Dynamic objects

The problem

- `l.next`
  - C: `l` is an instance of a `struct`, `next` is a field at a known offset
  - Java: `l` is a reference to a known `class`, `next` is a field at a known offset
  - Dynamic languages: `l` is a thing, lol, have fun
  - Also Java: `l` is a reference to a known `interface`

Whence an object?

- Dynamically: A map of field names to values
  - Remember: Values are blobs!
- Statically: A pointer to a data structure we define

Let’s build an object

```
struct Object {
    Map<string, blob> content;
};
```

- This works, but it’s hard to make it efficient
- Observation:
  Many objects, few ways of making them

Idea

- Separate *shape* (common) from *values* (per-instance)
- Make shapes quick to check/compare
- Make related objects share their shape
- Optimize by caching shape (*inline caching*)

Step one: Object

```
struct Object {
    Shape *shape; // Map<string,int> + more
    blob *content;
};
```

- Get/set: `o->content[o->shape->get(field)]`
- Add:
  - Reallocate content to get a new slot
  - Create/find a new shape with the new field->slot
  - Replace shape and content fields

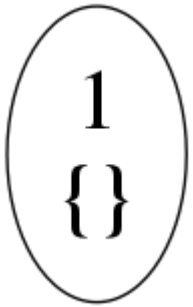
Step two: Shapes

- Want similar objects to have same shape
  - Same = Identical pointers

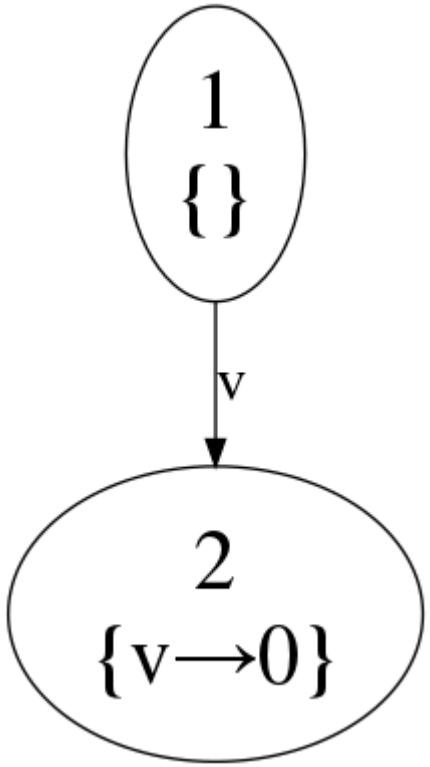
```
struct Shape {
    Map<string, int> slots;
    /* Shapes form a tree in which
     * steps are field additions */
    Map<string, Shape*> transitions;
};
```

Adding fields

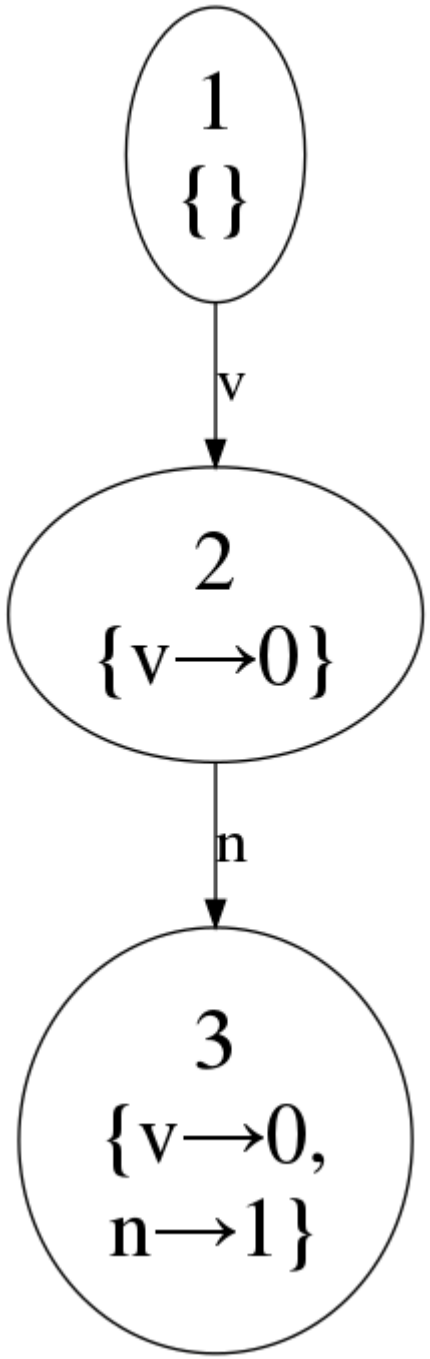
- Keeping transitions in shapes means shapes form a tree
- Add same fields → get same shape



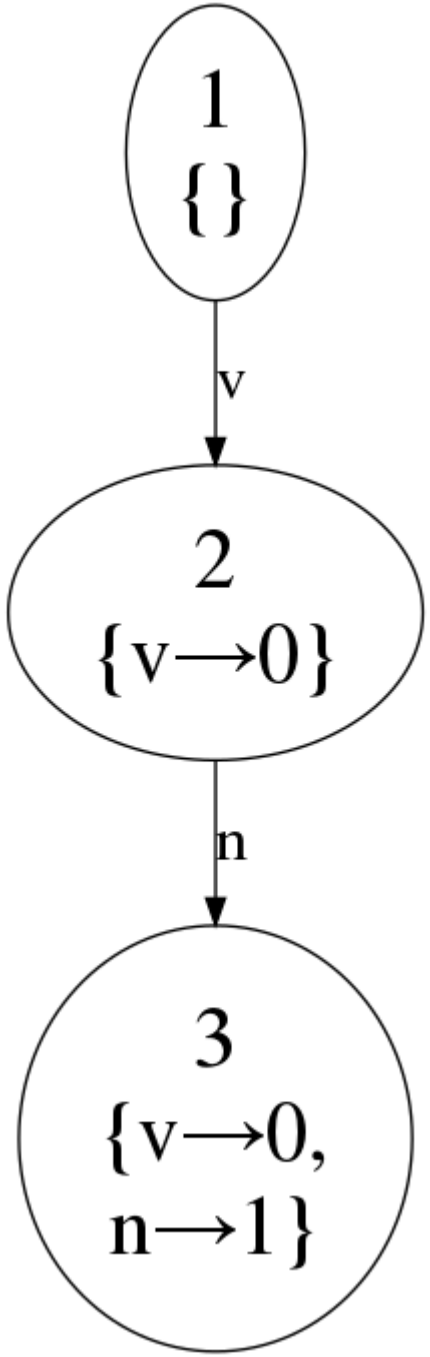
// l's shape is 1, m's shape is 1
// i.e., they both have no fields
[d] l.v = 42;



// l's shape is 1, m's shape is 1
// i.e., they both have no fields
[d] l.v = 42;
[h] allocate new shape (2)
[h] add "v"→idx 0 shape 2
[h] add "v"→shape 2 to shape 1
[h] extend l->content with 42
[h] l->shape = 2
[d] l.n = m;



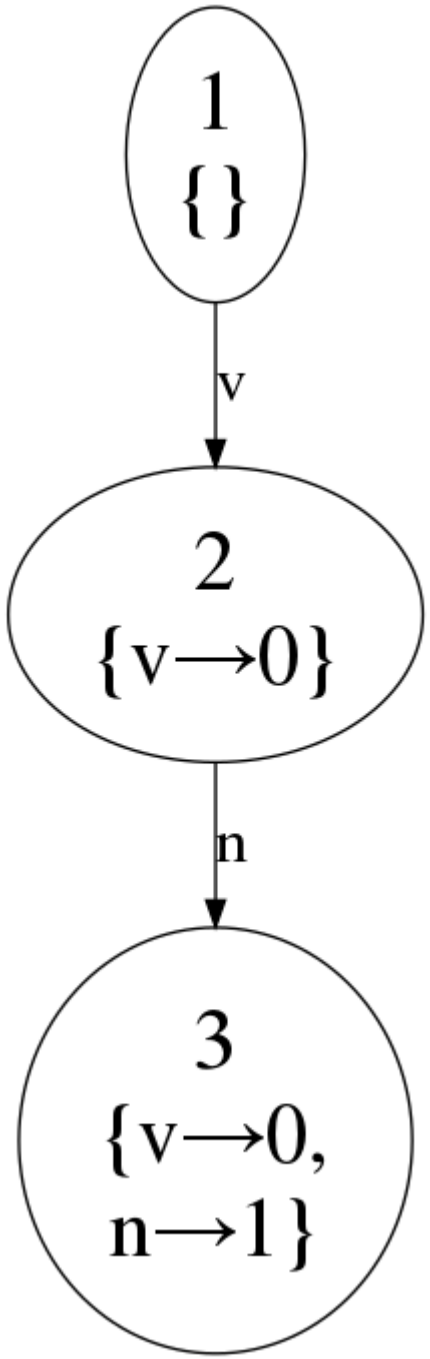
// l's shape is 1, m's shape is 1
// i.e., they both have no fields
[d] l.v = 42;
[h] allocate new shape (2)
[h] add "v"→idx 0 shape 2
[h] add "v"→shape 2 to shape 1
[h] extend l->content with 42
[h] l->shape = 2
[d] l.n = m;
[h] (same steps, shape 3)
[d] m.v = 12; m.v = null;



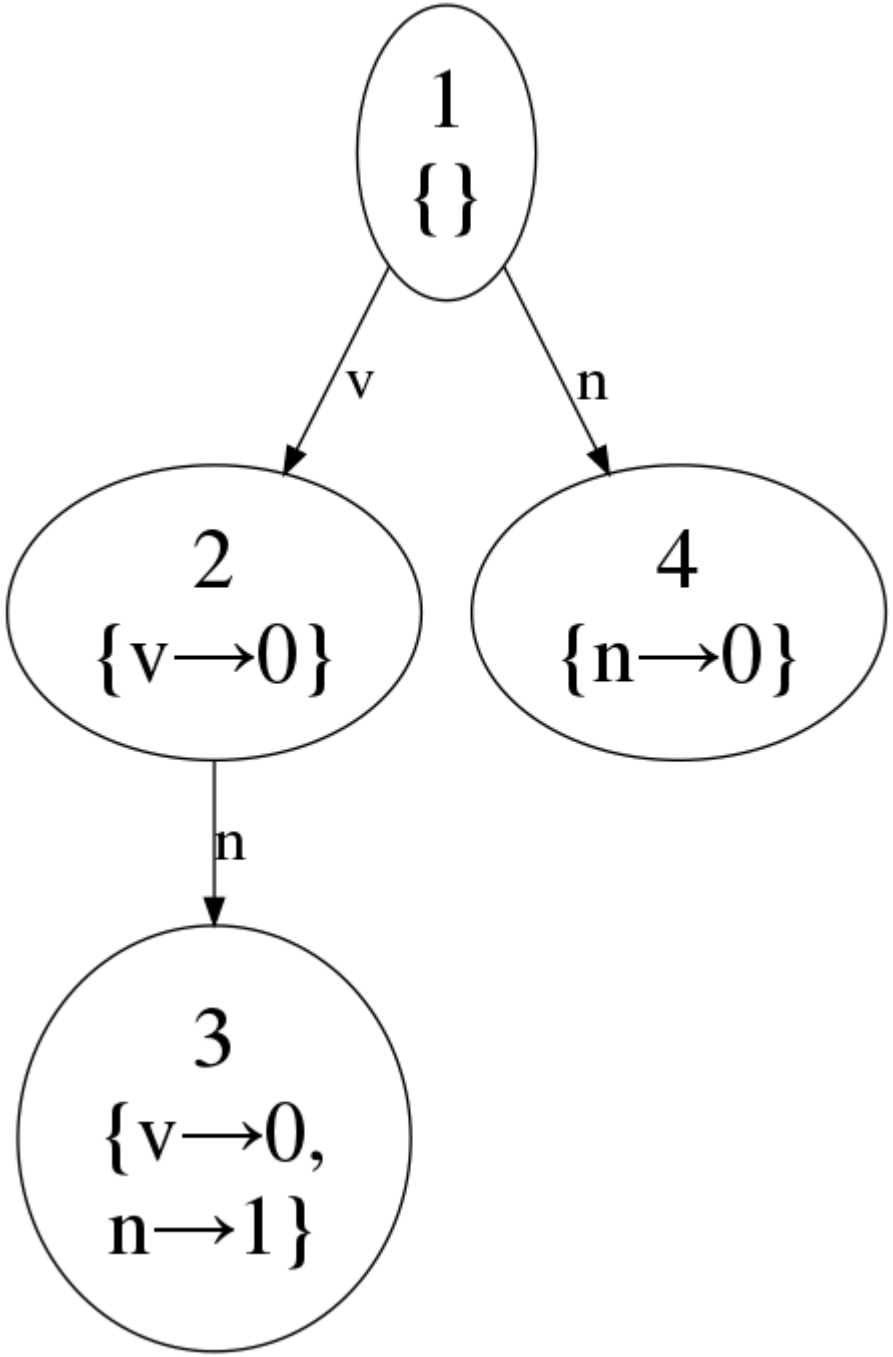
// l's shape is 1, m's shape is 1
// i.e., they both have no fields

[d] l.v = 42;
[h] allocate new shape (2)
[h] add "v"→idx 0 shape 2
[h] add "v"→shape 2 to shape 1
[h] extend l->content with 42
[h] l->shape = 2
[d] l.n = m;
[h] (same steps, shape 3)
[d] m.v = 12; m.n = null;
[h] extend m->content,
    m->shape = 3

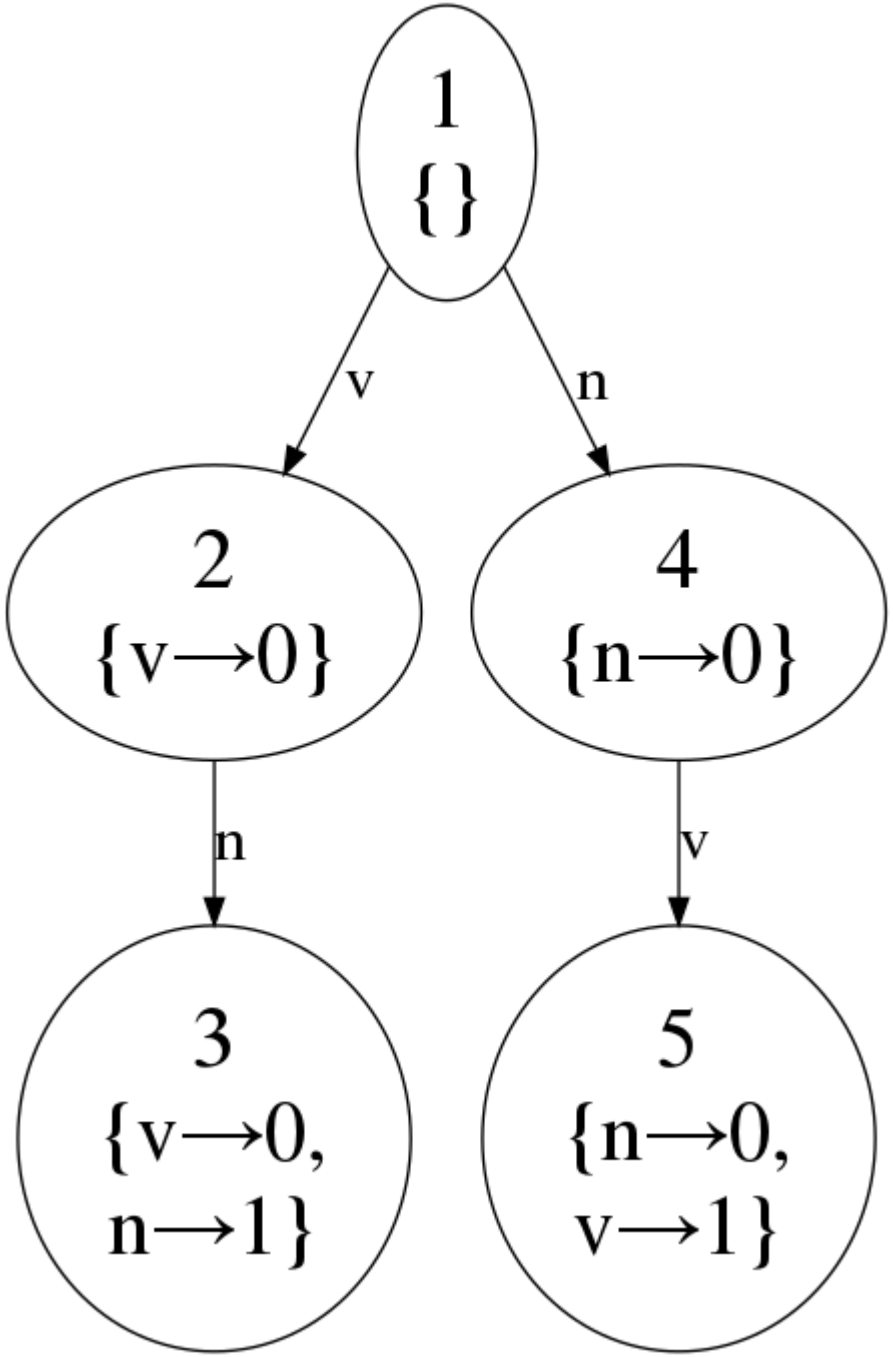
l->shape == m->shape



// l's shape is 3, m's shape is 3
// i.e., they both have v and n
// o's shape is 1
[d] o.n = 42;



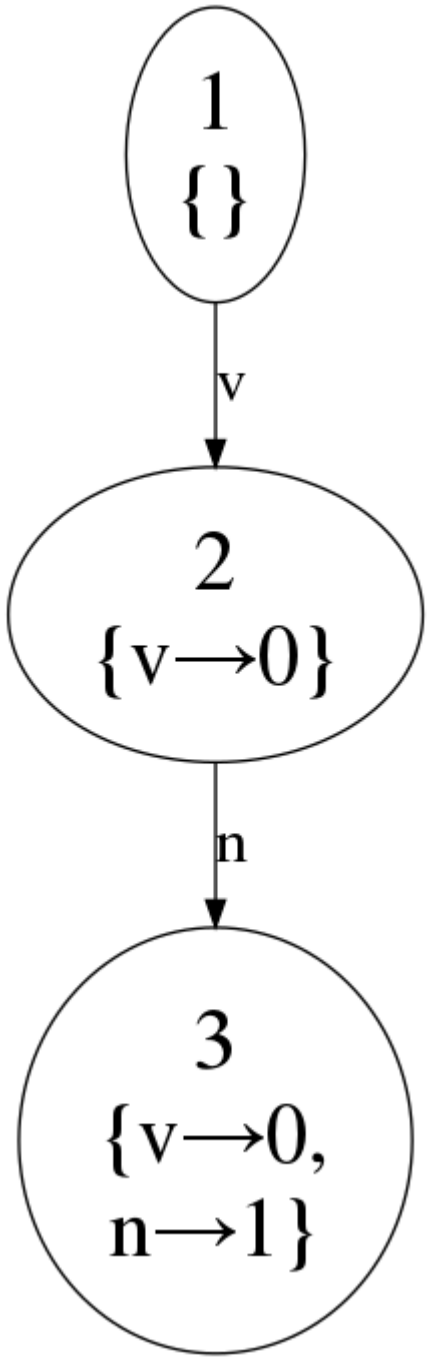
// l's shape is 3, m's shape is 3
// i.e., they both have v and n
// o's shape is 1
[d] o.n = 42;
[h] (same steps, new path)
[d] o.v = 12;



// l's shape is 3, m's shape is 3
// i.e., they both have v and n
// o's shape is 1
[d] o.n = 42;
[h] (same steps, new path)
[d] o.v = 12;
[h] (same steps)

- Even though l and o have the same fields, they have different shapes because they were added through different paths

SDyn



// l's shape is 3, m's shape is 3
// i.e., they both have v and n
l->shape == m->shape

- If we cache l->shape,
- check the shape at runtime,
- generate code with field locations burned in,
- then pass in m,
- it'll be fast!

Inline caching

- Code for reading a field:

```
static Shape *cachedShape;
static int cachedSlot;
if (o->shape != cachedShape) {
    cachedShape = o->shape;
    cachedSlot = o->shape->get(field);
}
return o->content[cachedSlot];
```

Inline caching

- We can also cache shape transitions
  - (If the cached shape doesn't have the field, change both shape and content)
- Most VM's implement a *polymorphic* inline cache
  - (Just means we cache multiple shape/slot pairs)

SDyn

GC background

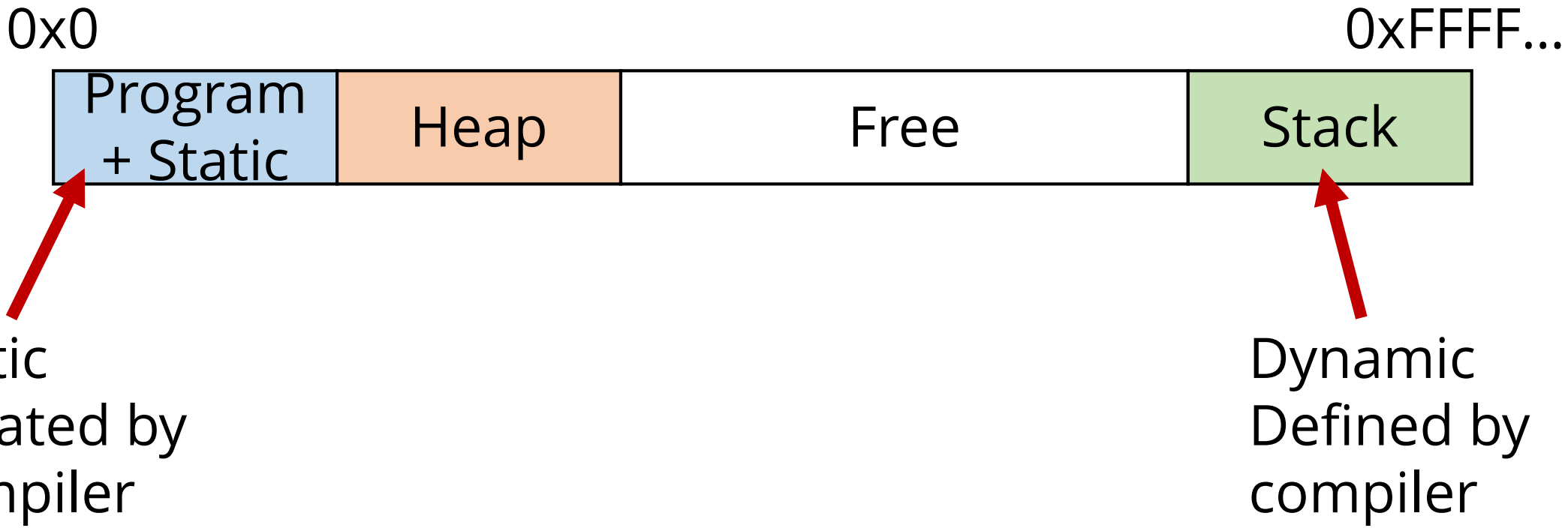
Manual memory management

Memory

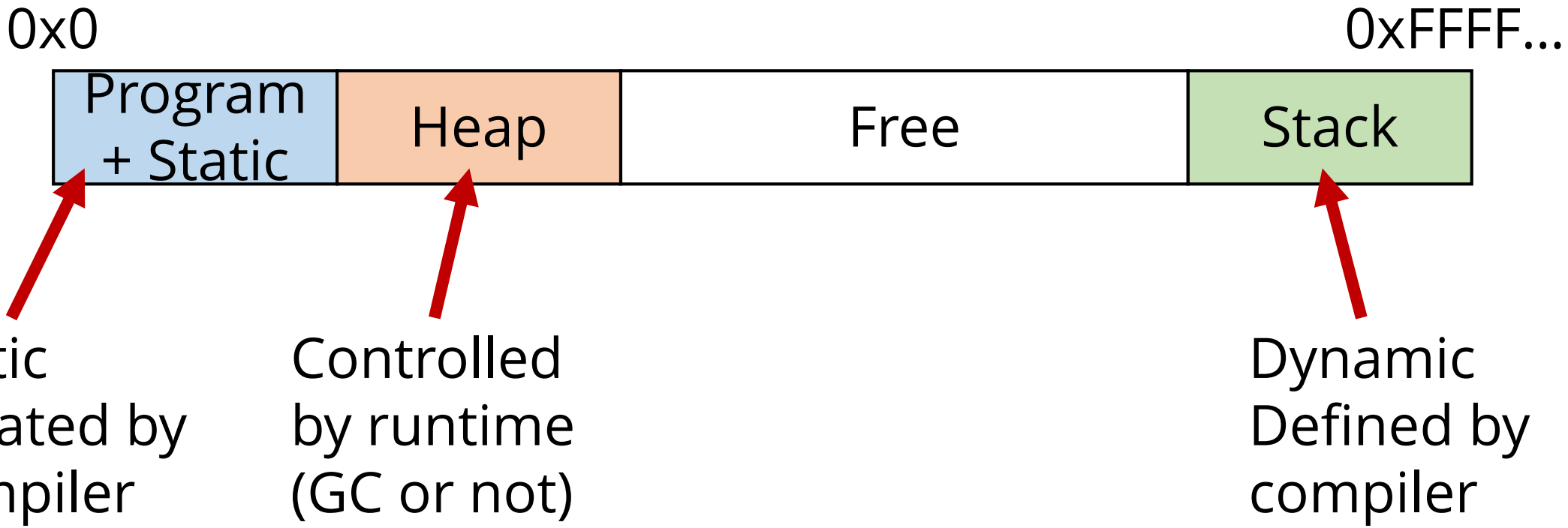
Memory

Static
Created by compiler

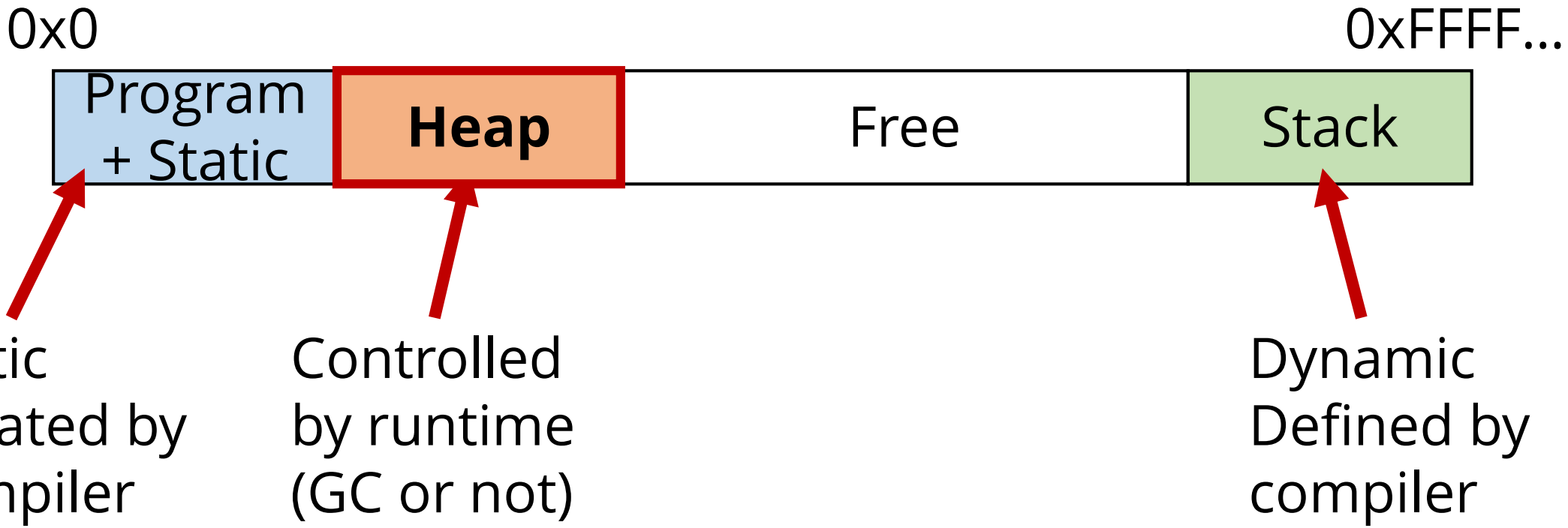
Memory


0x0
0xFFFF...
Program + Static
Heap
Free
Stack
Static
Created by compiler
Dynamic
Defined by compiler

Memory


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

Memory


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

Virtual memory

- Memory isn’t memory!
- Page tables give protection + control
- Not direct-to-RAM: Flexibility in allocation

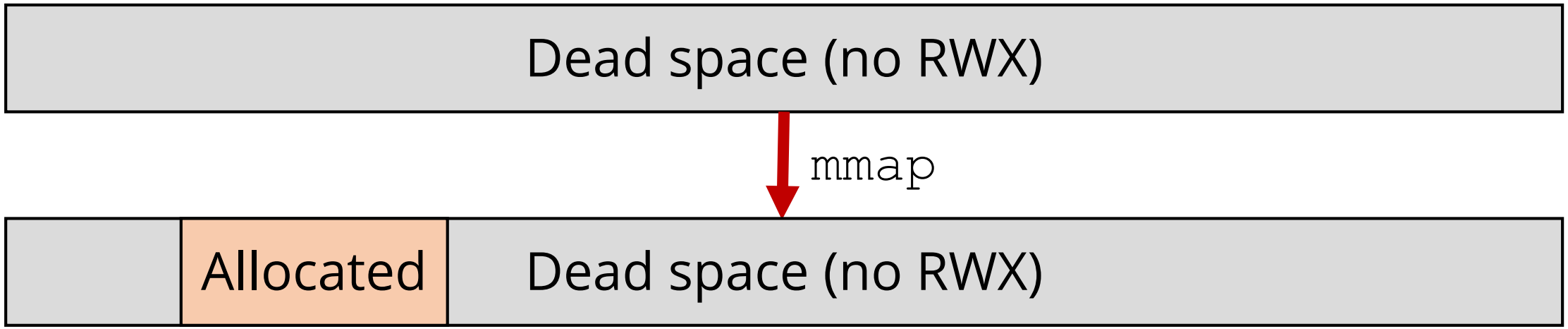
Manual memory

- `malloc(size)`:
  Returns pointer to `size` bytes of memory
- `free(ptr)`:
  Frees space returned by `malloc`
- Presents the illusion of “objects”
- Internally, much more going on!

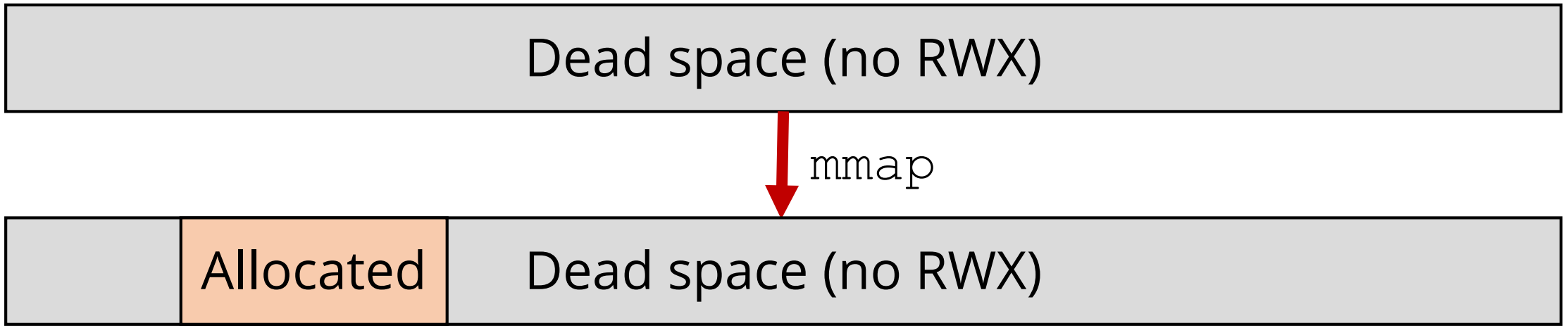
Memory to the manager

Dead space (no RWX)

Memory to the manager

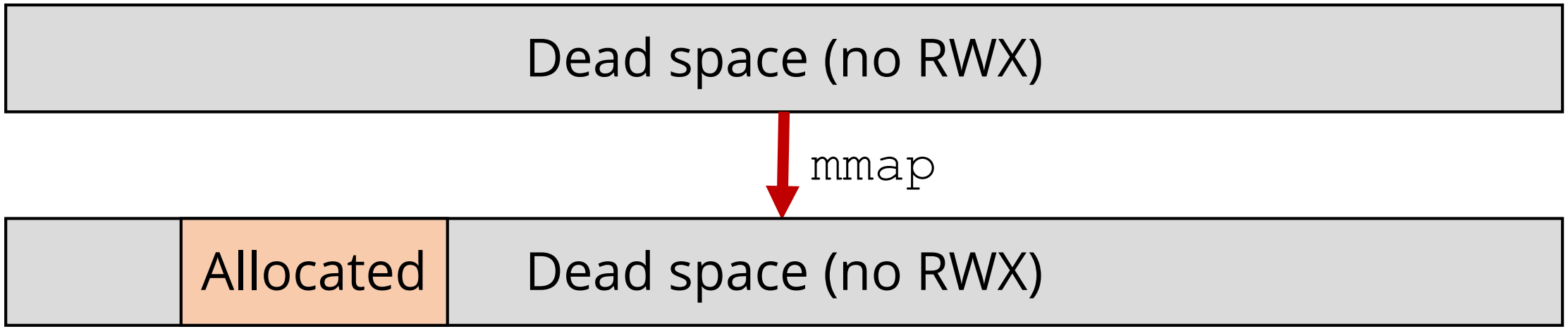


Memory to the manager

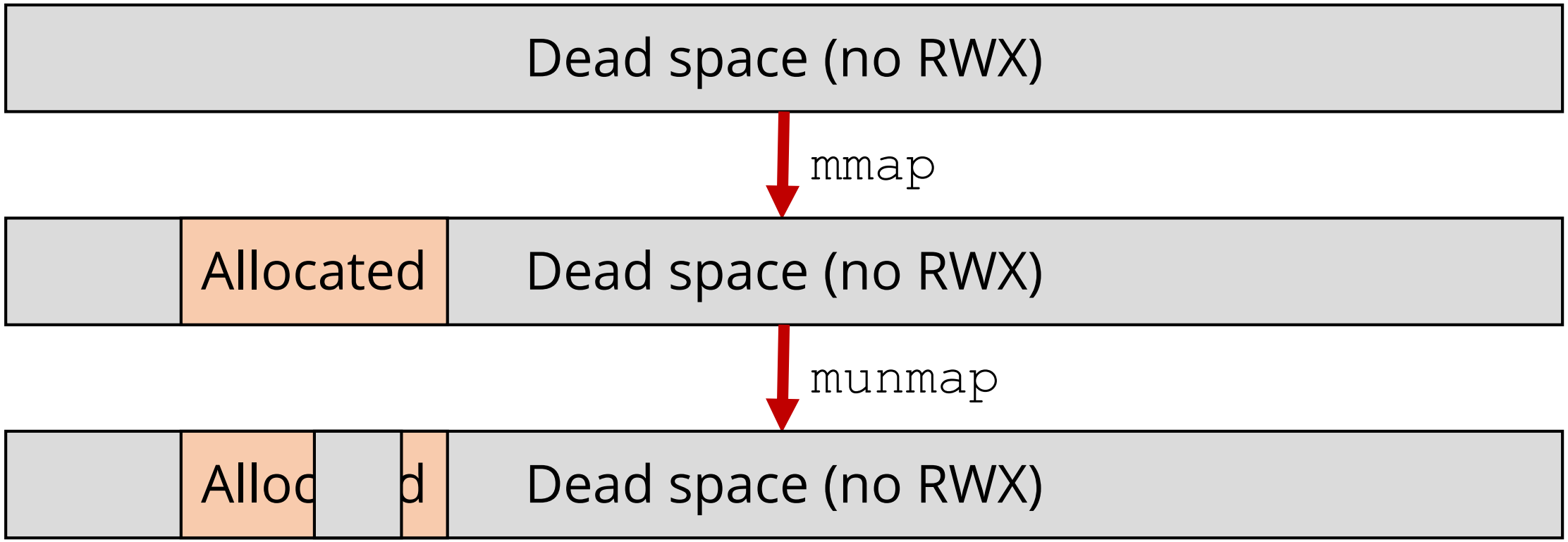


- `mmap` dumbly modifies page table:
  - No memory of its own changes
  - No object illusion

Memory to the manager



Memory to the manager



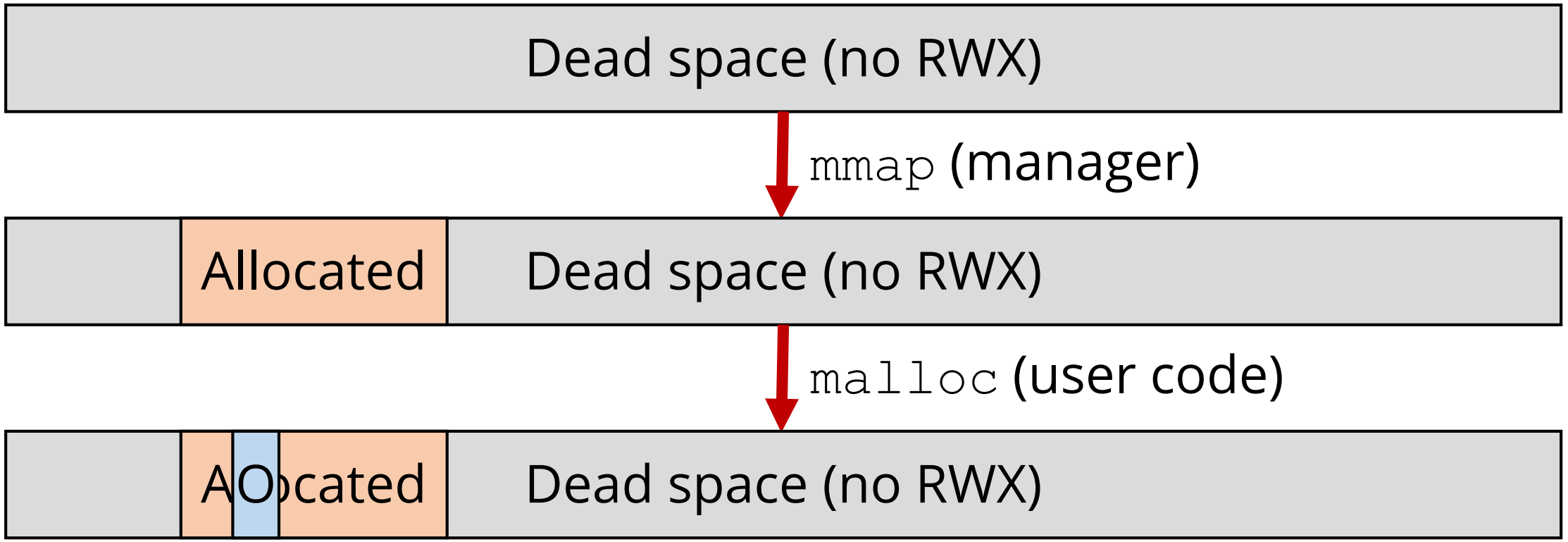
Memory to the manager

- Big chunks of free space
- Manager chooses size
- Manager must remember where
- Chicken and egg: Need static space for pointers to allocated space

Pools

- Keep track of memory in “pools”
  - (Typically) Fixed size
  - Maintained in list and/or (static) array
- Manager gives memory from pools
  - Manager implements object illusion

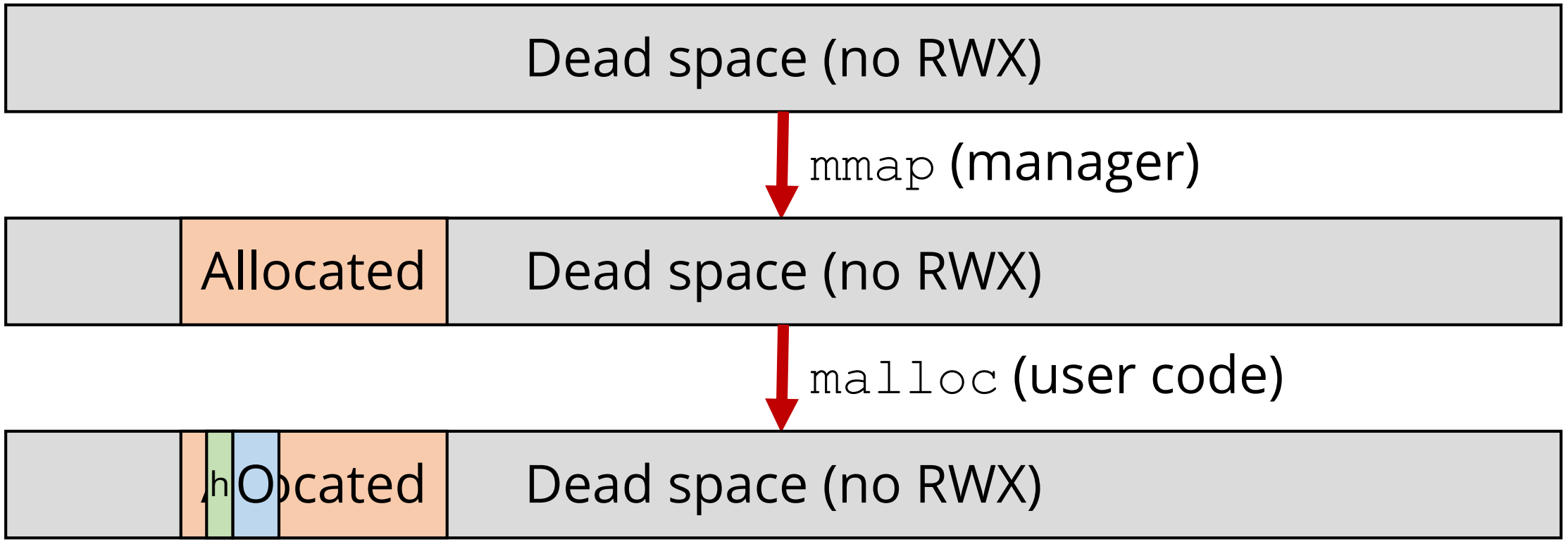
Memory to the manager



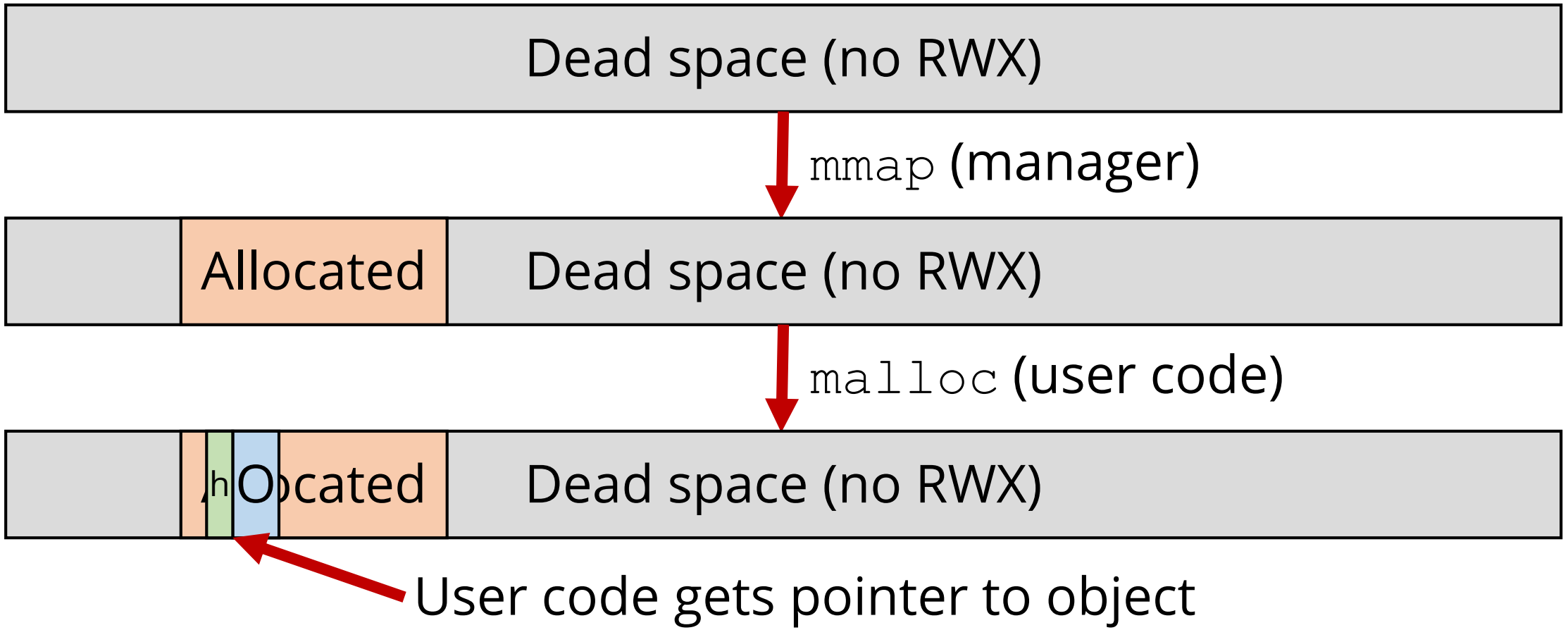
Object illusion

- User code: Pointer is sufficient info
- Manager: Need to know location and size
- Solution: Object headers

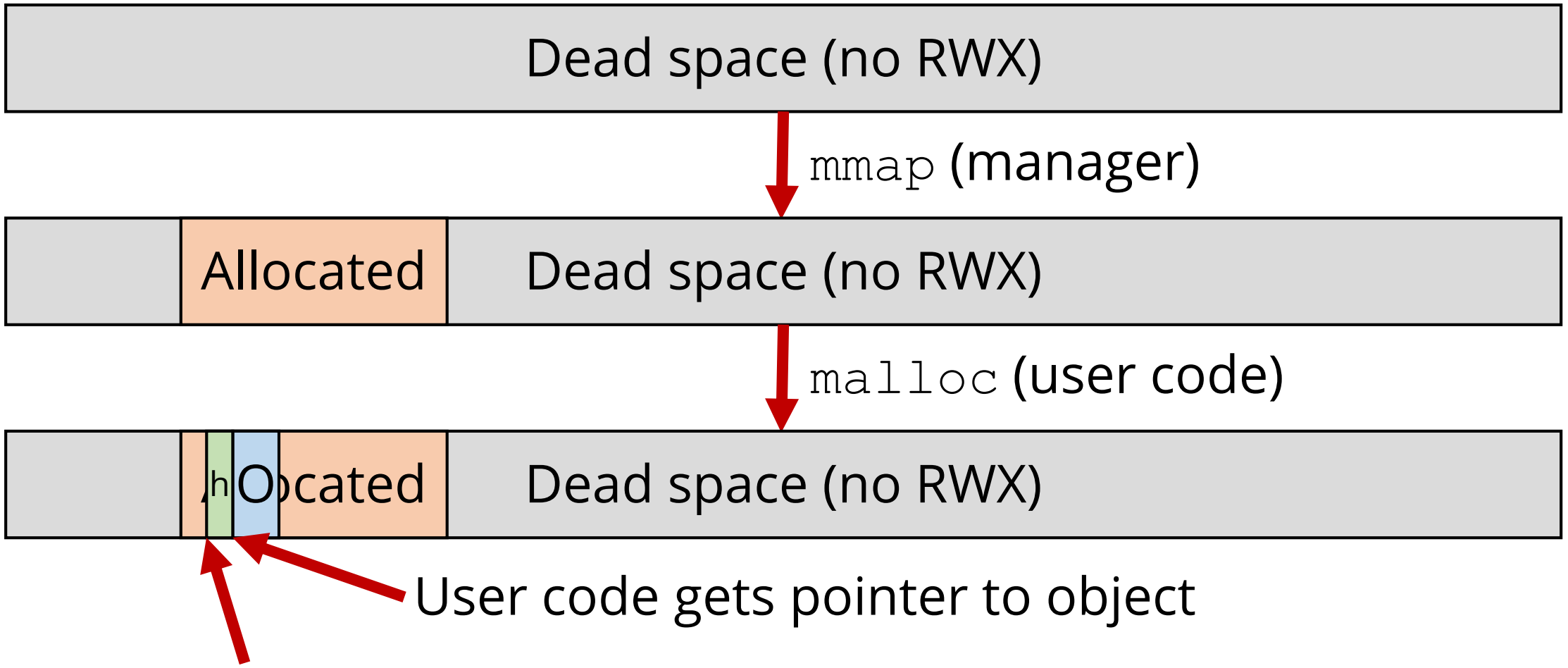
Memory to the manager



Memory to the manager



Memory to the manager



Object illusion

```
struct ObjectHeader {
  size_t objectSize;
};
```

…

```
((struct ObjectHeader *) someObject)[-1].objectSize
```

Object illusion

```
struct ObjectHeader {
   size_t objectSize;
};
```

Must at least remember size

…

```
((struct ObjectHeader *) someObject)[-1].objectSize
```

The simplest manager

- `malloc(size)`:
  Call `mmap` to allocate `size + sizeof(ObjectHeader)` bytes, put size in header, give pointer to space after header

- `free(ptr)`:
  Use object header to get size, call `munmap`

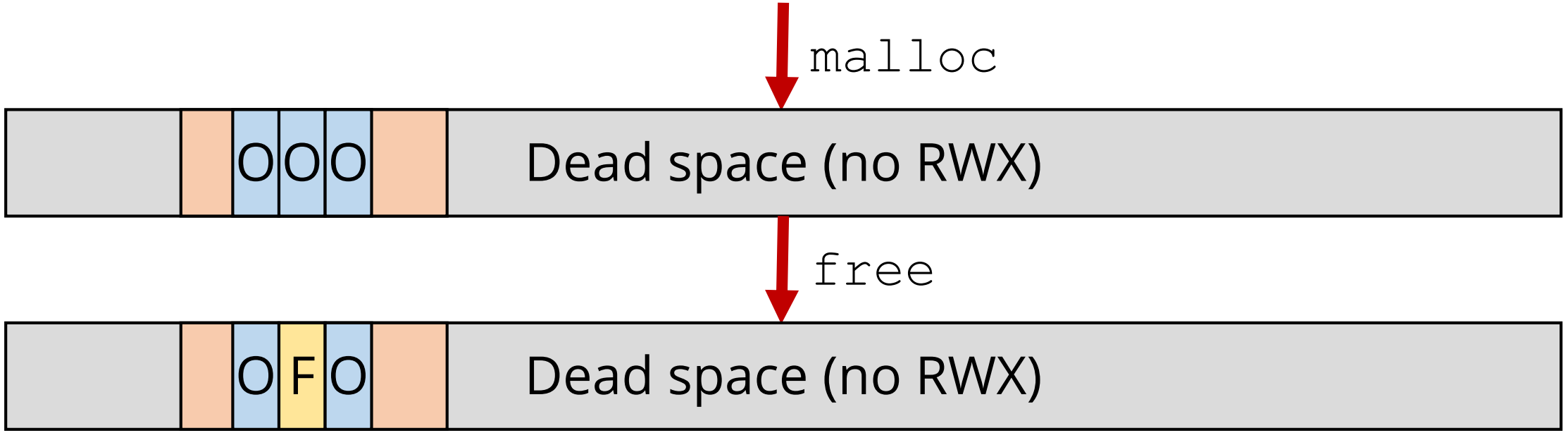
The stupidest manager

- `mmap` works in pages (usu 4096 bytes)
- Most objects much smaller
- This is why we need pools!

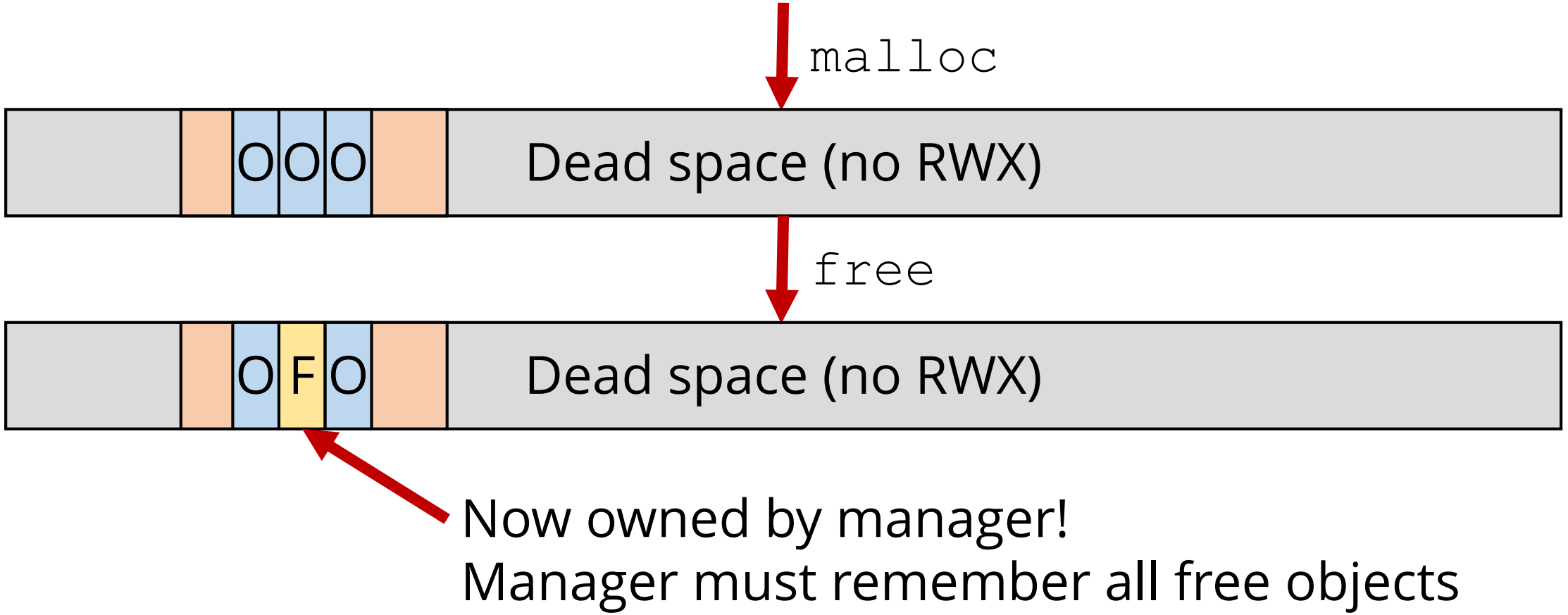
Real management

- Manager must break pool into objects
- `free` can no longer return space to OS
- Manager must keep track of `free`'d space
- Concept of "free object"

Memory to the manager



Memory to the manager



Free objects

- Keep on “free list”
- List head pointer at beginning of pool
- List next pointer in free objects

Free objects

```
struct ObjectHeader {
  size_t objectSize;
};

struct FreeObject {
  struct FreeObject *next;
};

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

Free objects

```
struct ObjectHeader {
  size_t objectSize;
};

struct FreeObject {
  struct FreeObject *next;
};

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

All objects, including free ones, have an object header, so don't need size here!

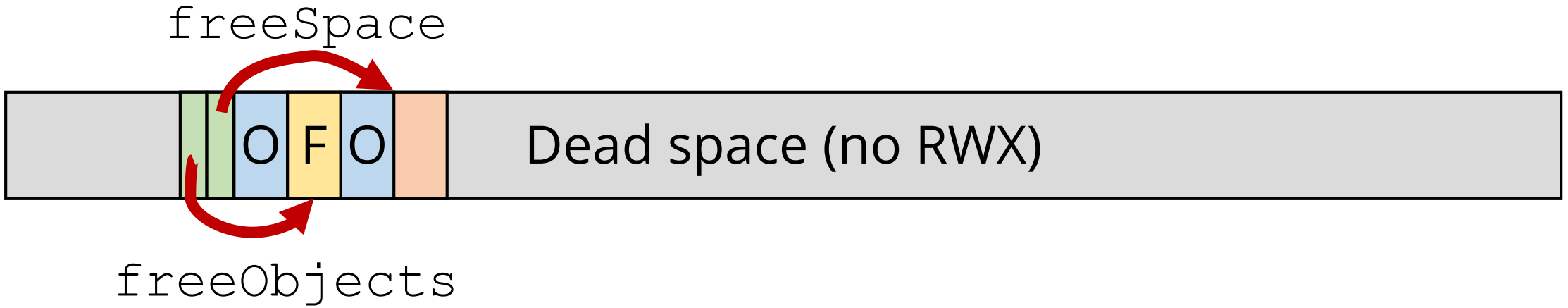
Free objects

```
struct ObjectHeader {
  size_t objectSize;
};

struct FreeObject {
  struct FreeObject *next;
};

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

All objects, including free ones, have an object header, so don't need size here!

This struct defines the static data in a pool: Remaining space is for allocated/free objects

Memory to the manager

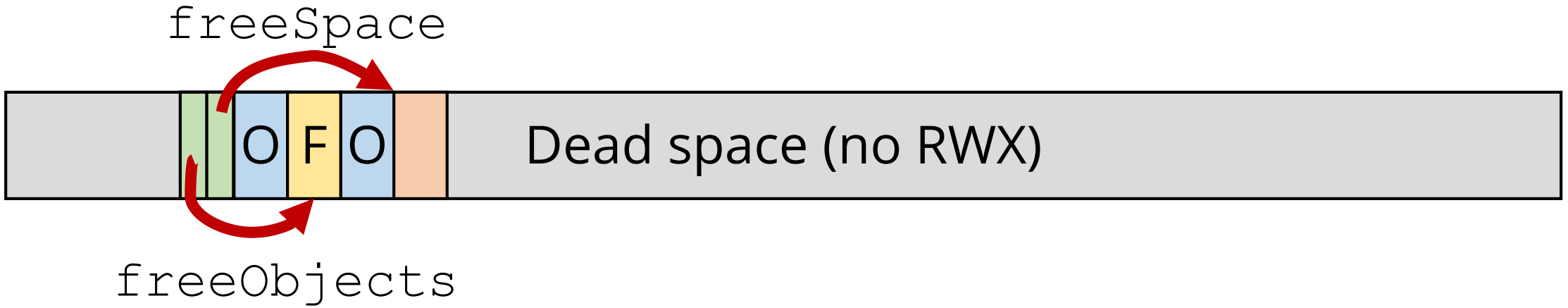


Allocation

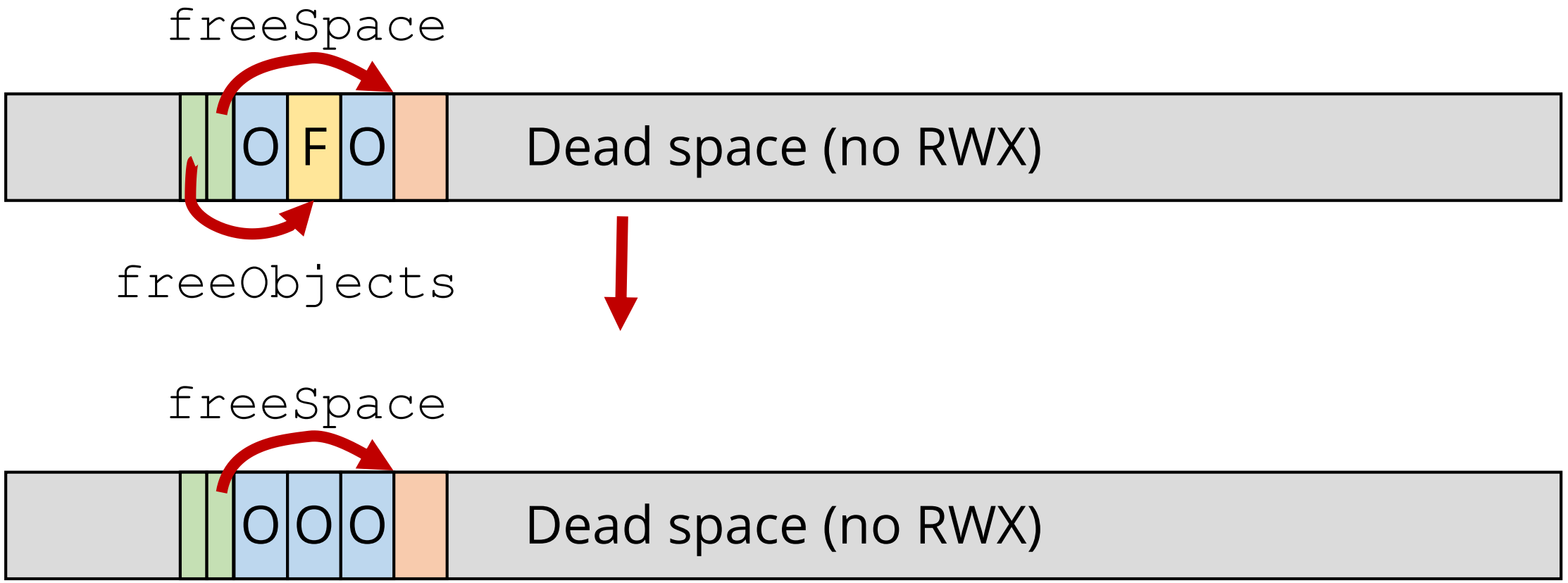
- With free list:
  - First try to find a suitable object[1] on the free list
  - If found, remove from free list and return
  - If not found, allocate new object from free space
  - If no free space, allocate new pool

[1] This process is extremely complicated

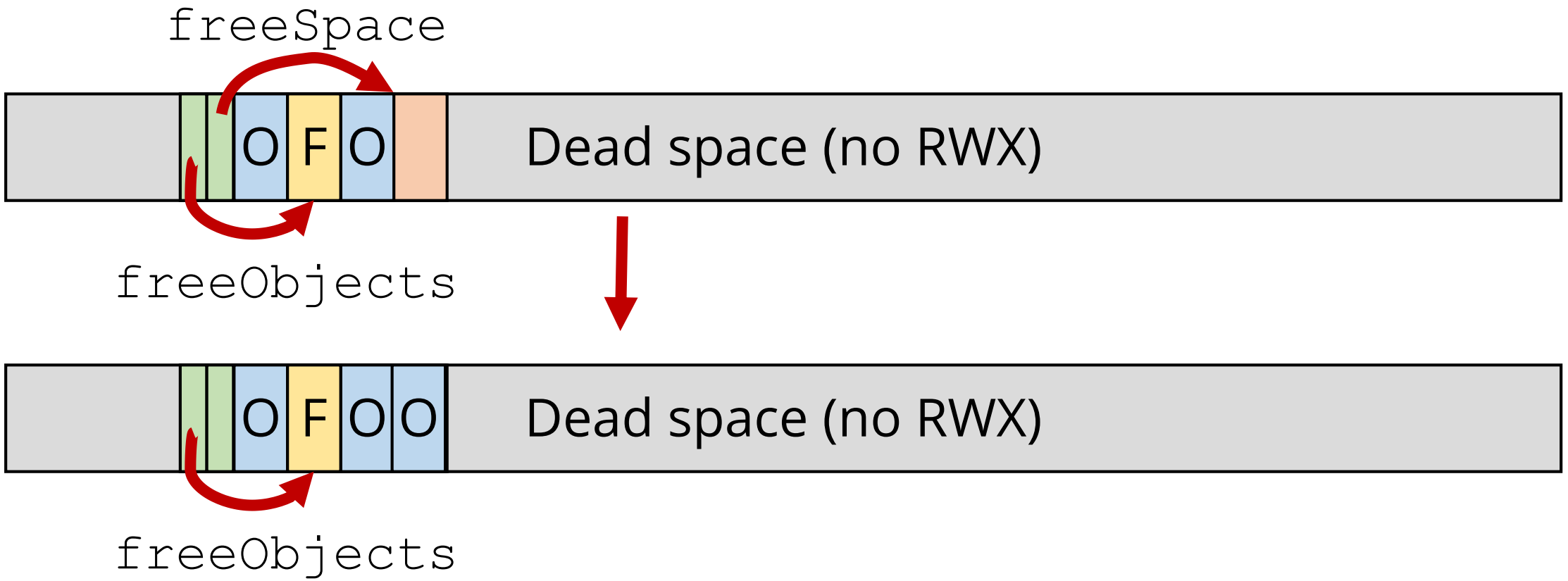
Memory to the manager



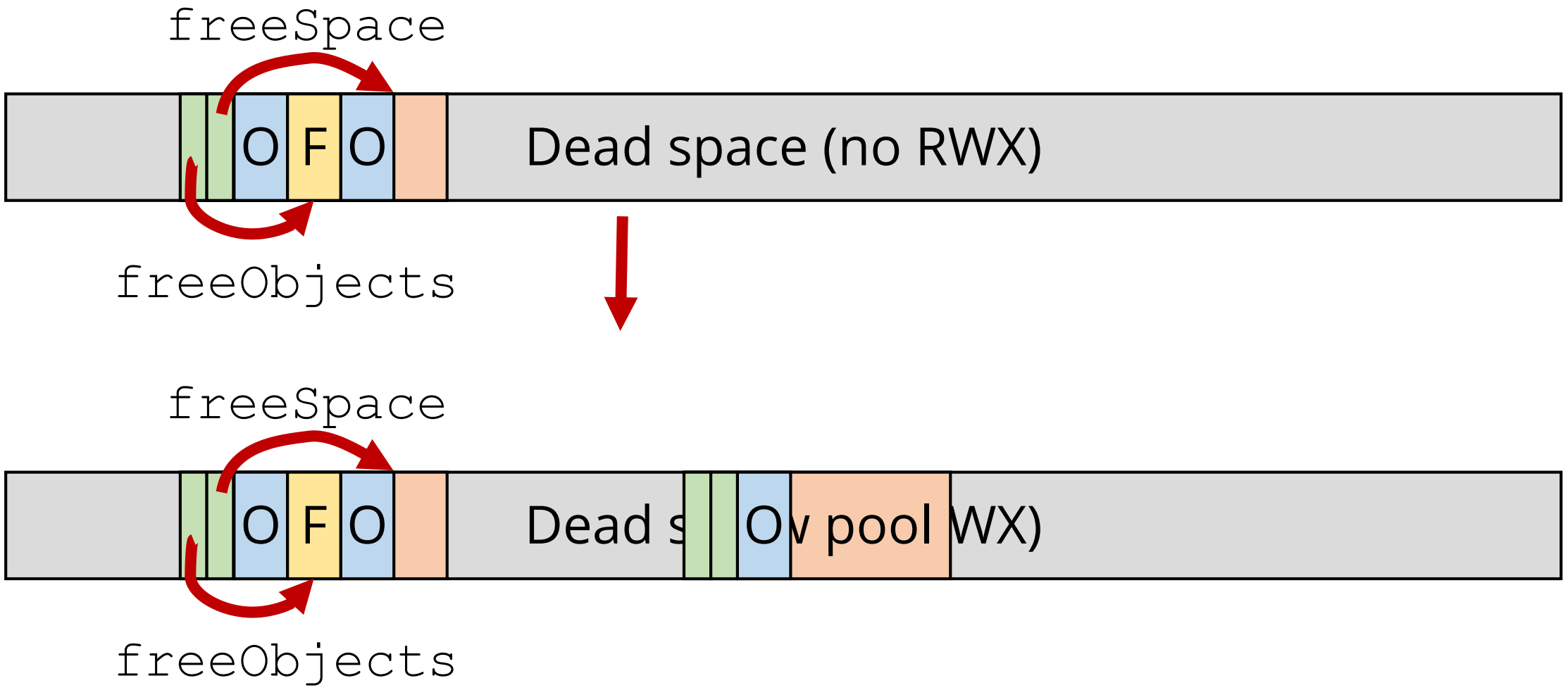
Memory to the manager



Memory to the manager



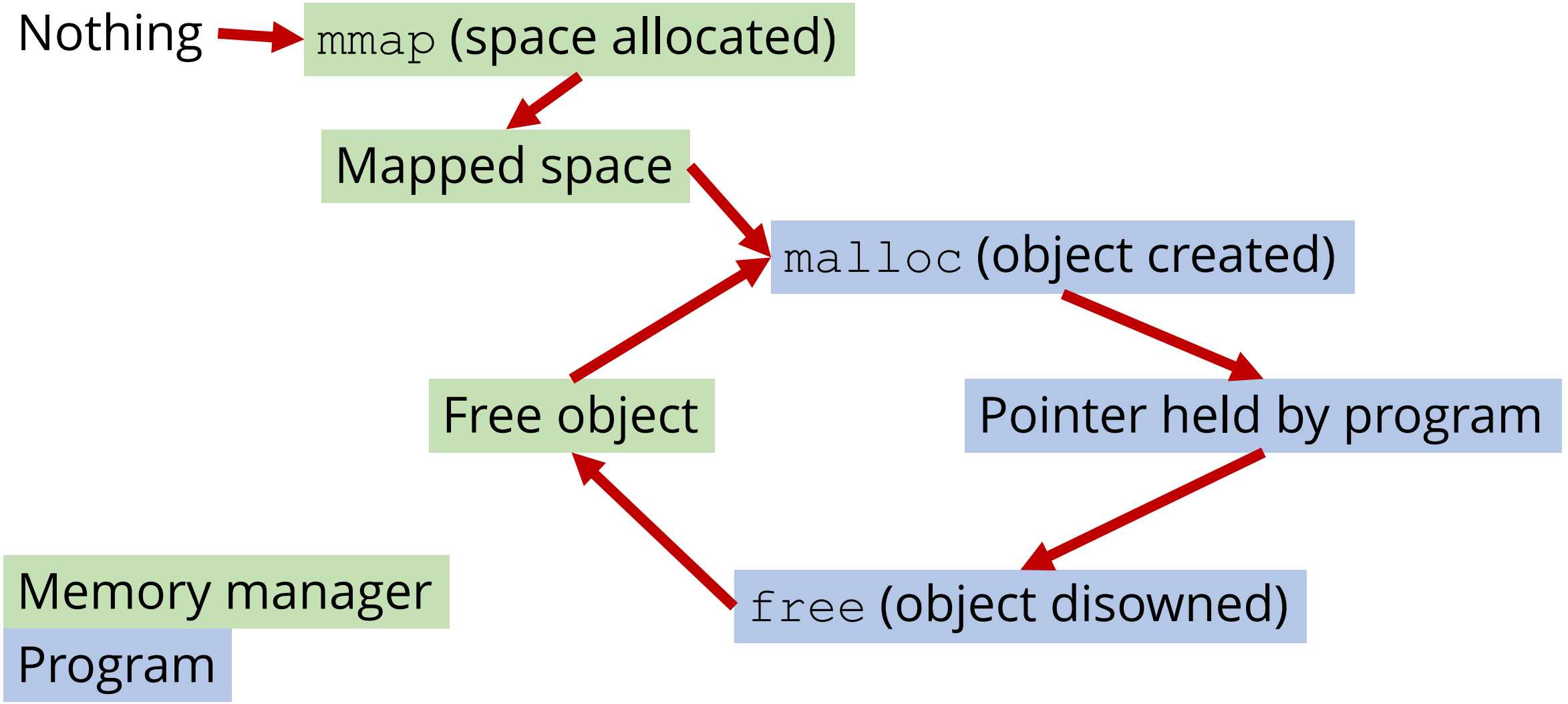
Memory to the manager



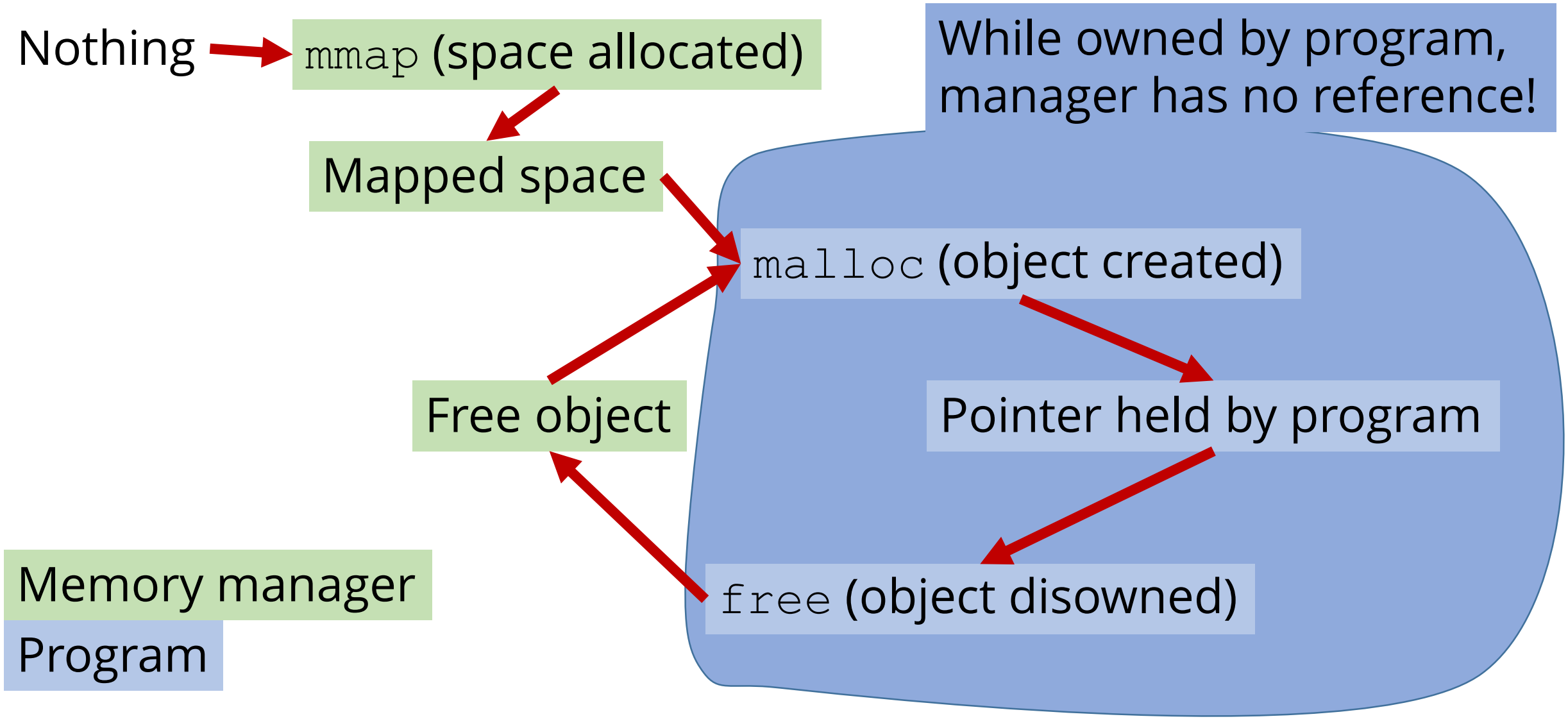
Considerations

- When object is allocated, manager has no pointer
- When object is free, not given back to OS
- Hardware, OS and manager all distinct

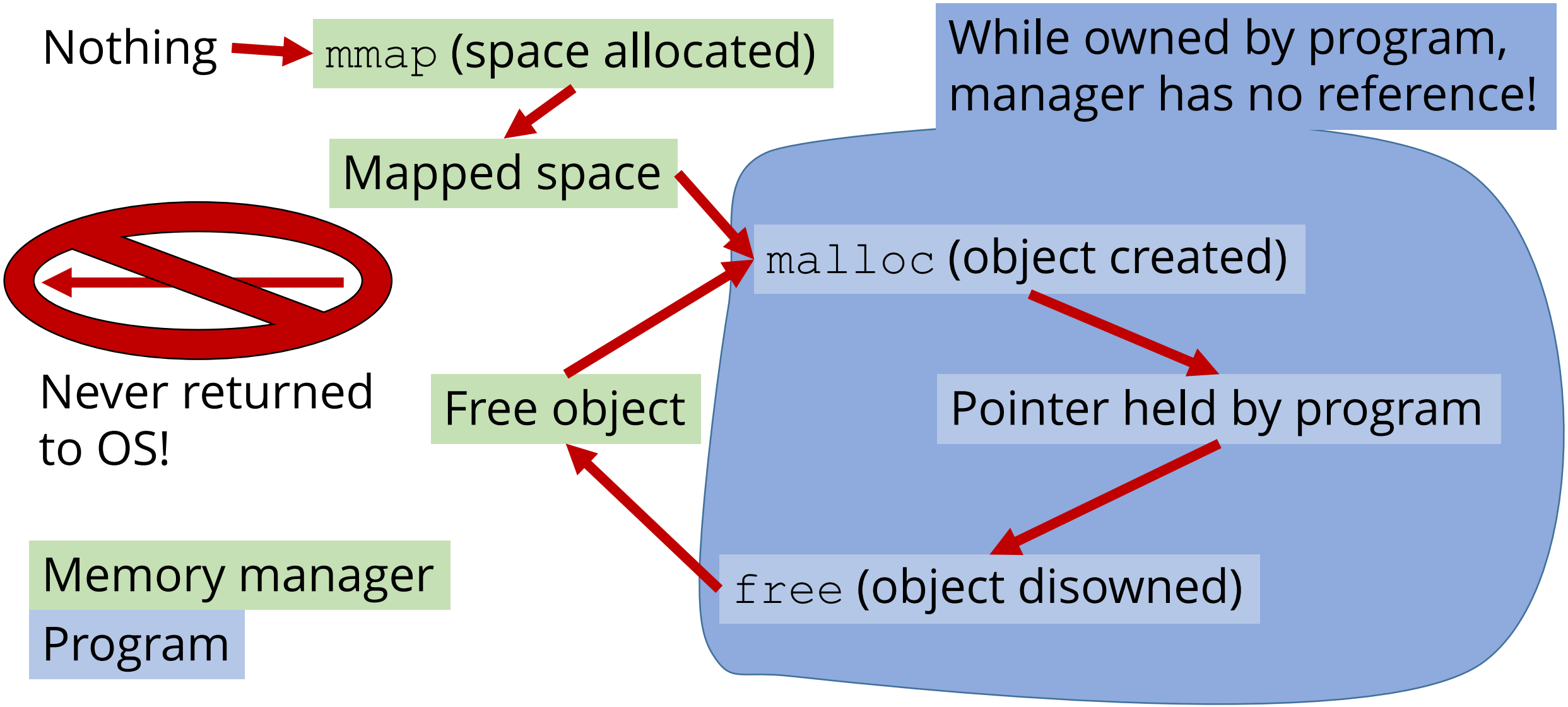
The life of a pointer



The life of a pointer



The life of a pointer



Pools

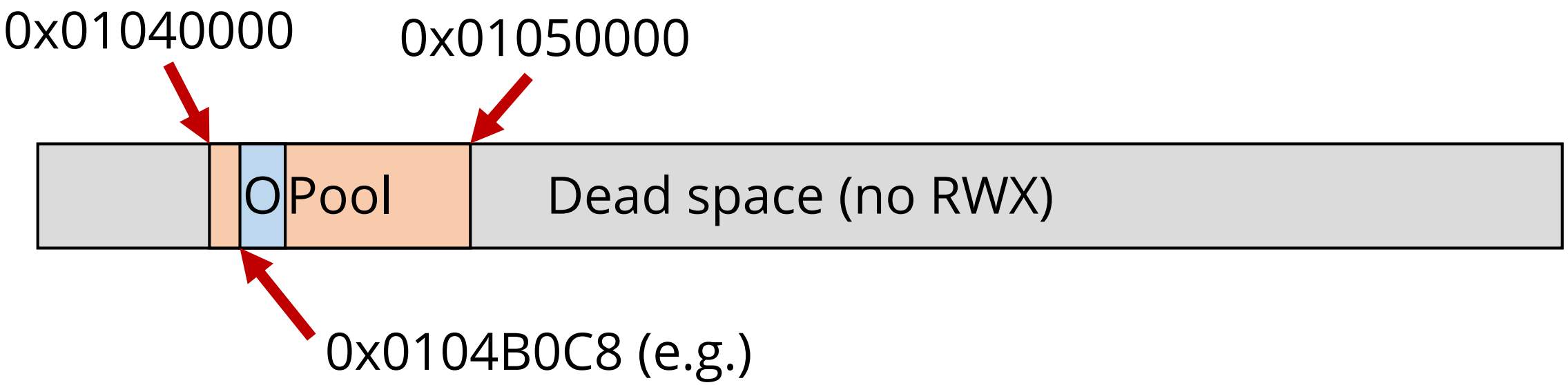
- We may have multiple pools
- Free list per pool or global?
  - If per pool: How to get from `free(o)` to pool?
  - If global: Thread contention ☹

Alignment

- Alignment allows magic with pointers!
- Remember: We can control *where* pools are mapped

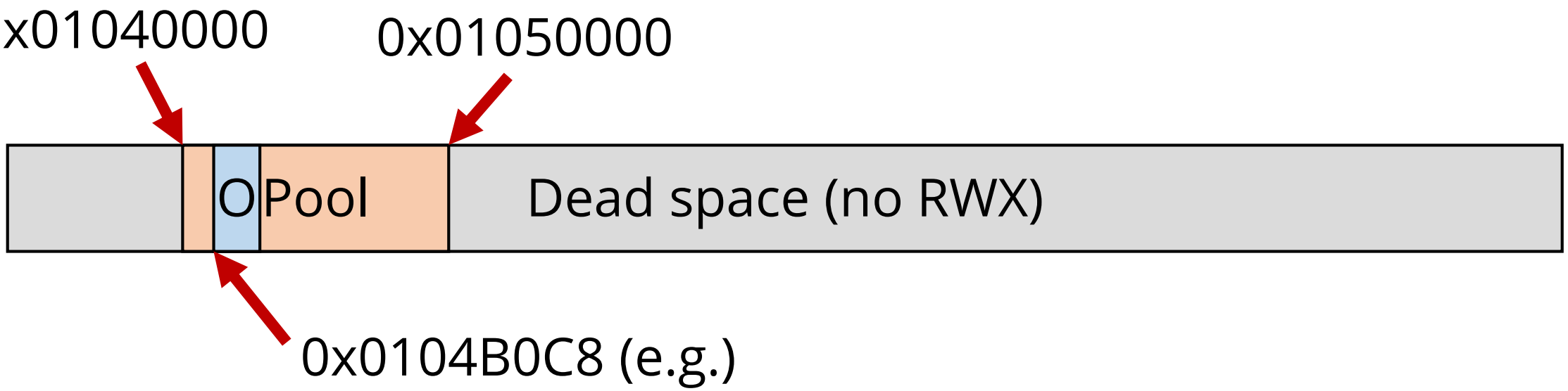
Alignment

Ex: Pools aligned to multiples of 0x00010000:



Alignment

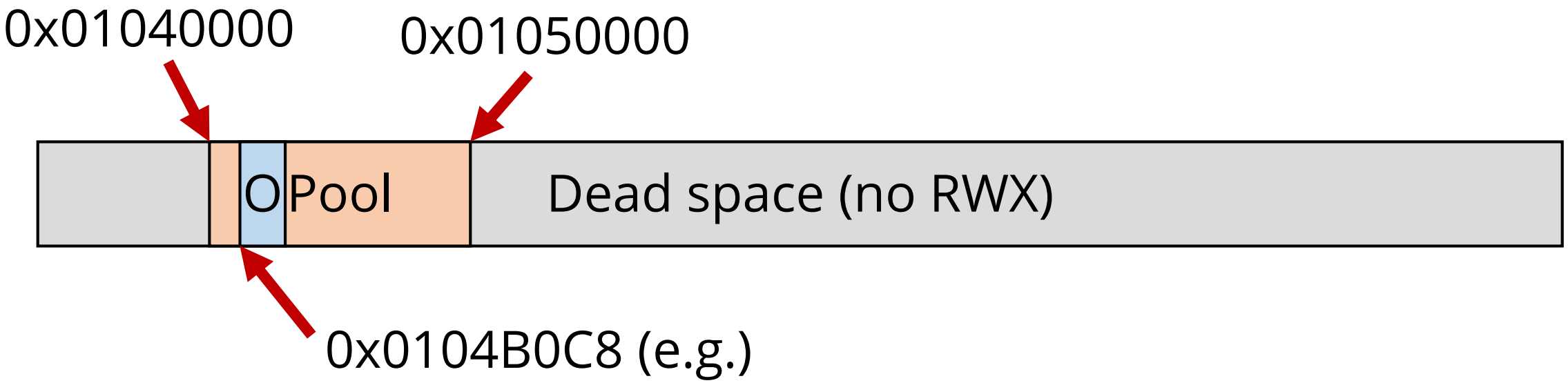
Ex: Pools aligned to multiples of 0x00010000:



“Pool mask”: 0xFFFF0000

Alignment

Ex: Pools aligned to multiples of 0x00010000:

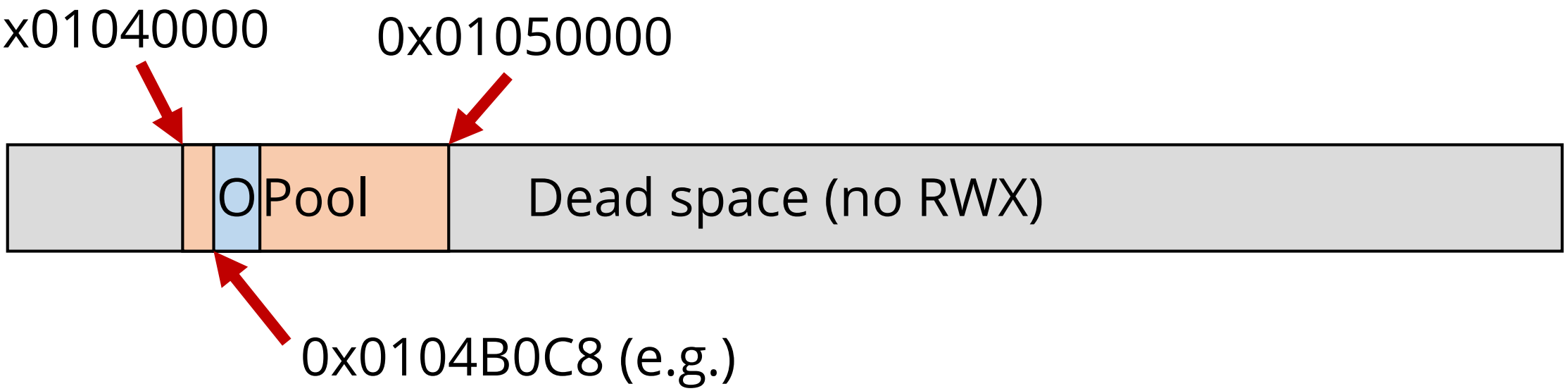


“Pool mask”: 0xFFFF0000

```
(0x0104B0C8 & 0xFFFF0000) == 0x01040000
```

Alignment

Ex: Pools aligned to multiples of 0x00010000:



“Pool mask”: 0xFFFF0000

```
(0x0104B0C8 & 0xFFFF0000) == 0x01040000
(struct Pool *) ((size_t)  p & POOL_MASK)
```

```
void free(void *o) {
  struct FreeObject *fo = (struct FreeObject *) o;
  struct ObjectHeader *oh = &((struct ObjectHeader *) o)[-1];
  struct Pool *p = (struct Pool *) ((size_t) o & POOL_MASK);

  fo->next = p->freeList;
  p->freeList = o;
}
```