Introduction to GC

CS842:
Automatic Memory Management
and Garbage Collection

Course

- This is a coding course
- You will implement several garbage collectors
- Short presentations of recent topic
- http://the.gregor.institute/t/c/

Reqs and grading

- Compilers and OS background helpful
- `((struct GC_Pool *) (((size_t) &p) & 0xFFFFF000))`
    - 60%: Projects (code)
    - 15%: Presentation
    - 25%: Final

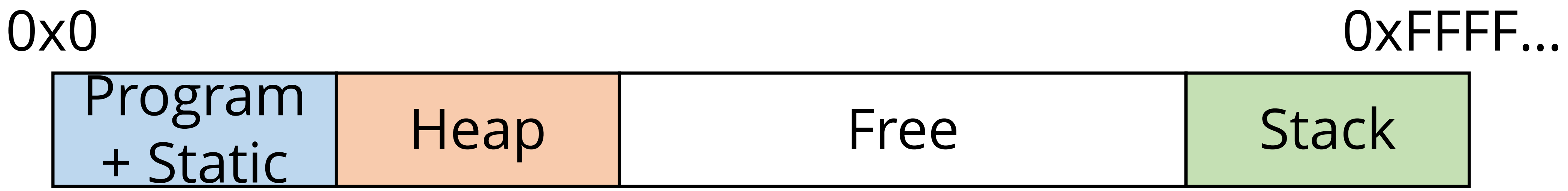
Schedule (preliminary)

	M	W
Sept 14	Intro/Background	Basics/ideas
Sept 21	Allocation/layout	GGGGC
Sept 28	Mark/Sweep	Mark/Sweep
Octo 5	Copying GC	Ref C
Octo 12	Mark/Compact	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

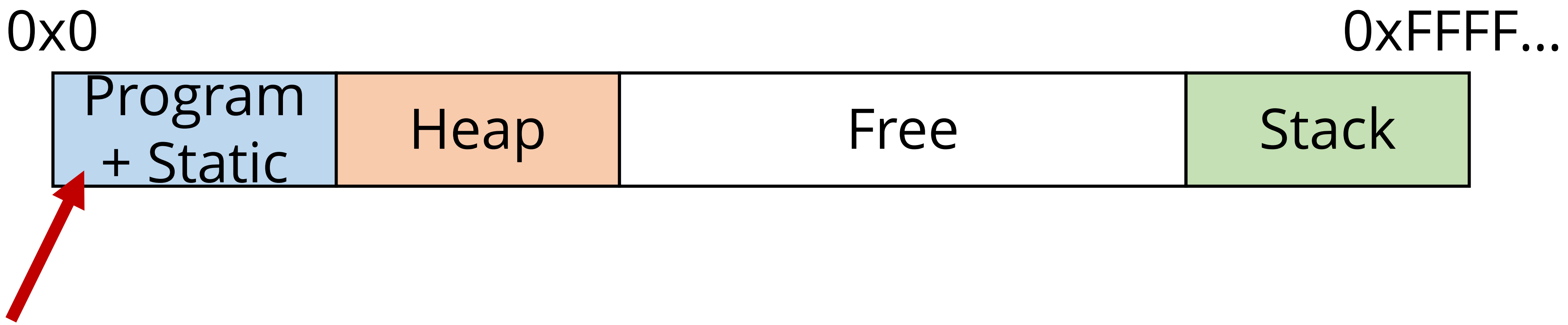
Background

Manual memory management

Memory

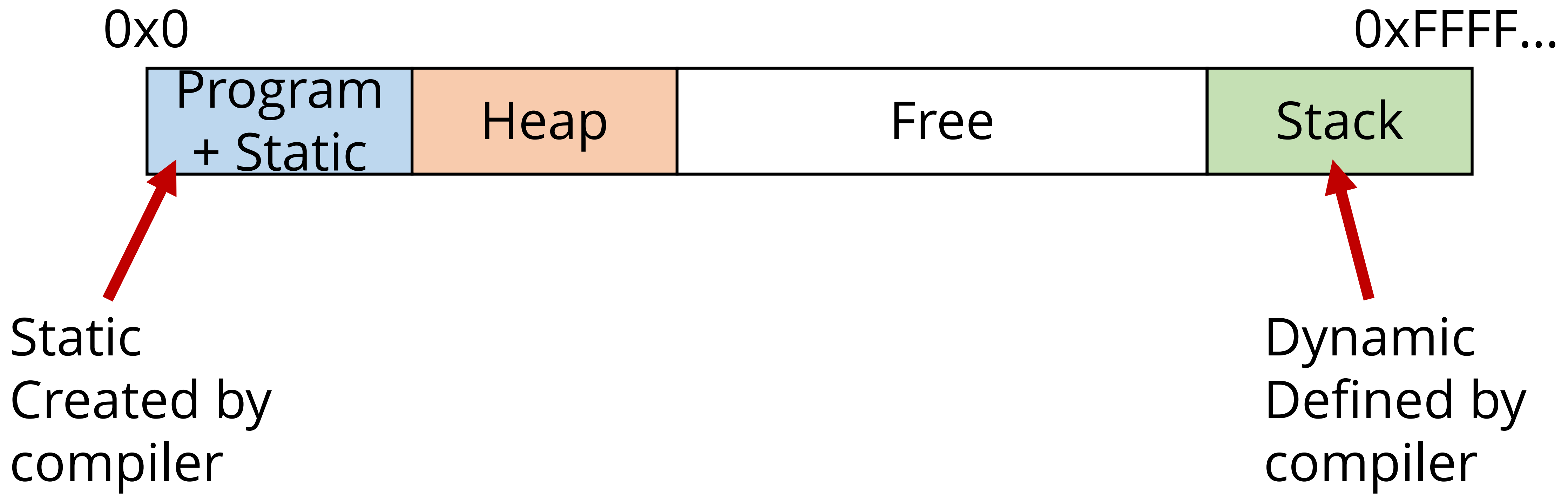


Memory

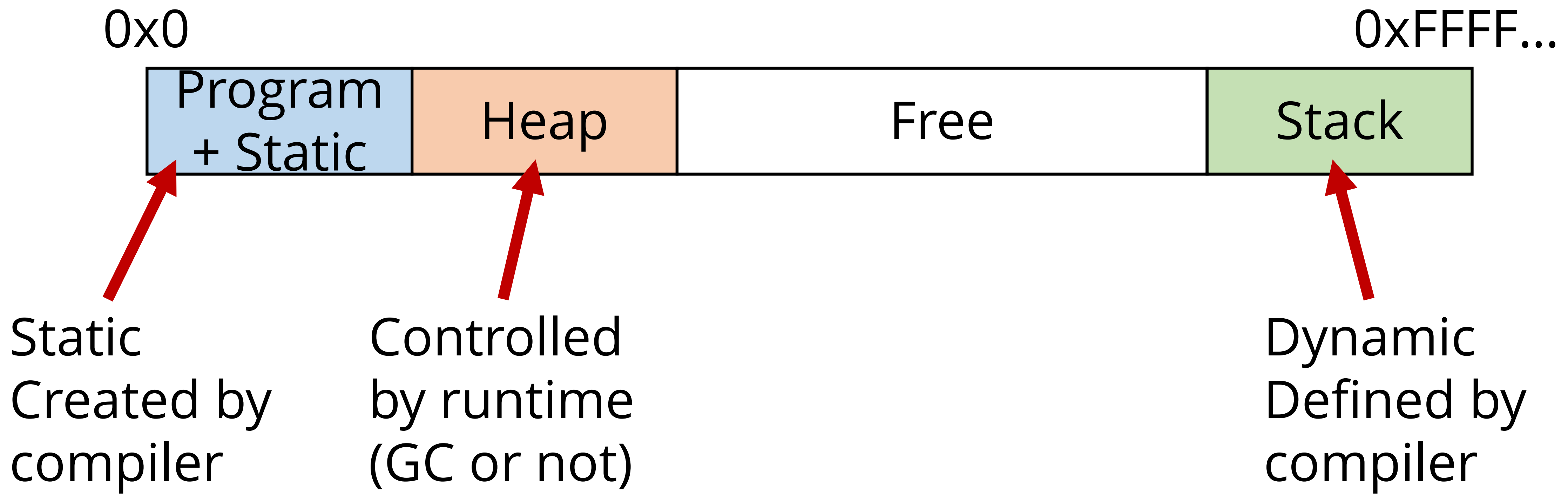


Static
Created by
compiler

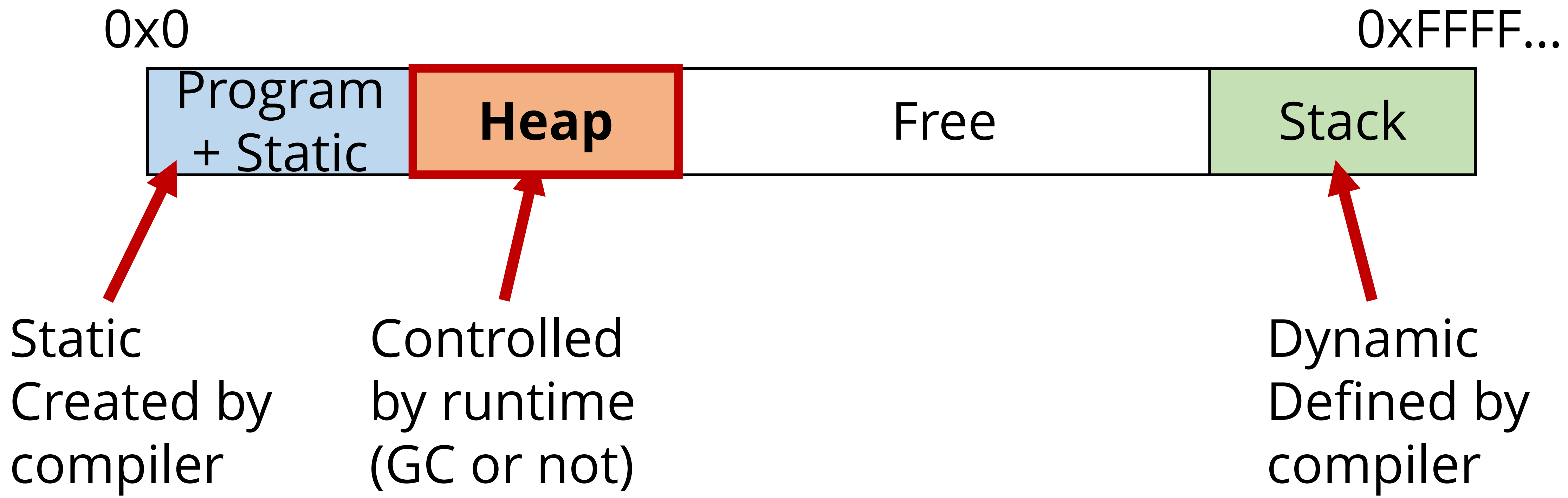
Memory


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

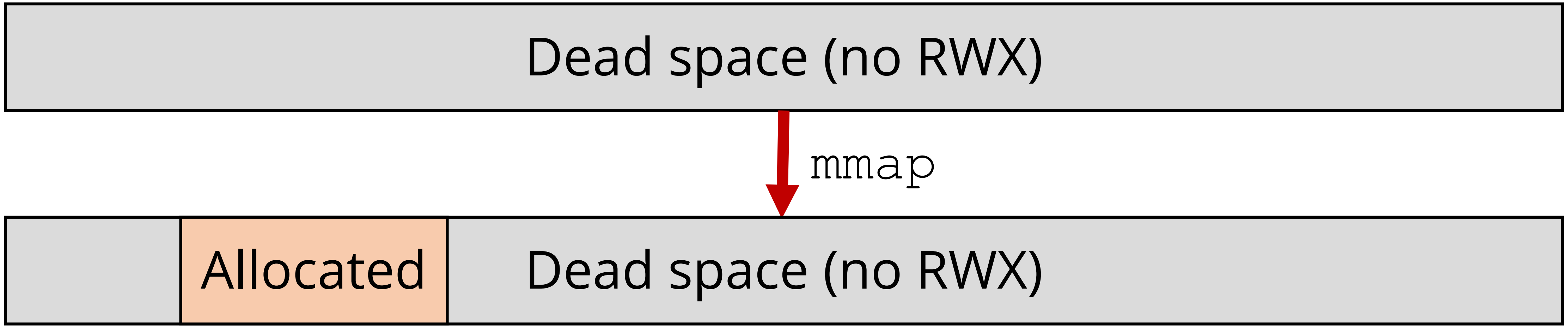
Manual memory

- `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!

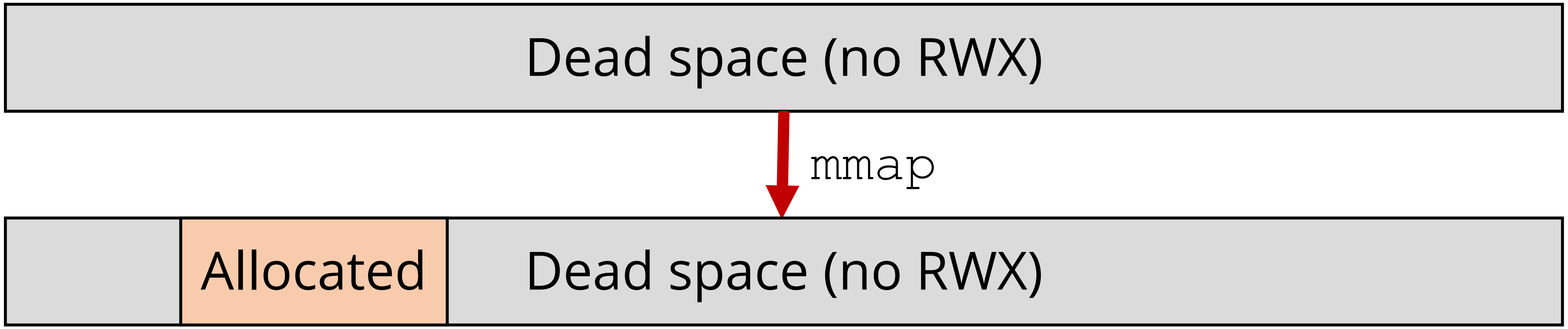
Memory to the manager

Dead space (no RWX)

Memory to the manager

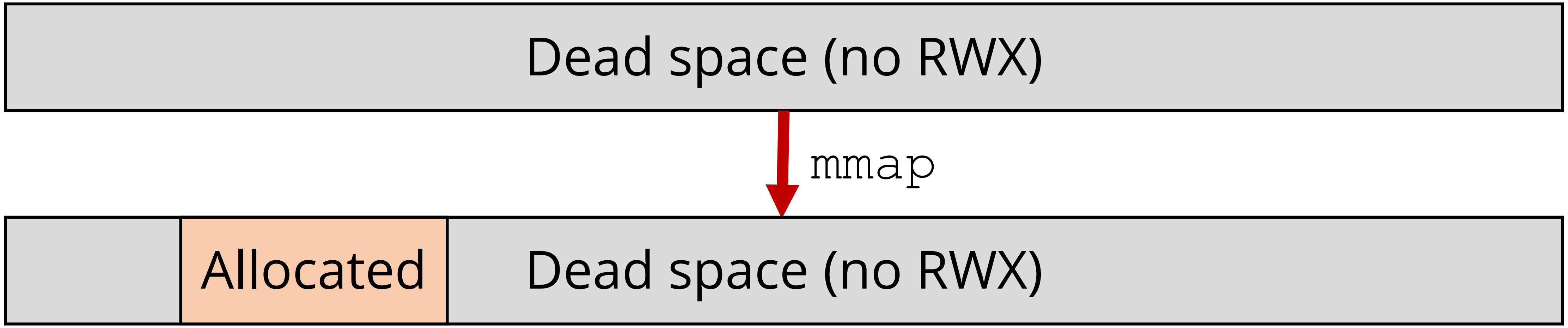


Memory to the manager

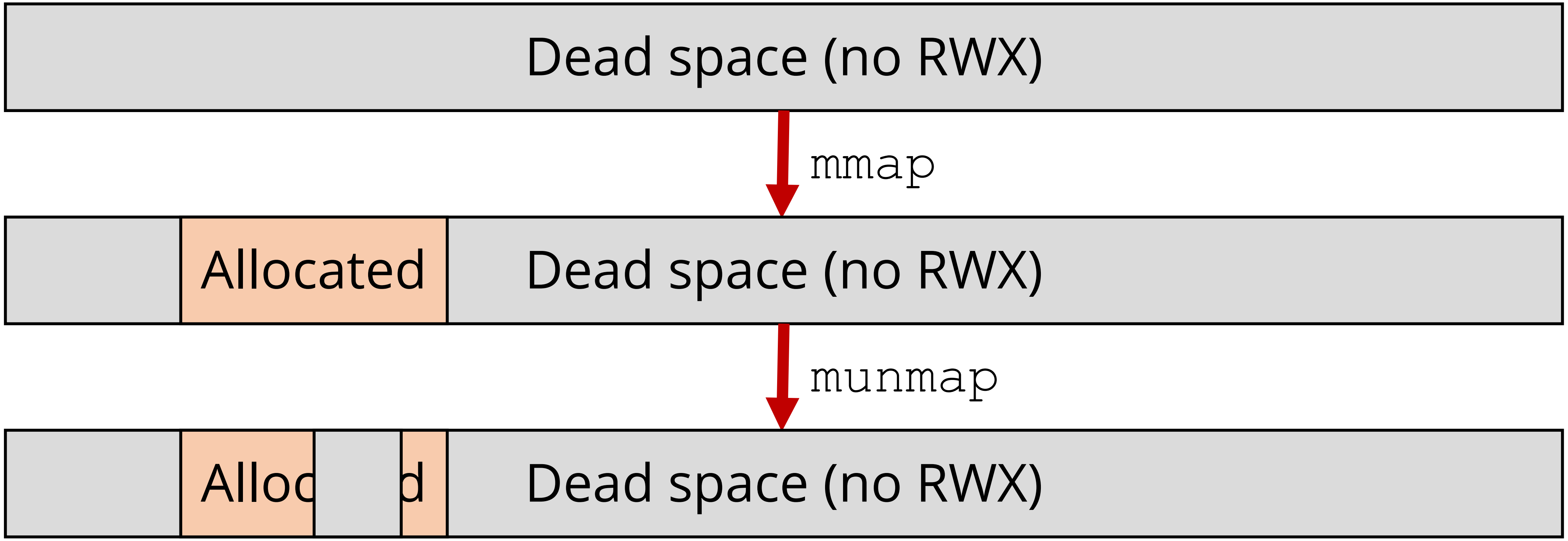


- `mmap` dumbly modifies page table:
  - No memory of its own changes
  - No object illusion

Memory to the manager



Memory to the manager



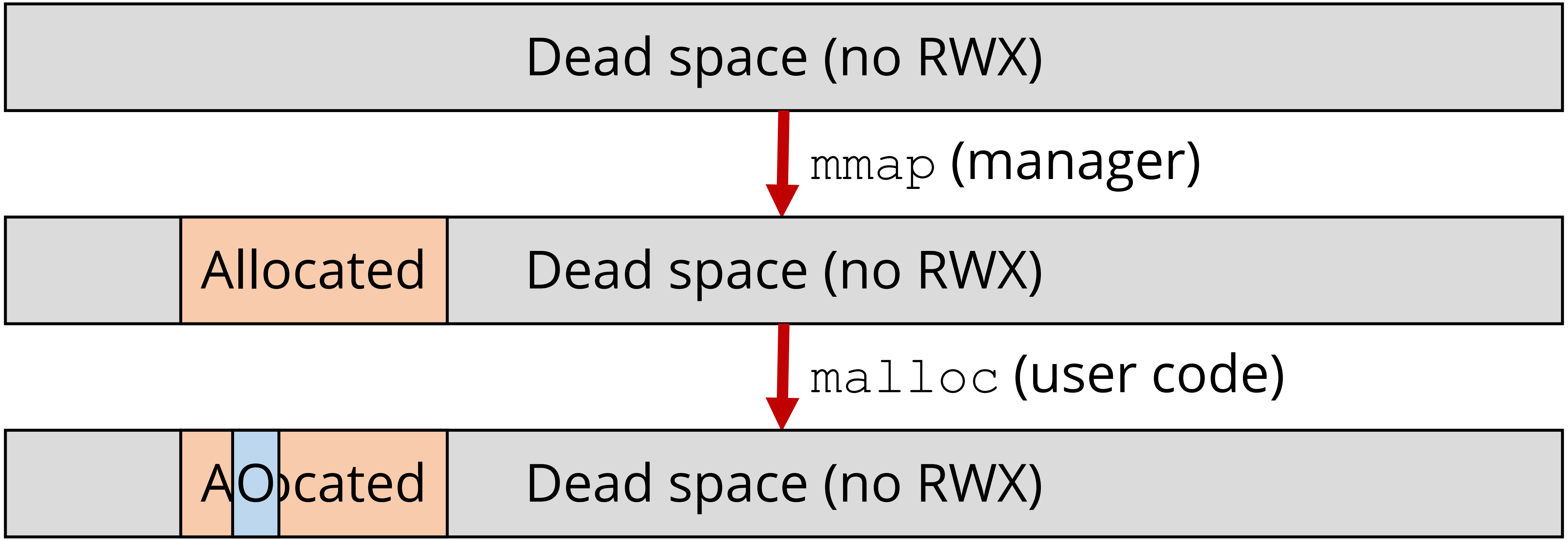
Memory to the manager

- 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

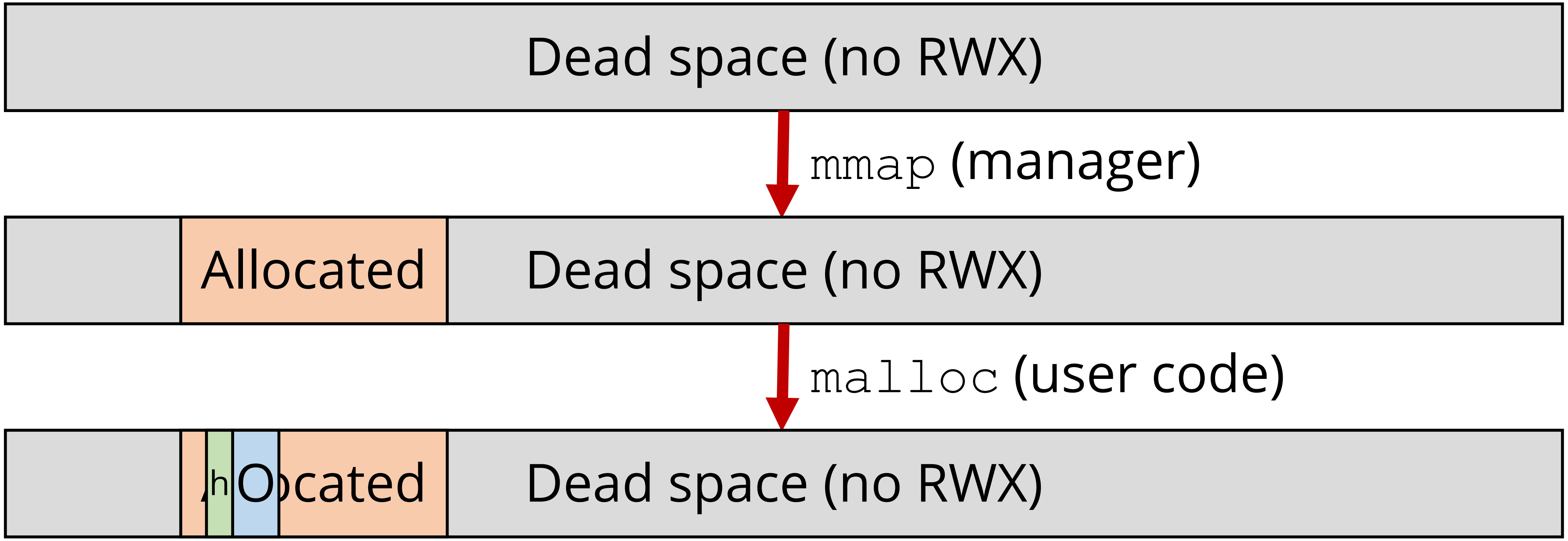
Memory to the manager



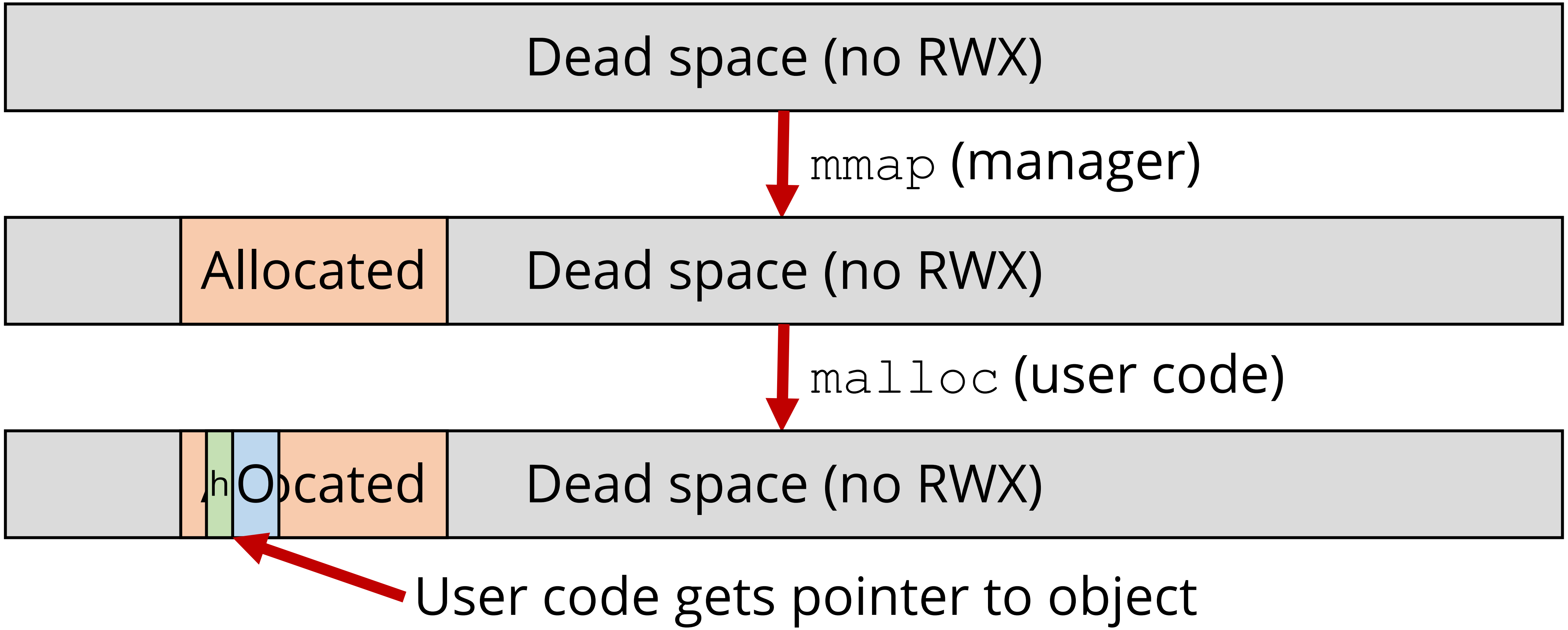
Object illusion

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

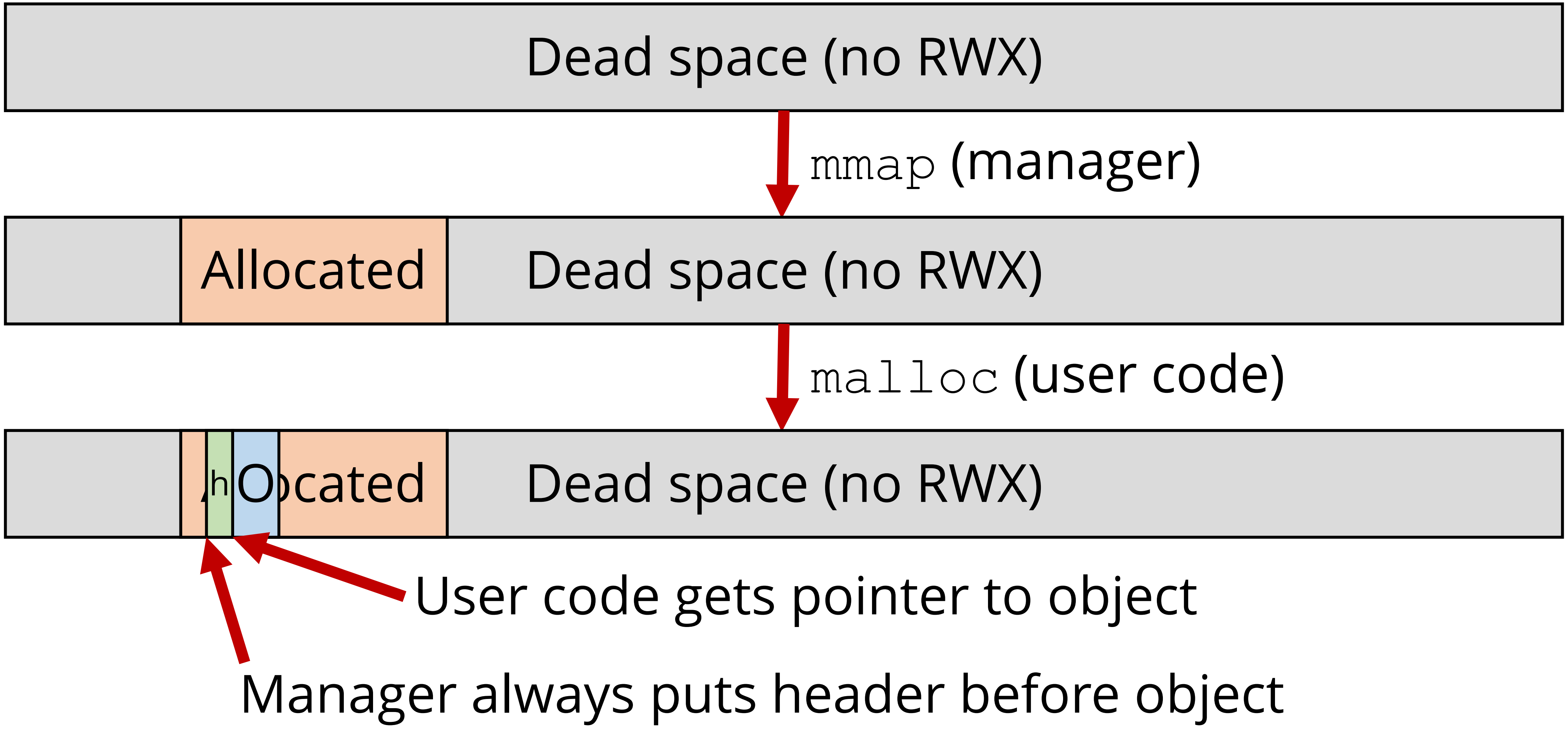
Memory to the manager



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`

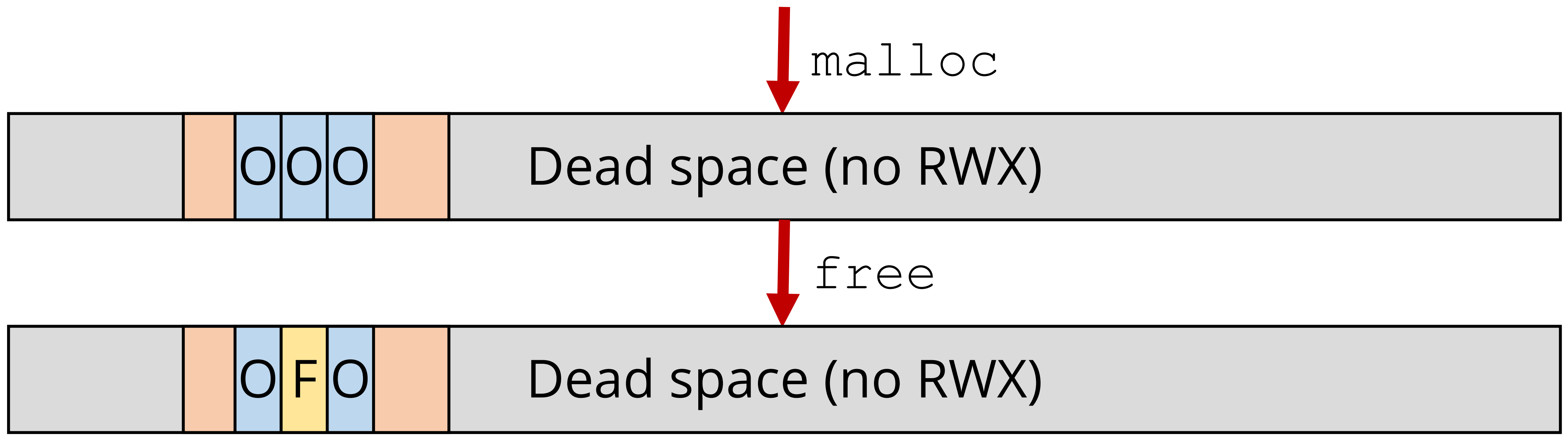
The stupidest manager

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

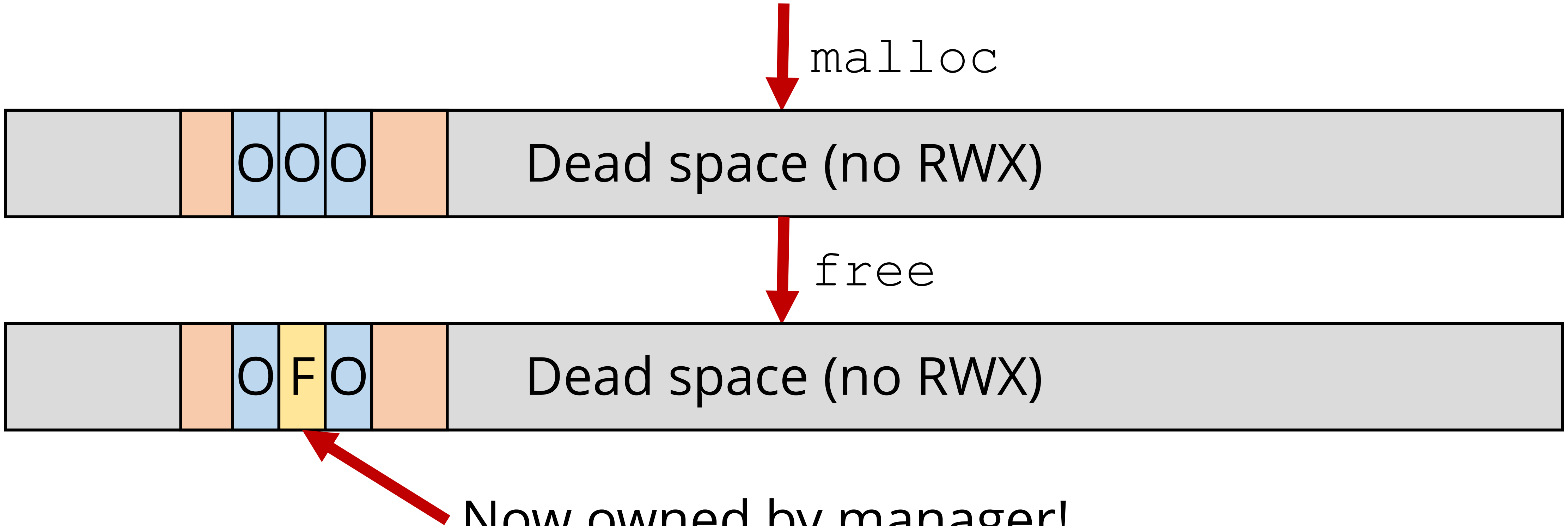
Real management

- 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



Now owned by manager!
Manager must remember all free objects

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!

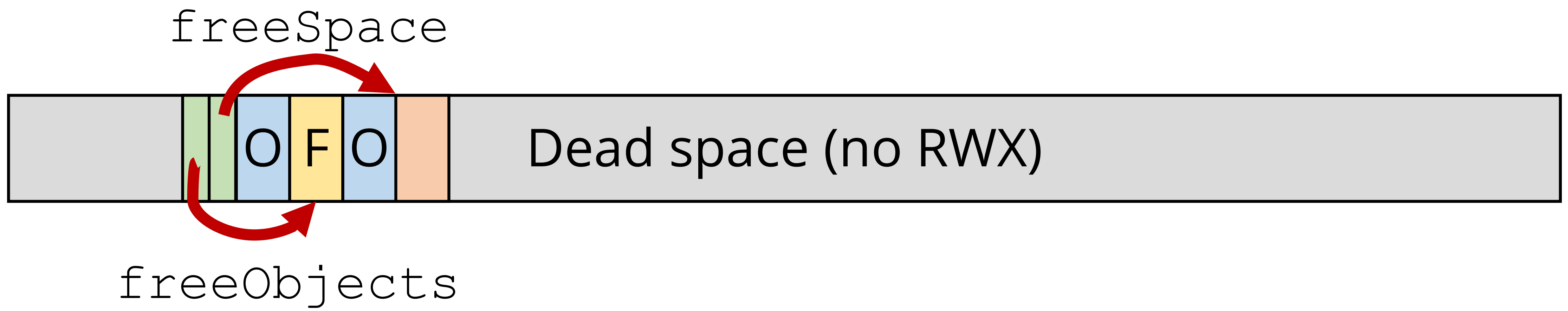
Free objects

```
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

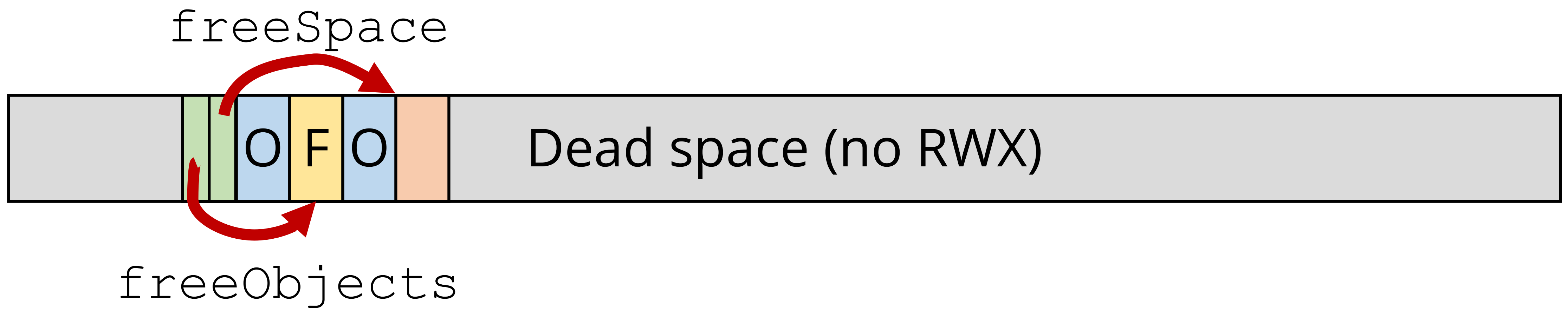


Allocation

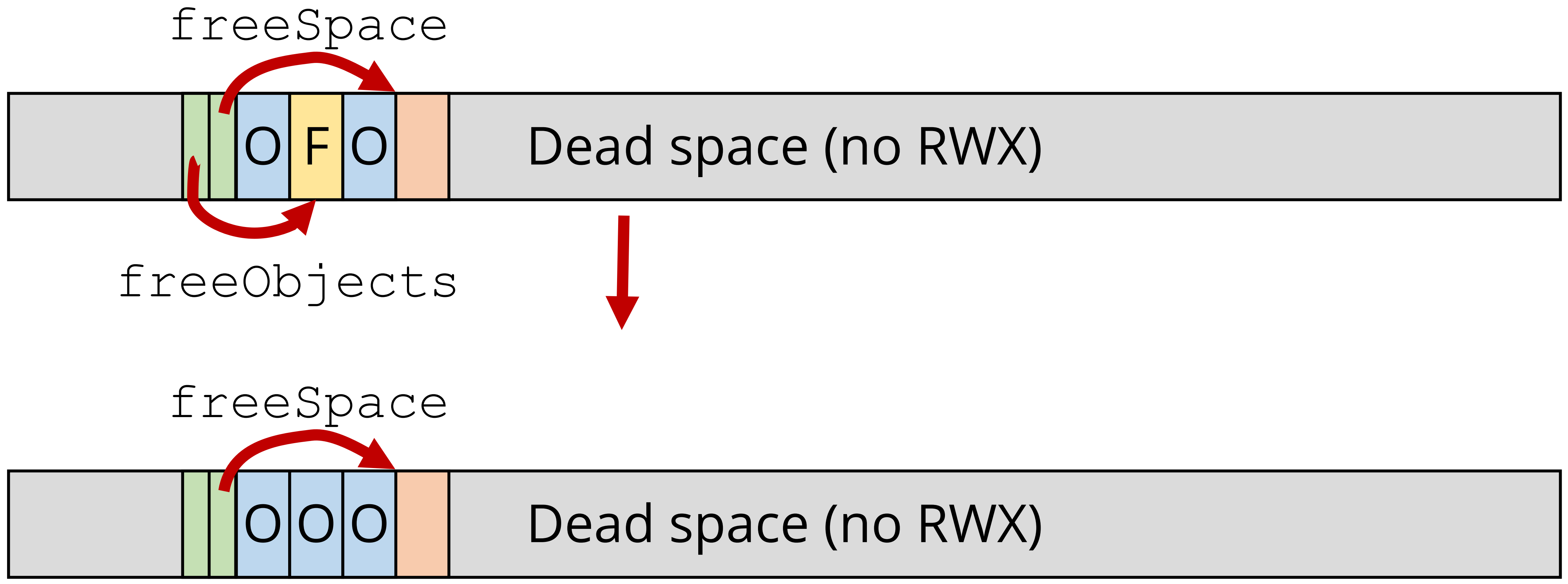
- With free list:
  - First try to find a suitable object[1] on the free list
  - If found, remove from free list and return
  - If not found, allocate new object from free space
  - If no free space, allocate new pool

[1] This process is extremely complicated

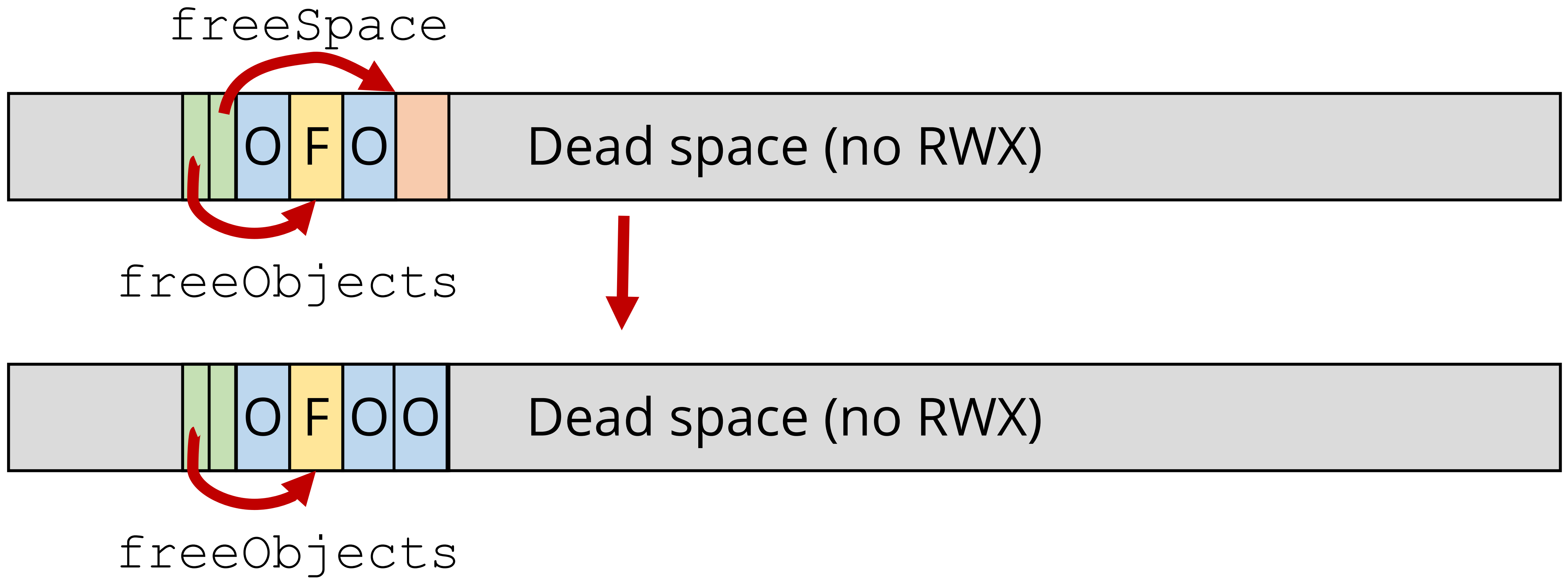
Memory to the manager



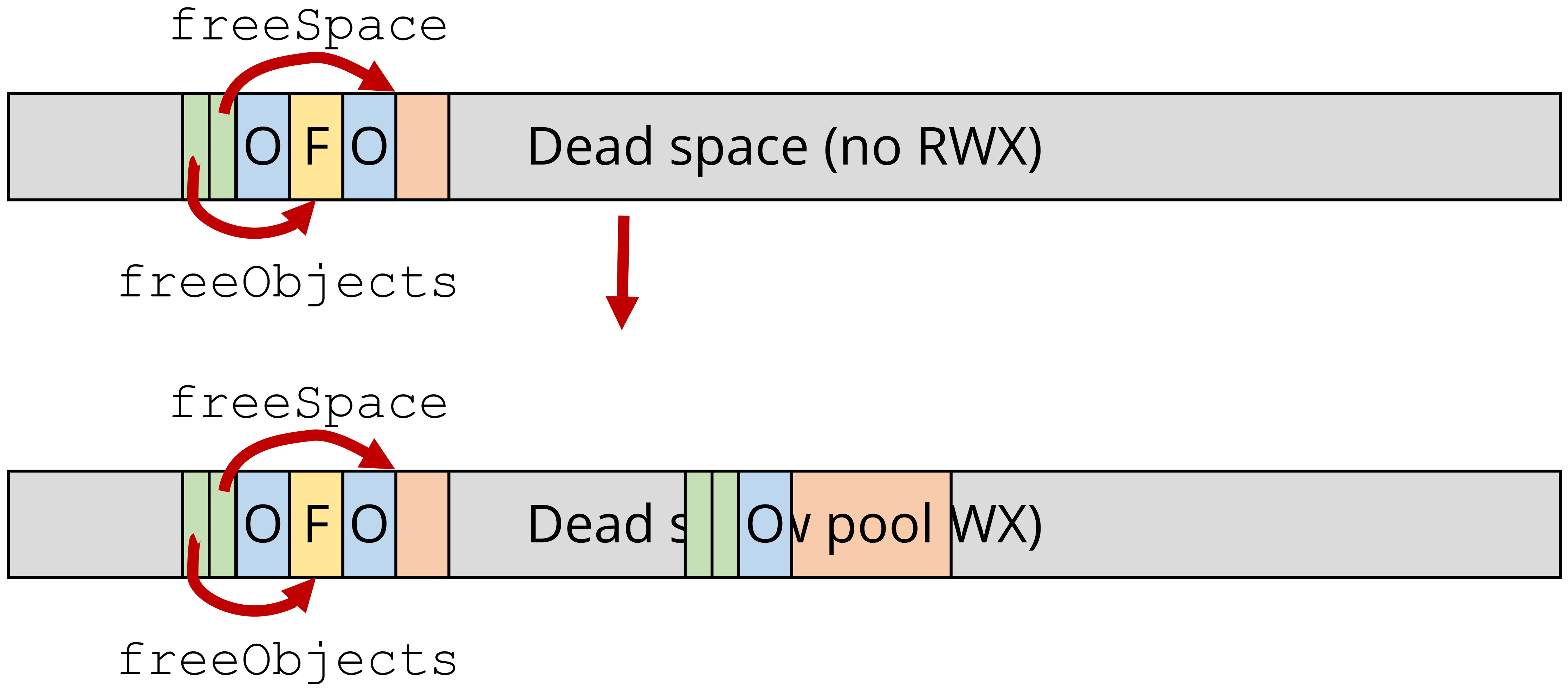
Memory to the manager



Memory to the manager



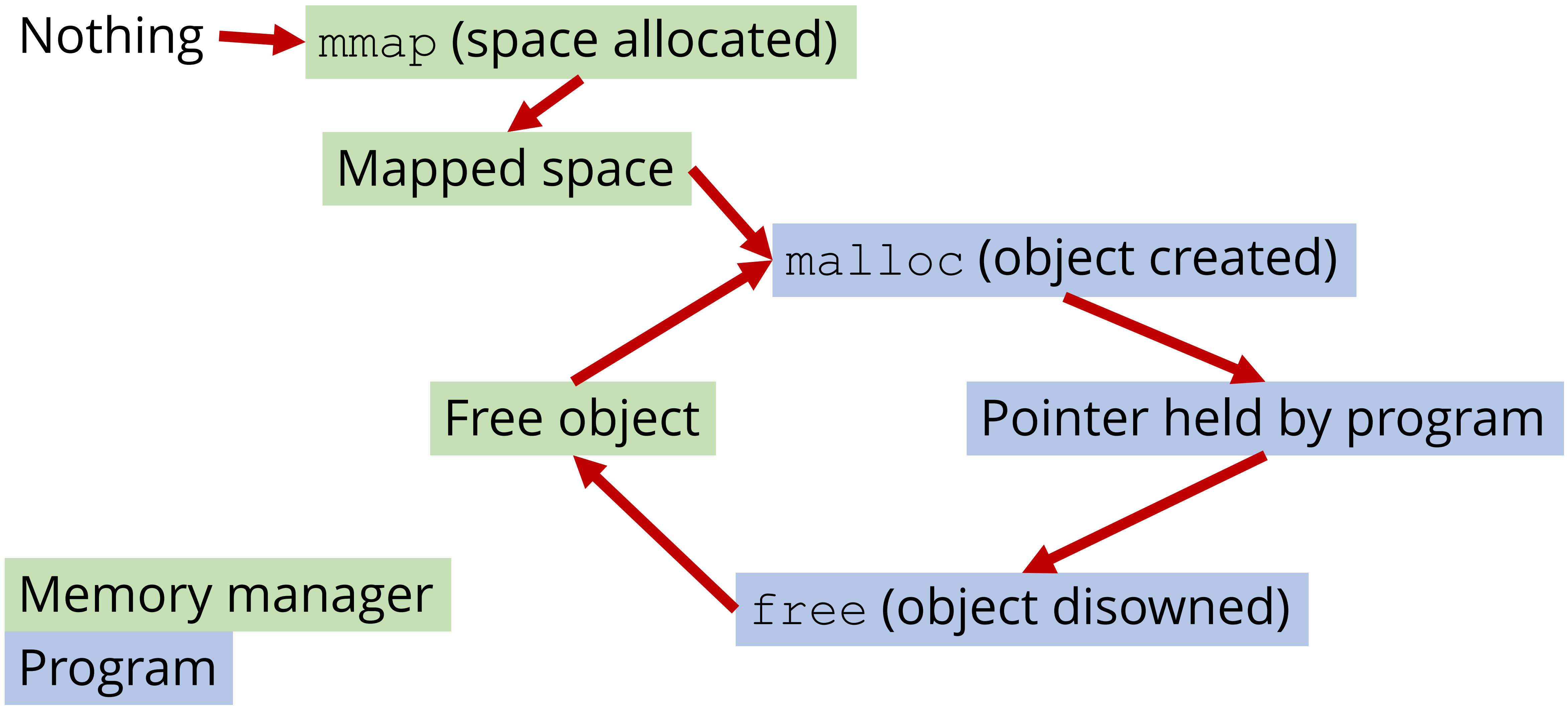
Memory to the manager



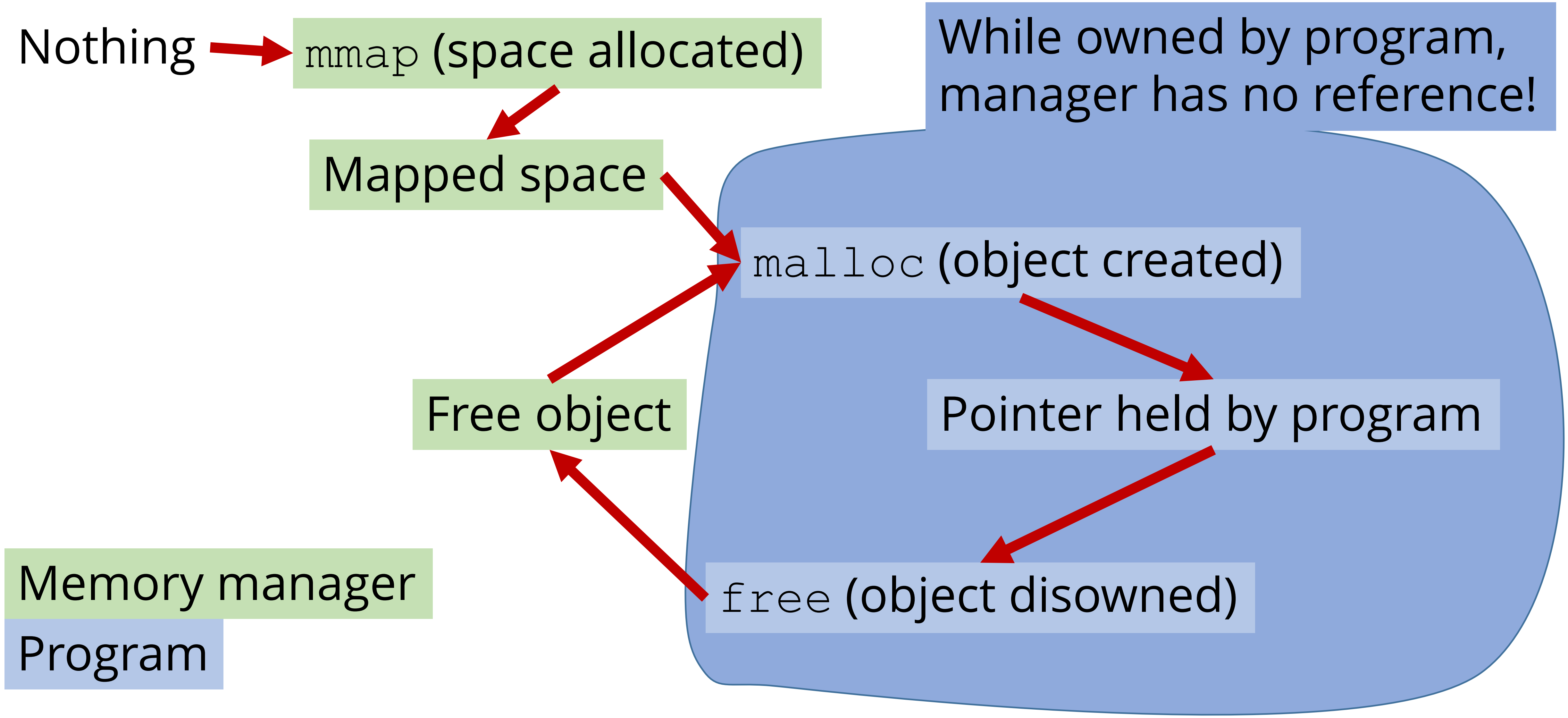
Considerations

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

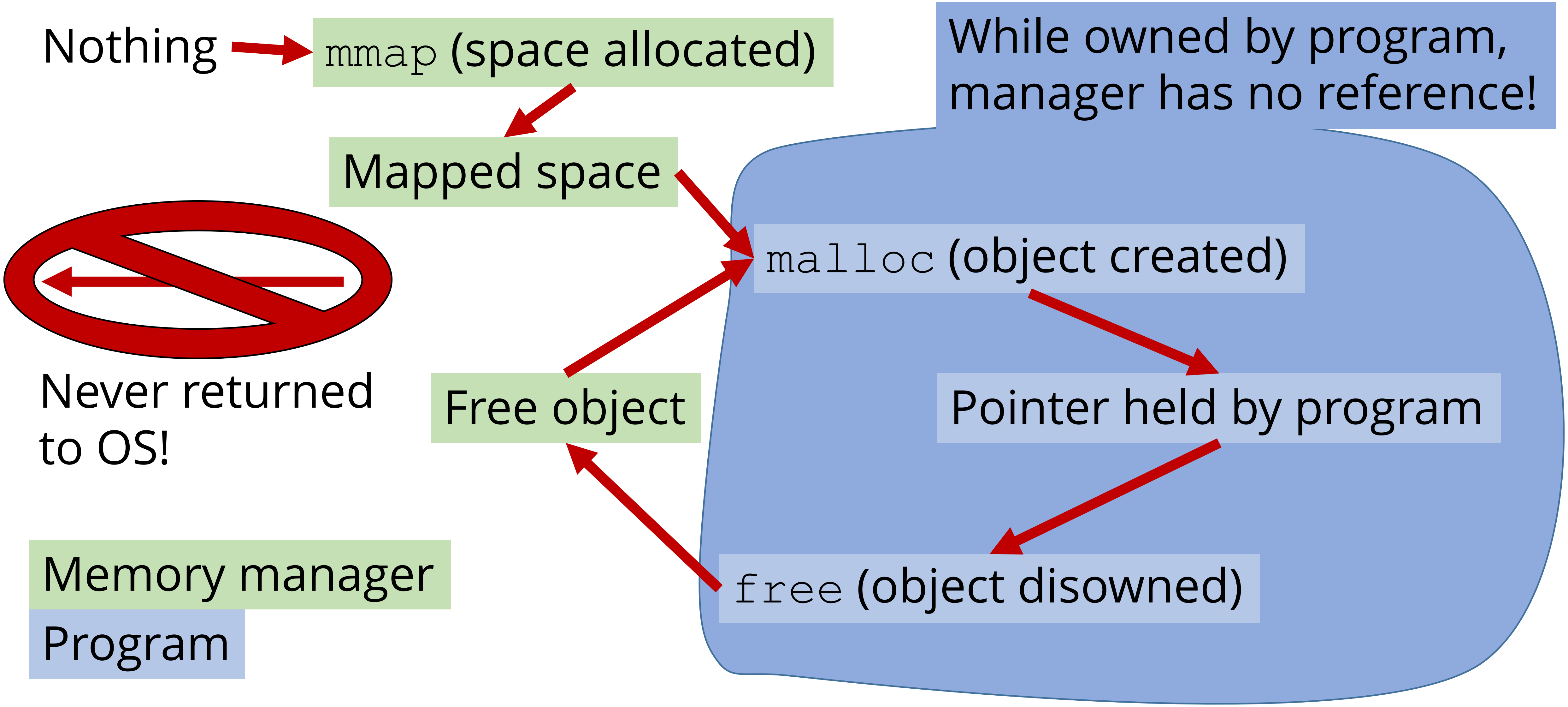
The life of a pointer



The life of a pointer



The life of a pointer



Pools

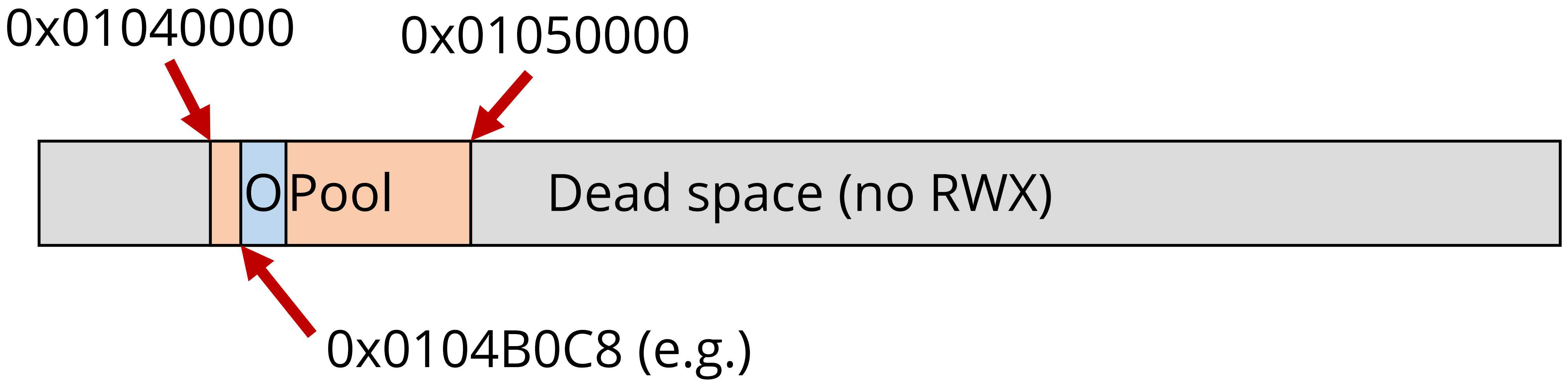
- We may have multiple pools
- Free list per pool or global?
  - If per pool: How to get from `free(o)` to pool?
  - If global: Thread contention ☹

Alignment

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

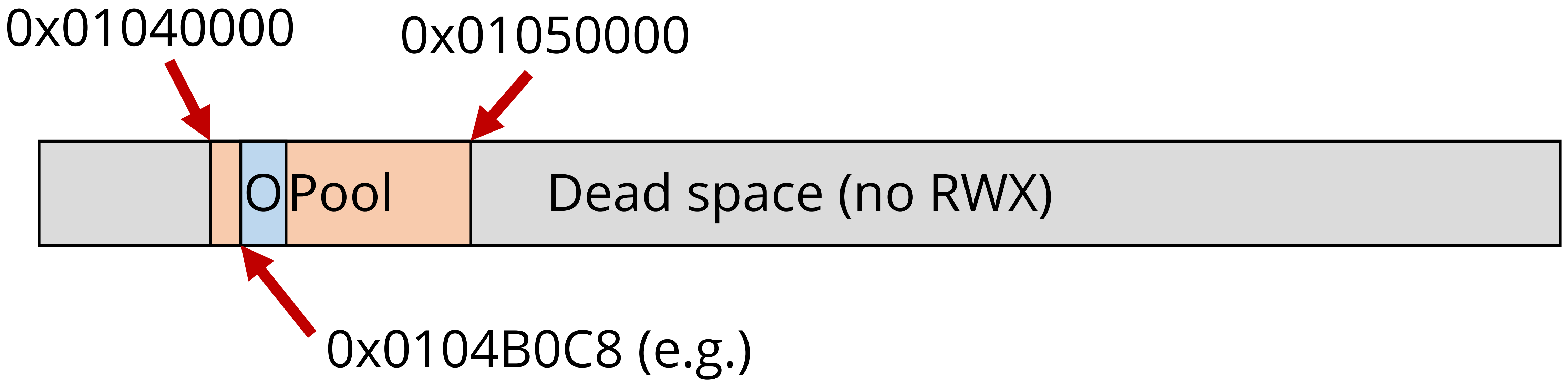
Alignment

Ex: Pools aligned to multiples of 0x00010000:



Alignment

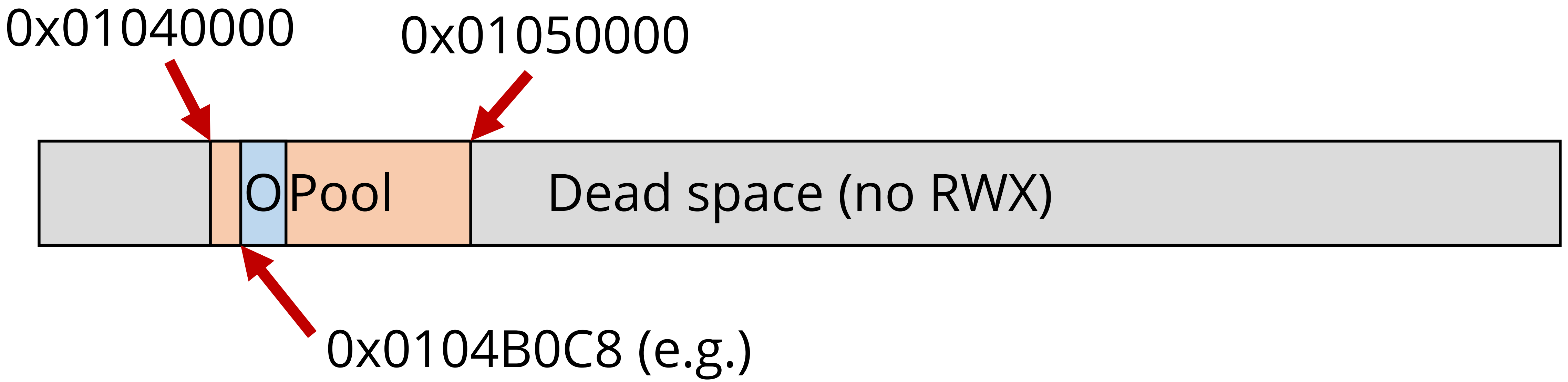
Ex: Pools aligned to multiples of 0x00010000:



“Pool mask”: 0xFFFF0000

Alignment

Ex: Pools aligned to multiples of 0x00010000:

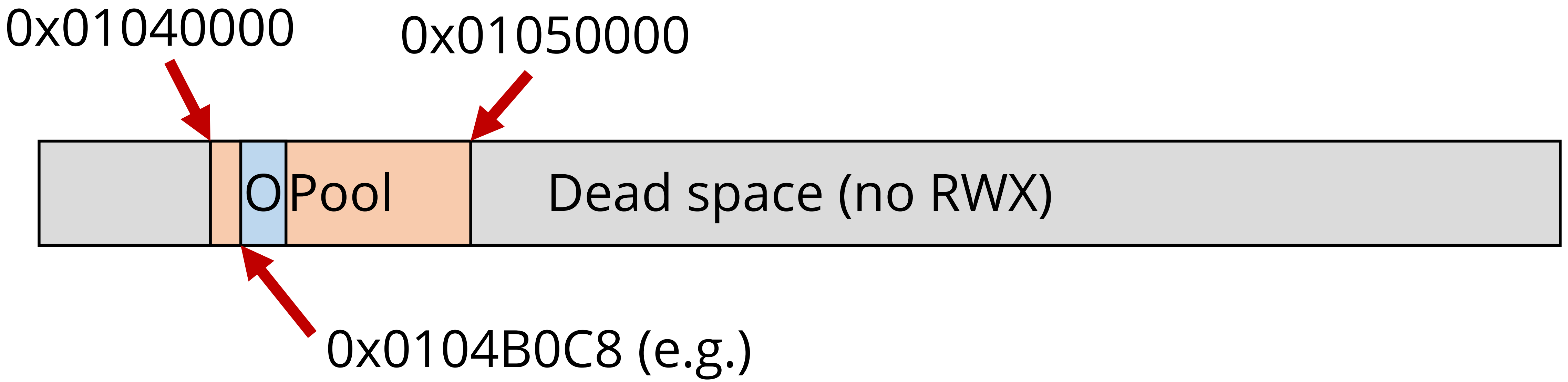


"Pool mask": 0xFFFF0000

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

Alignment

Ex: Pools aligned to multiples of 0x00010000:



“Pool mask”: 0xFFFF0000

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

```c
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;
}
```