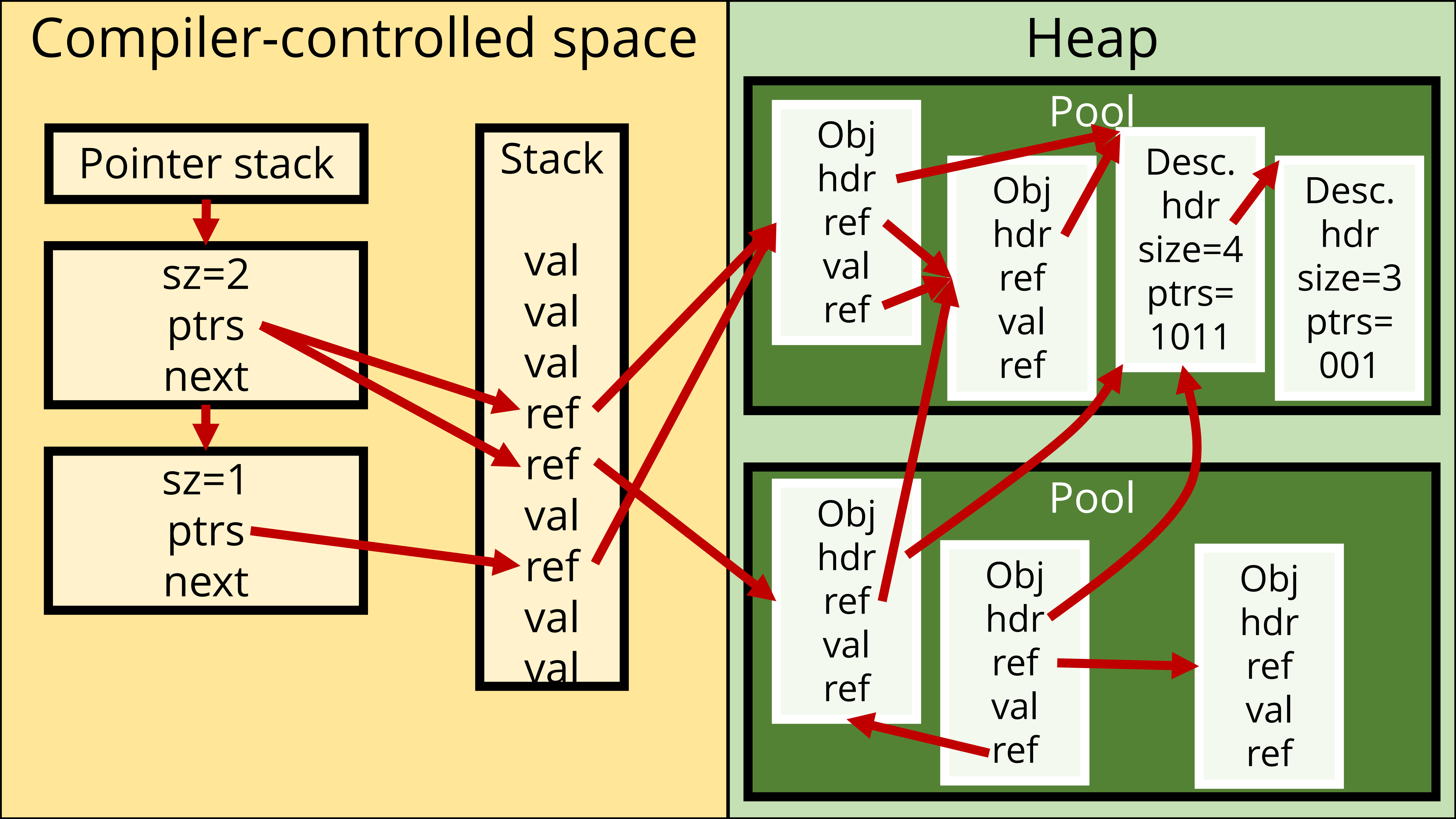
The Devil is in the Details

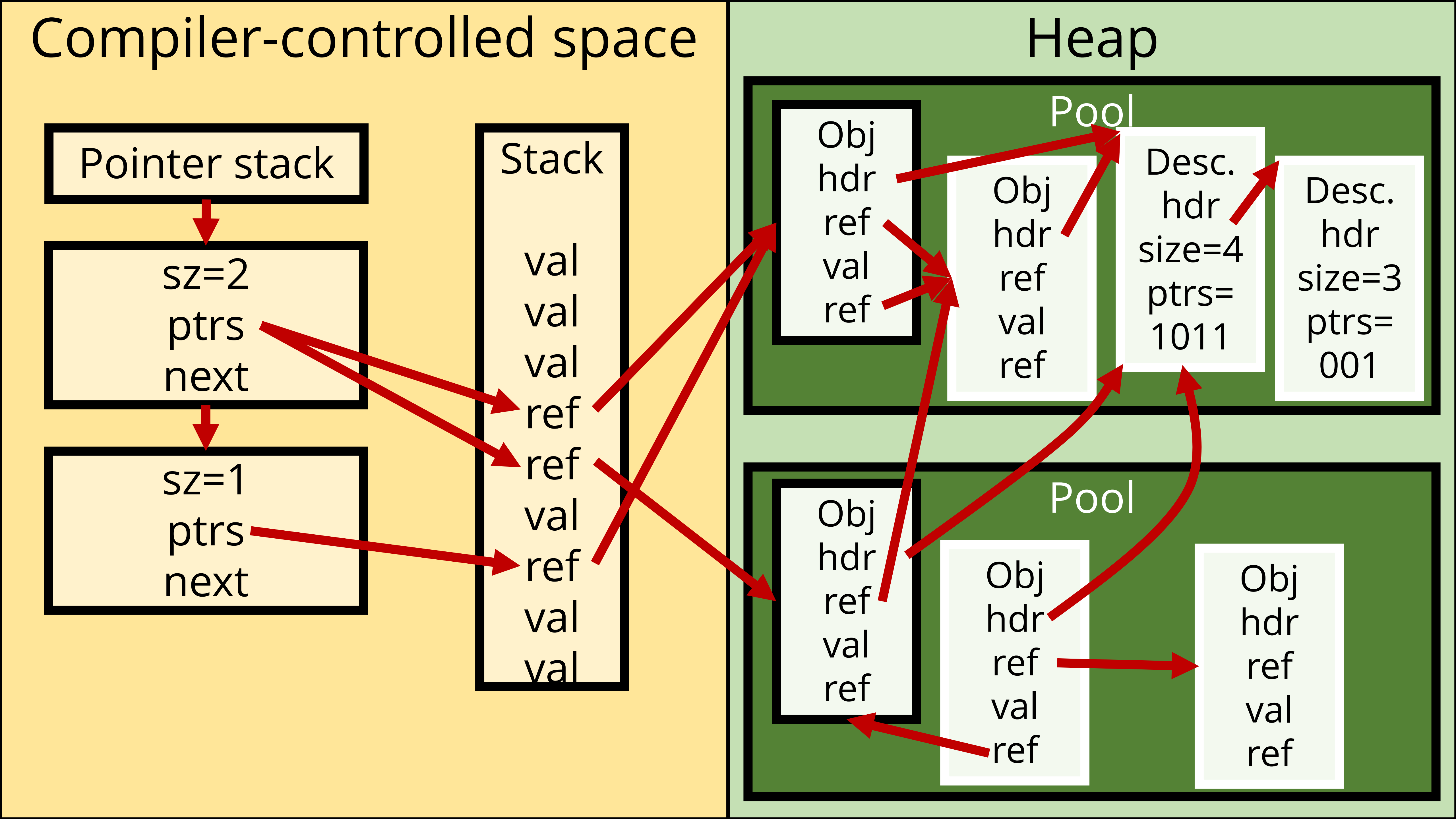
Schedule

	M	W
Sept 14	Intro/Background	Basics/ideas
Sept 21	Allocation/layout	GGGGC
Sept 28	Mark/Sweep	Copying GC
Octo 5	Details	Ref C
Octo 12	Thanksgiving	Mark/Compact
Octo 19	Partitioning/Gen	Generational
Octo 26	Other part	Runtime
Nove 2	Final/weak	Conservative
Nove 9	Ownership	Regions etc
Nove 16	Adv topics	Adv topics
Nove 23	Presentations	Presentations
Nove 30	Presentations	Presentations

Project 1

- Superficially: Make a mark-and-sweep collector
  - Free-list allocator, mark phase, sweep phase
- Really: Bits and bits and bits and bits


Compiler-controlled space
Heap
Pointer stack
sz=2
ptrs
next
sz=1
ptrs
next
Stack
val
val
val
ref
ref
val
ref
val
val
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Desc.
hdr
size=4
ptrs=
1011
Desc.
hdr
size=3
ptrs=
001
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref



Compiler-controlled space

Pointer stack

sz=2
ptrs
next

sz=1
ptrs
next

Stack
val
val
val
ref
ref
val
ref
val
val

Heap

Pool

Obj
hdr
ref
val
ref

Obj
hdr
ref
val
ref

Desc.
hdr
size=4
ptrs=
1011

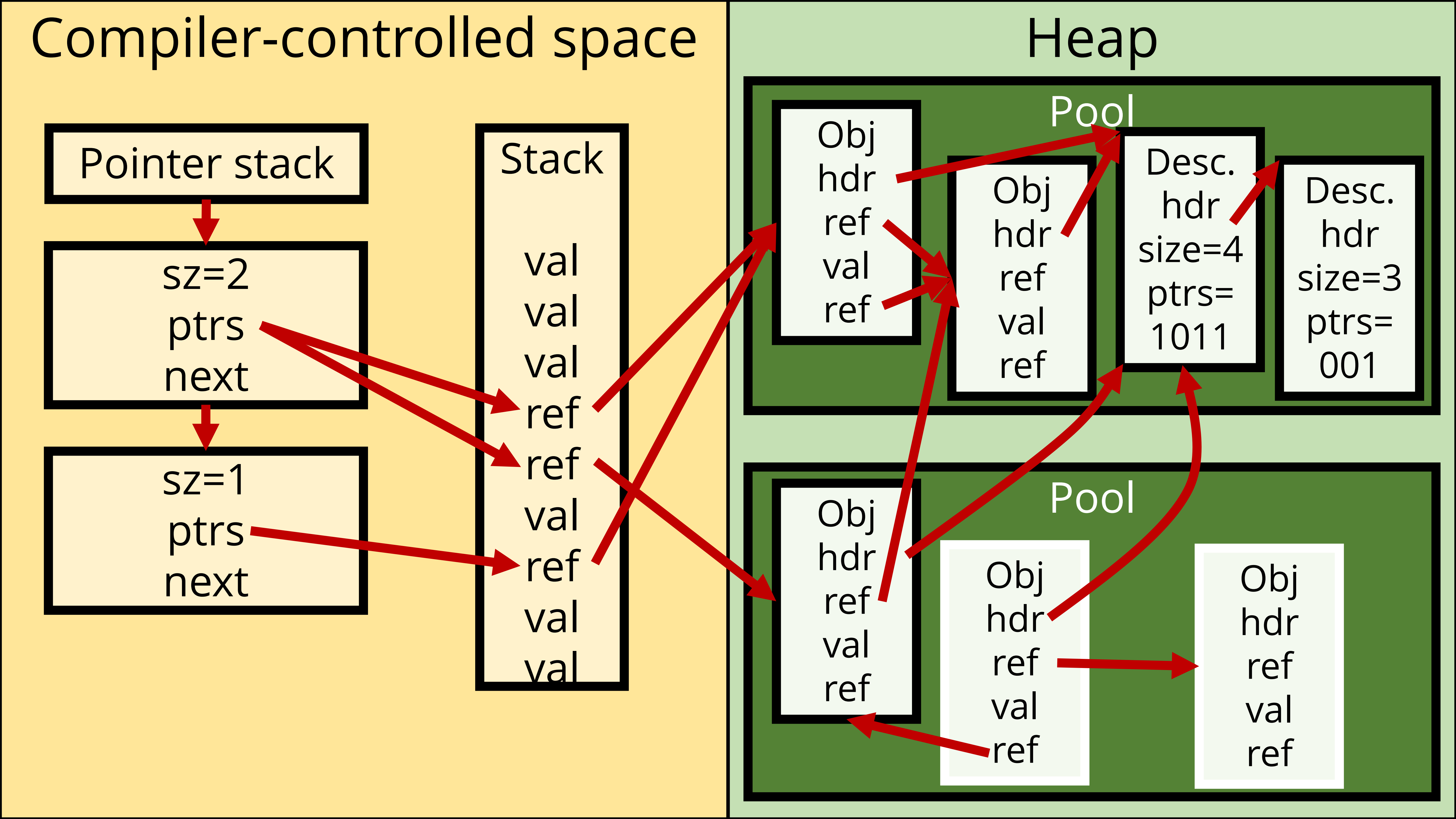
Desc.
hdr
size=3
ptrs=
001

Pool

Obj
hdr
ref
val
ref

Obj
hdr
ref
val
ref

Obj
hdr
ref
val
ref



Compiler-controlled space

Pointer stack

sz=2
ptrs
next

sz=1
ptrs
next

Stack
val
val
val
ref
ref
val
ref
val
val

Heap

Pool

Obj
hdr
ref
val
ref

Obj
hdr
ref
val
ref

Desc.
hdr
size=4
ptrs=
1011

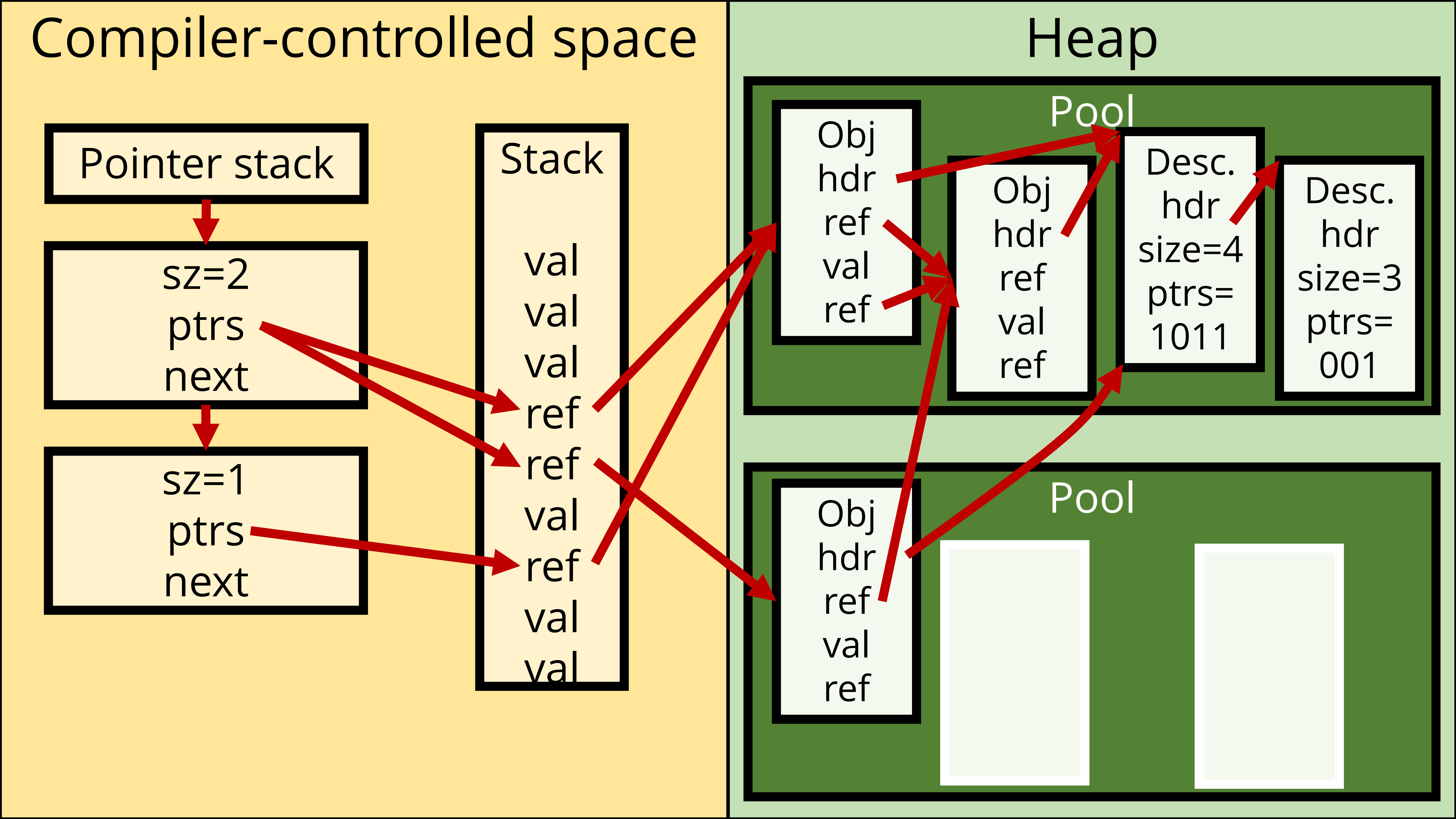
Desc.
hdr
size=3
ptrs=
001

Pool

Obj
hdr
ref
val
ref

Obj
hdr
ref
val
ref

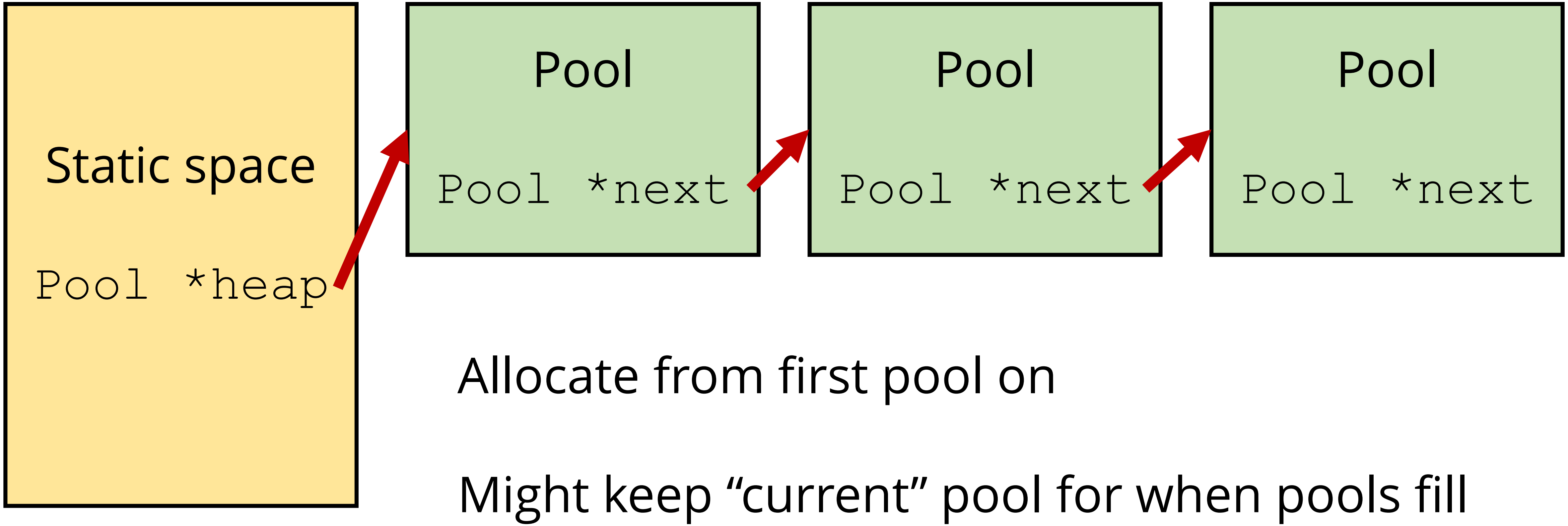
Obj
hdr
ref
val
ref


Compiler-controlled space
Heap
Pointer stack
sz=2
ptrs
next
sz=1
ptrs
next
Stack
val
val
val
ref
ref
val
ref
val
val
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Desc.
hdr
size=4
ptrs=
1011
Desc.
hdr
size=3
ptrs=
001
Pool
Obj
hdr
ref
val
ref

Heap

- OS is dumb: Gives you some pages
- GC maintains pools
- “Heap” is all pools
- GC must keep track

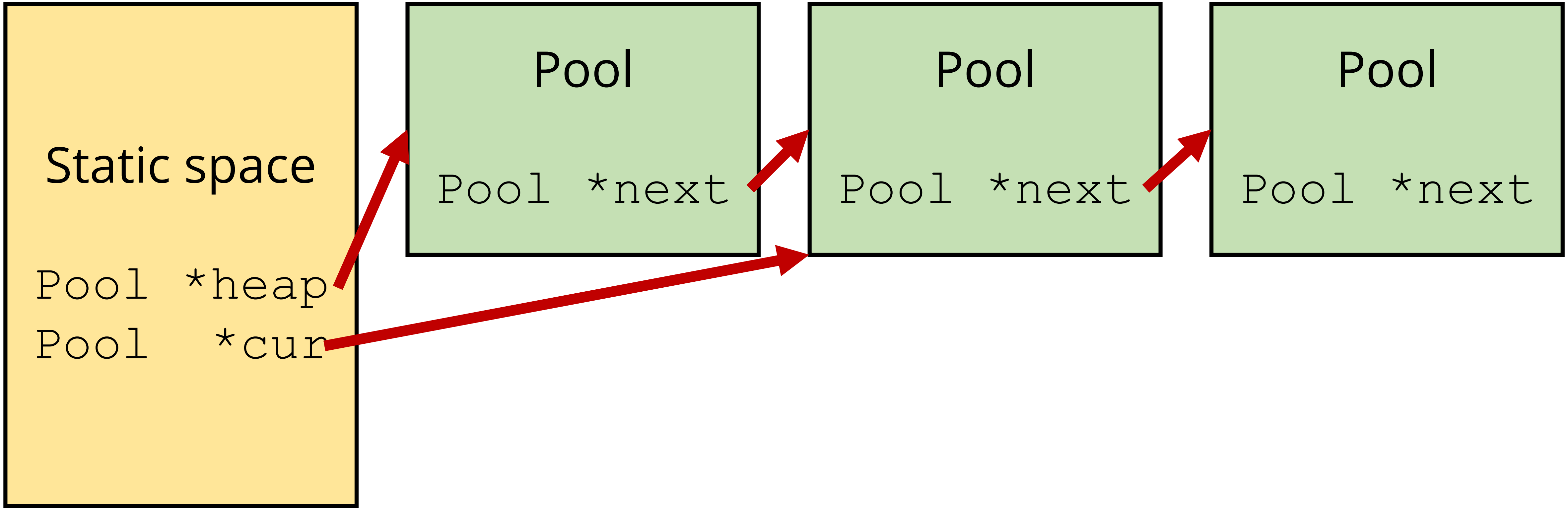
Keeping pools



Allocate from first pool on

Might keep “current” pool for when pools fill

Keeping pools



Static space

Pool *heap
Pool *cur

Pool

Pool *next

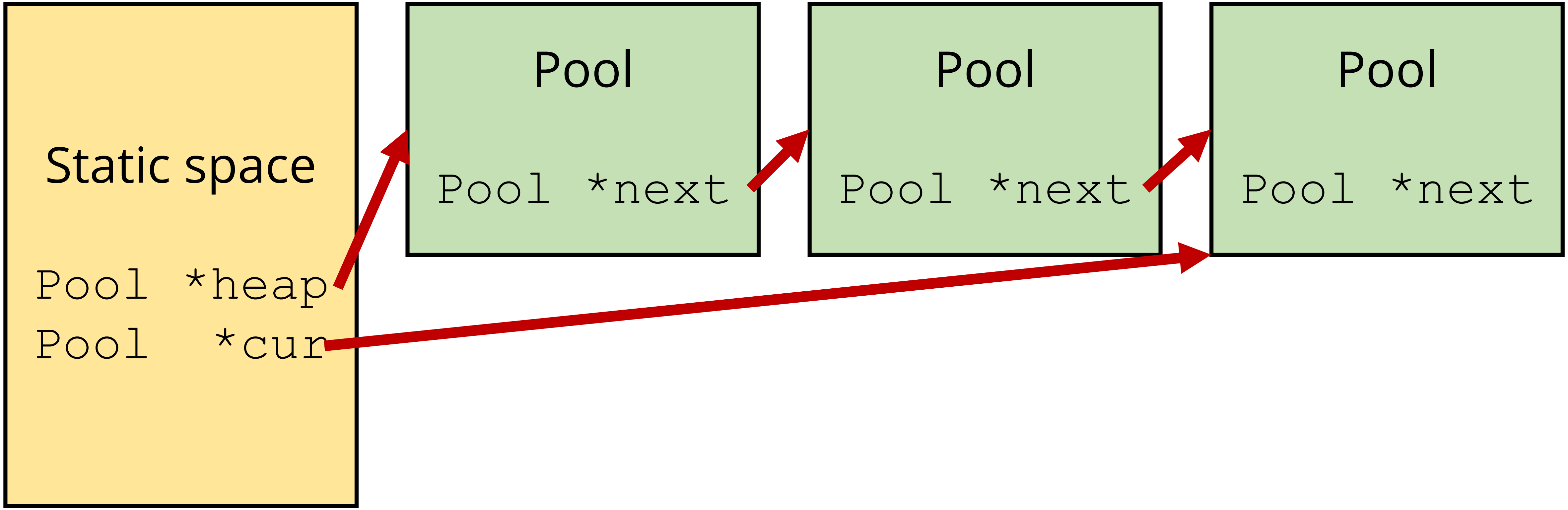
Pool

Pool *next

Pool

Pool *next

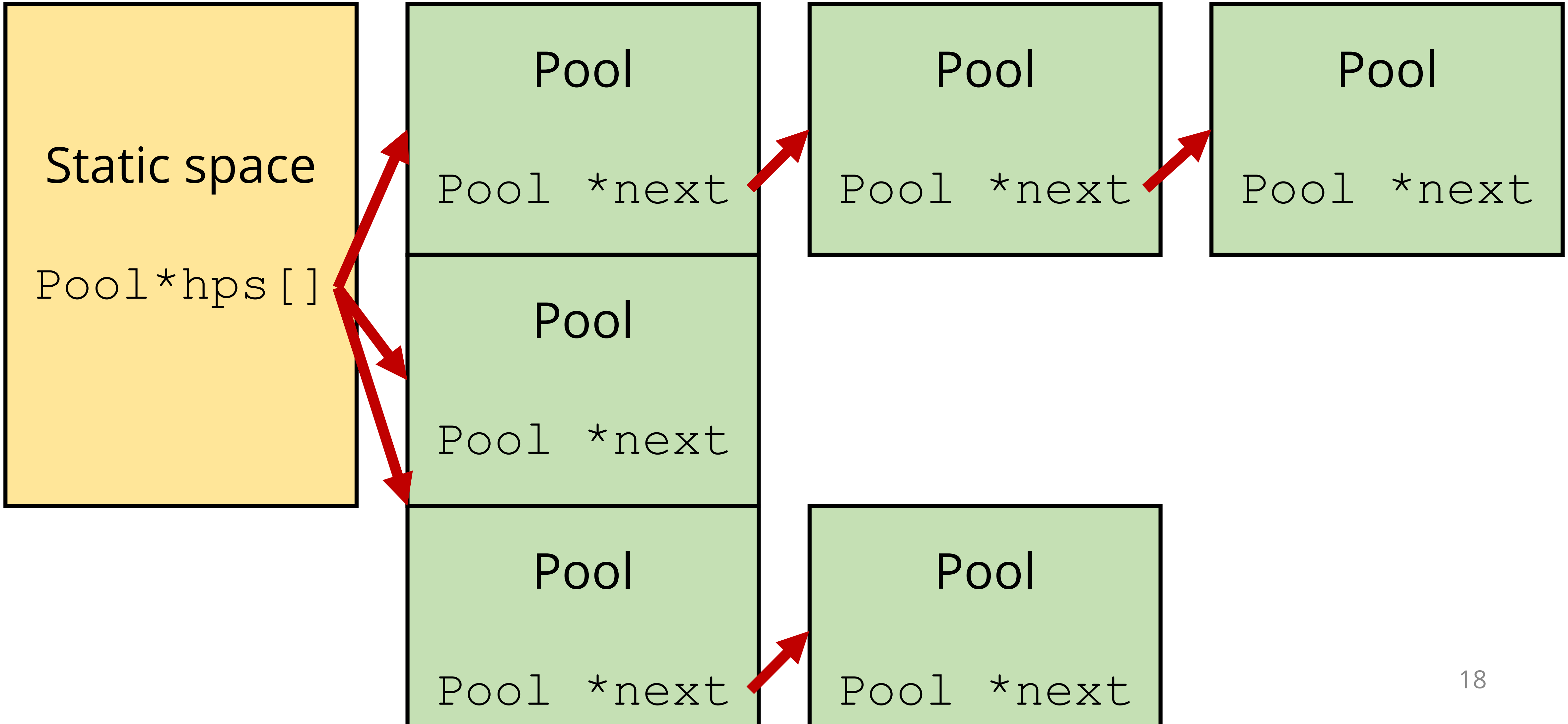
Keeping pools



Segregated blocks

- With segregated blocks, pools have fixed-sized objects
- No reason to mingle dissimilar pools

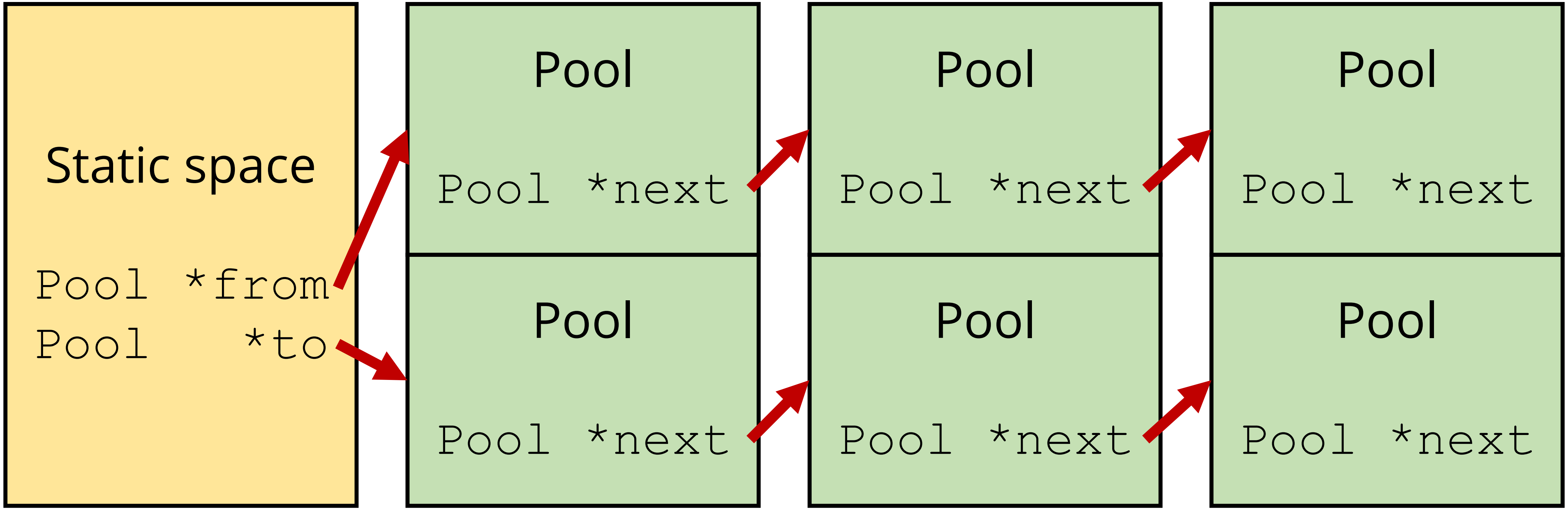
Pools w/ segregated blocks



Pools w/ semispace copying

- Need fromspace and tospace
- Pool “spaces” are non-intersecting, equal size

Keeping pools



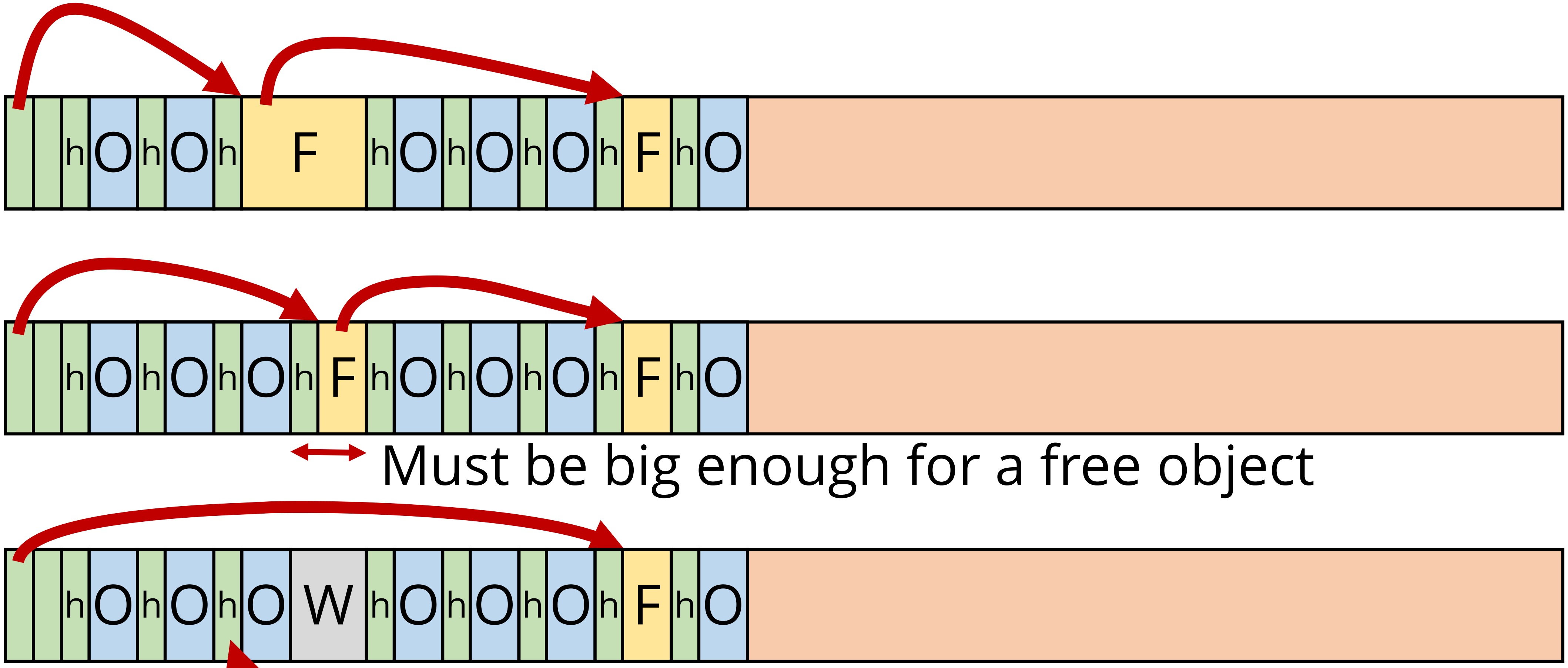
Free-lists

- Global or per-pool?
- Global: Thread contention (not an issue for now)
- Per-pool:
  - Go through every pool every allocation? Or
  - Accept lost space after large allocations?

Free-list order

- Mark-and-sweep makes address-ordered free-list
- Pools aren’t necessarily address-ordered
- Should they be?

Splitting vs overallocation


h O h O h F h O h O h O h F h O
h O h O h O h F h O h O h O h F h O
h O h O h O W h O h O h O h F h O

Must be big enough for a free object

Header must specify sizeof(O+W)

Overallocating

- Can be avoided:
  - Bitmapped-fits
  - Allocation granule ≥ size of free object
  - Non-free-list allocation
- Let’s think about headers…

Overallocating

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

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

Cannot change per object

Does not represent overallocated size

Objects

- GC only knows:
  - Size
  - Location of references
- Both are in descriptor, also a GC object!
- Must make sure to keep object descriptors alive

Objects

- Mutator is assumed correct
- References always point to heap, pointer stack is correct, etc
- Mutator wrong → crash

Sizes and optimal configuration

- Several important metrics
  - L = size of live objs
  - H = size of heap
  - D = size of dead
- L mostly static
- Most objects die young
- H=L*3 typical, H=L*5 often ideal

So wasteful!

- If (H >>> L), I’m wasting space!
- Problem of fairness
  - Can solve with IPC
- Memory is cheap
- Time is expensive

Tradeoffs

- You choose H, but not L
- H >>> L:
  - Less frequent GC
  - Mark-and-sweep: More time spent in GC (latency)
- H ≈ L:
  - Very frequent GC