

Allocation

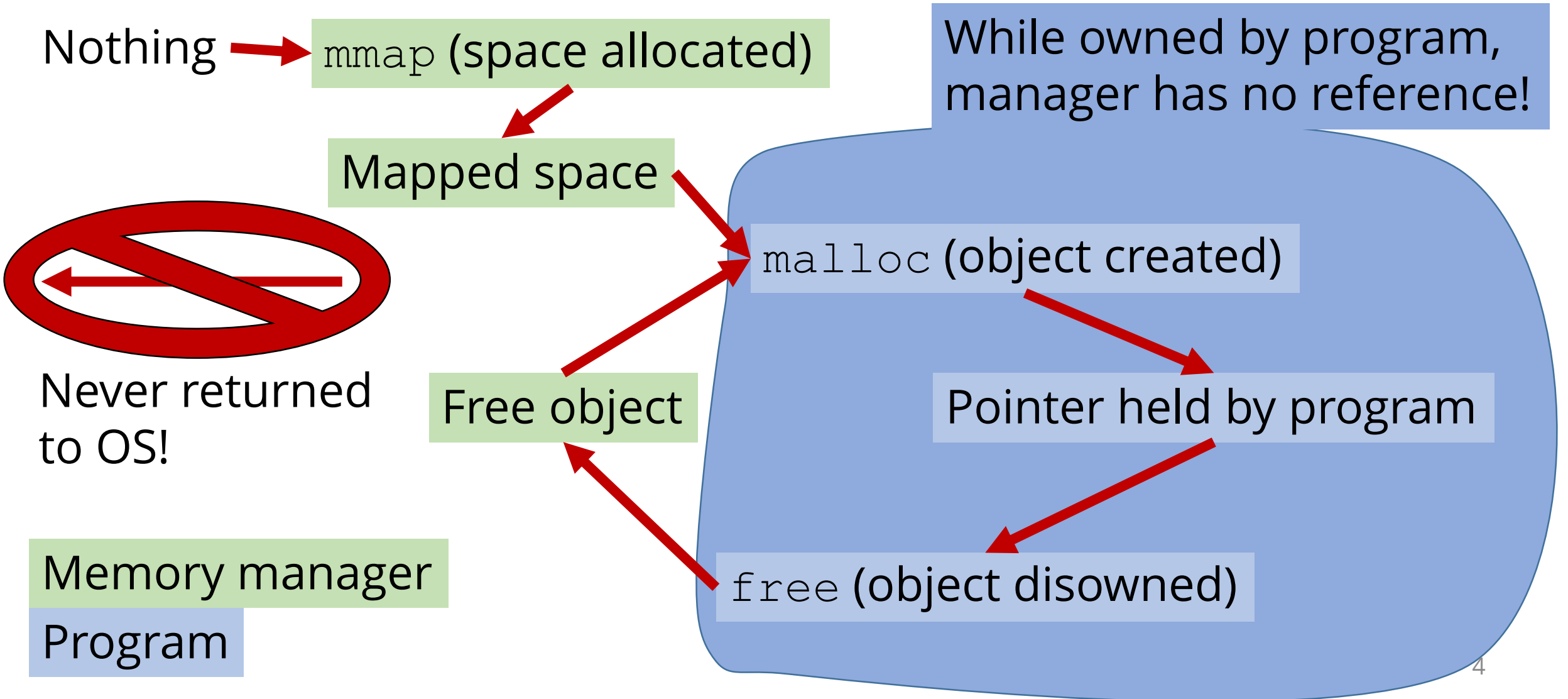
GC basics summary

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

Allocation and revokation

- How you allocate depends on how you free
- Automatic memory management has more options than manual

Review



Manual allocation review

- With free list:
 - First try to find a suitable object 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

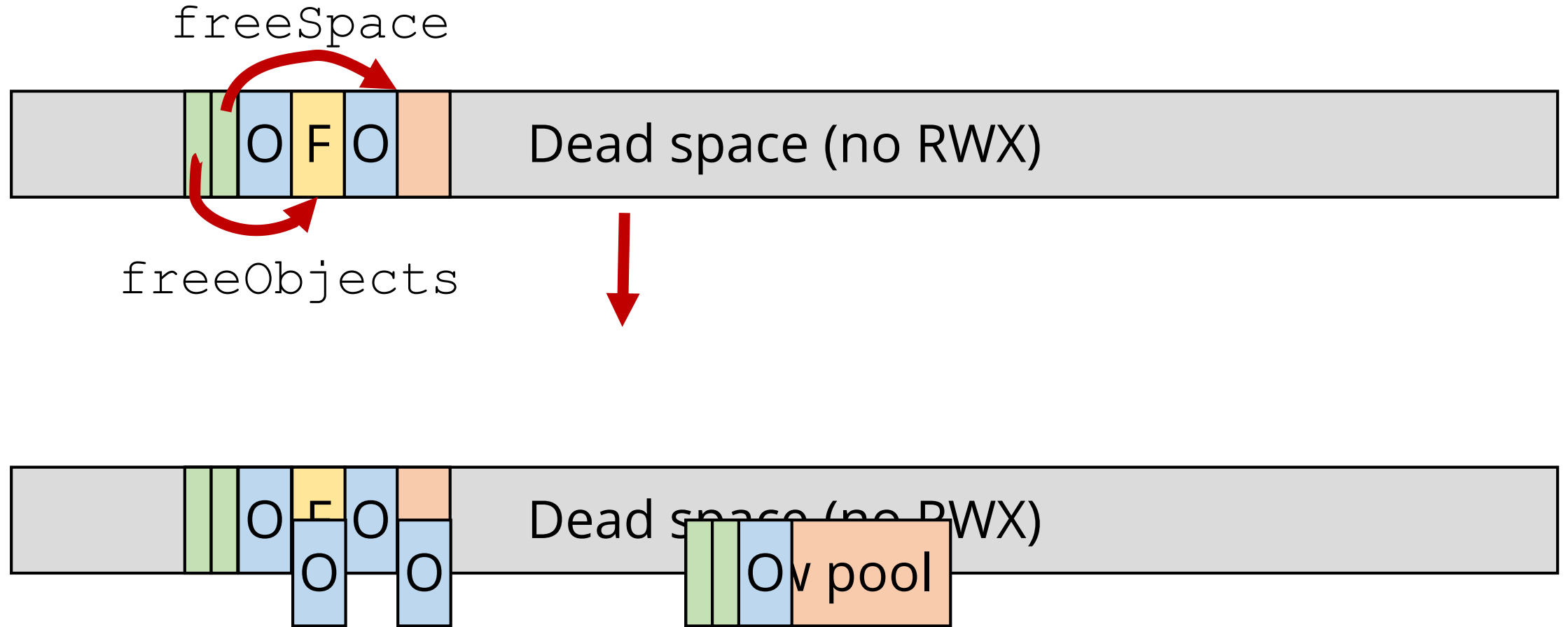
GC and allocation

- Style of GC will affect style of allocation
- Free-list still fundamental

Alignment

- Will need to align allocation to make usable objects
- Most systems require (at least) word alignment
- Round all sizes up to *allocation granule*
- Some objects need larger alignment, but we ignore those in this course

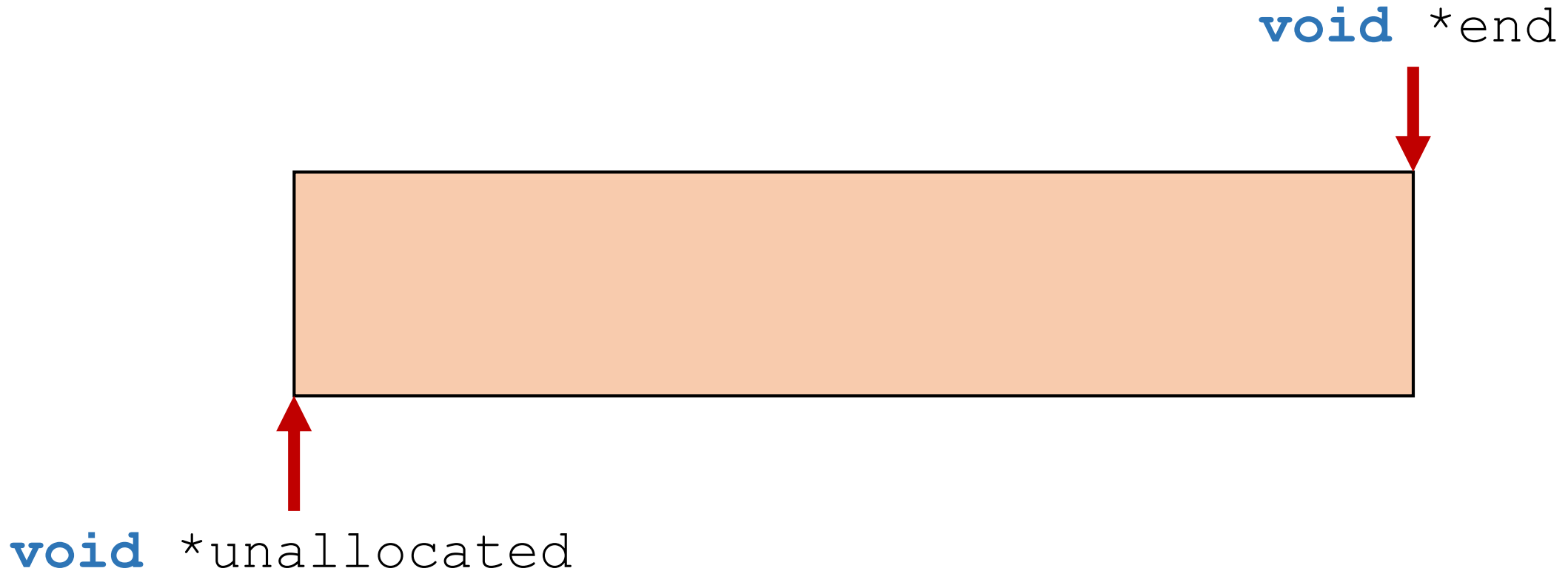
Allocation options



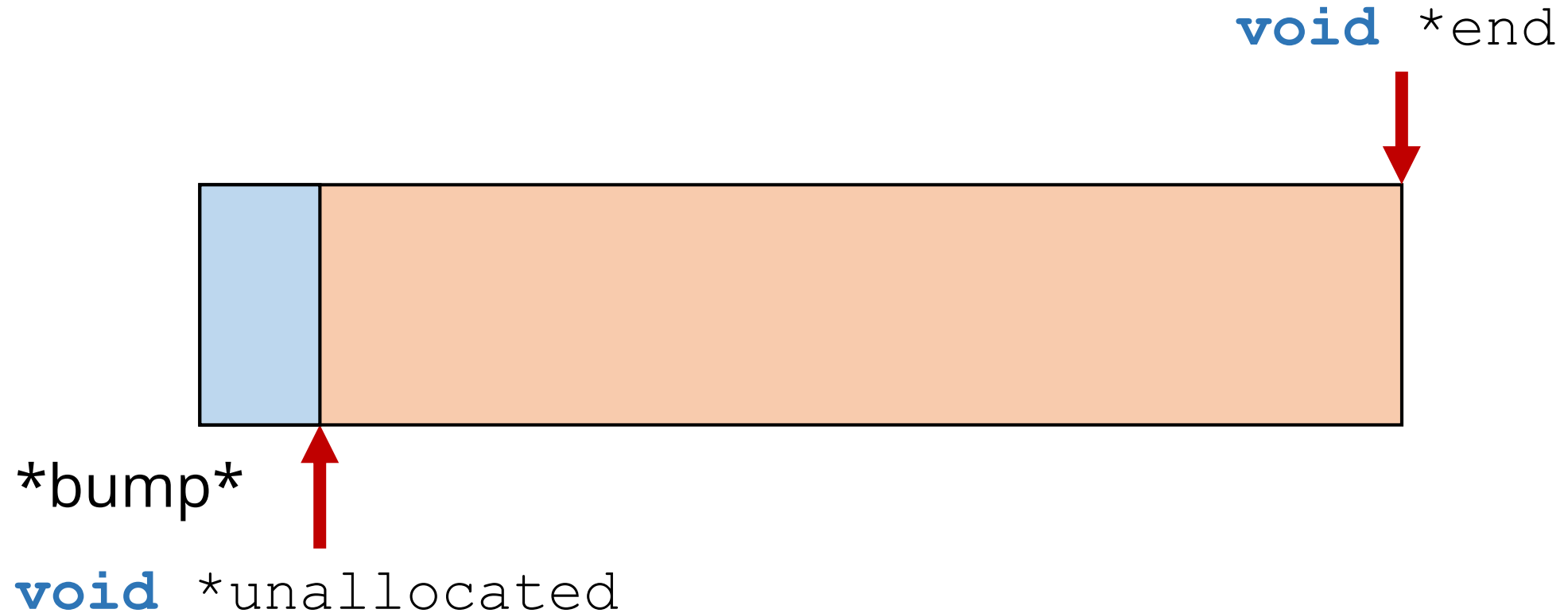
Big free regions

- Obvious option: Allocate from beginning towards end
- “Bump-pointer allocation” or “Sequential allocation”

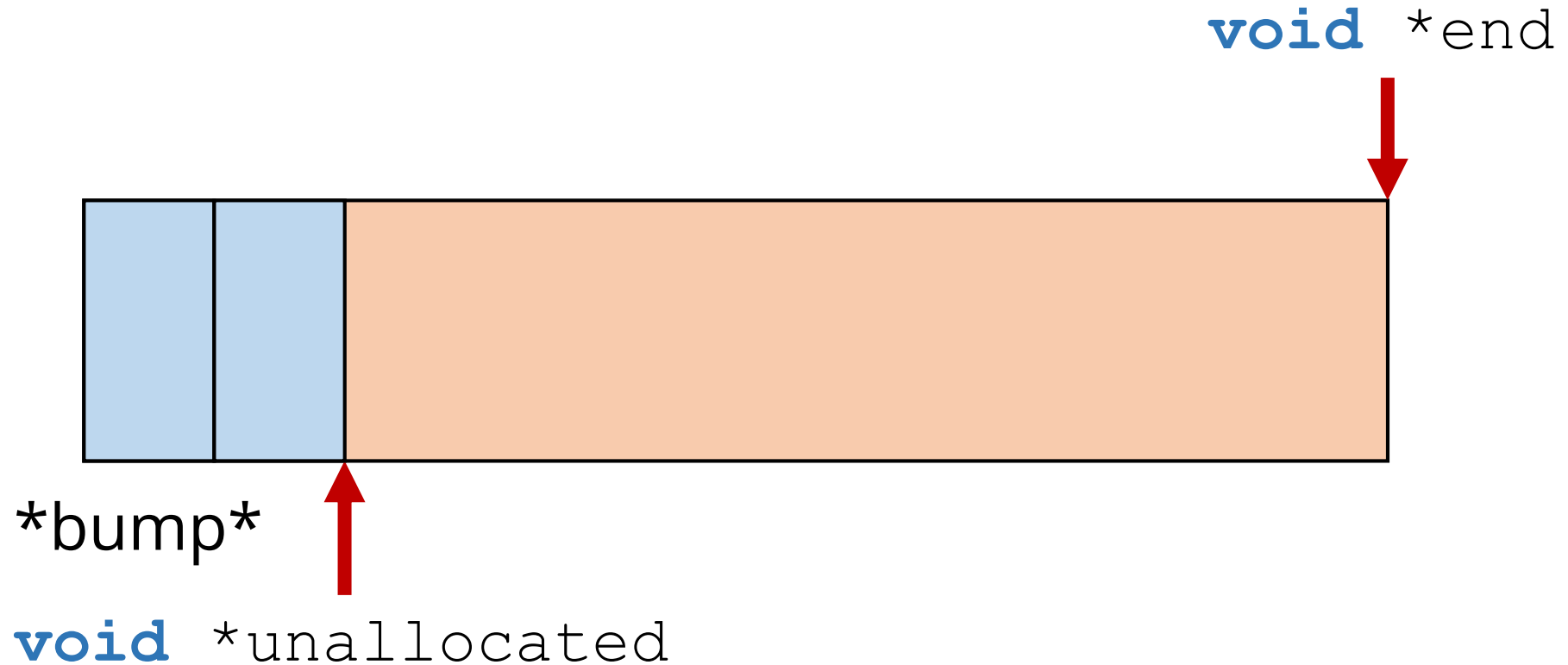
Bump-pointer allocation



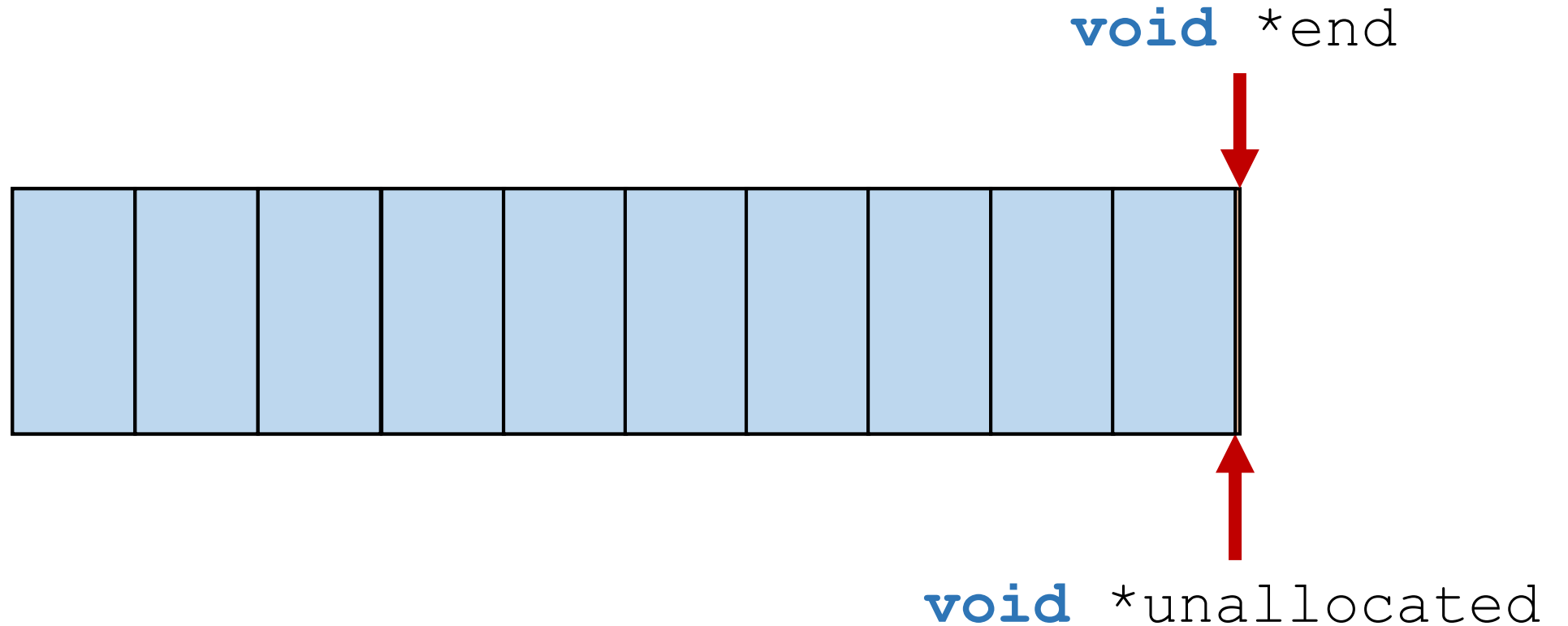
Bump-pointer allocation



Bump-pointer allocation



Bump-pointer allocation

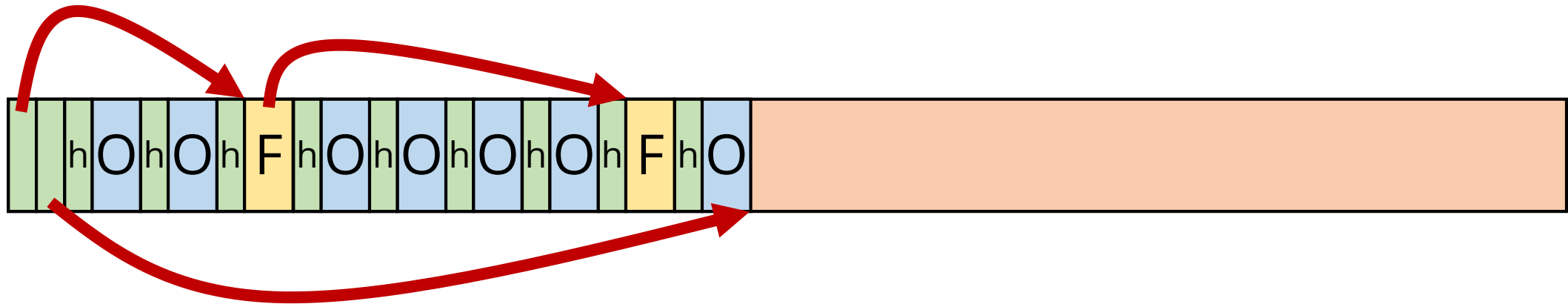


Why bump?

- Easy and fast
- No wasted space during allocation
- Only requires two pointers¹ per pool
- But: Only makes sense for big free regions

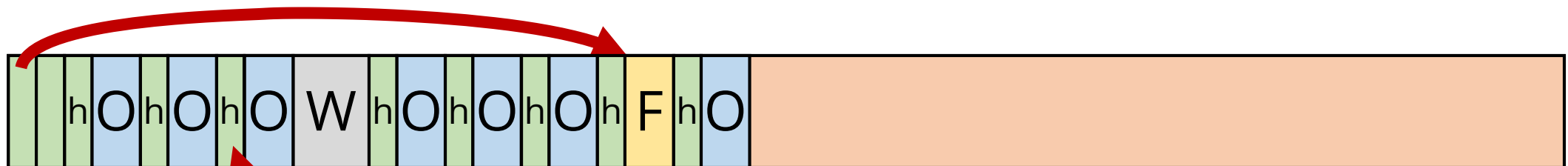
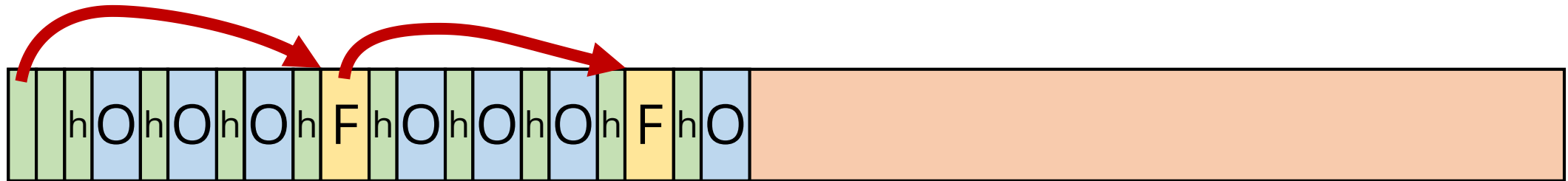
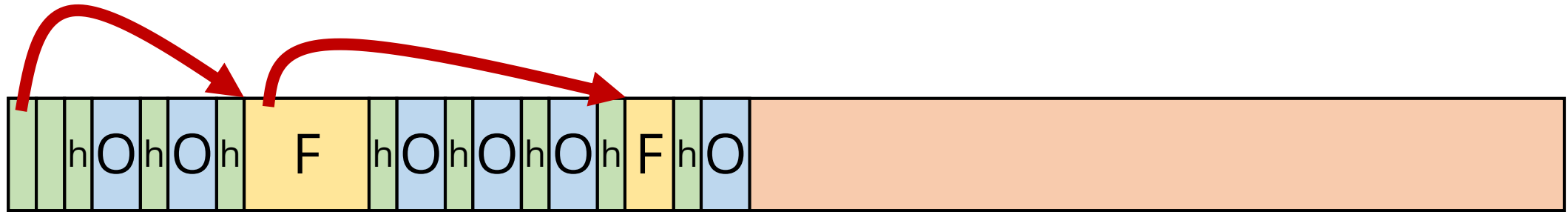
¹ The end pointer may be implicit

Free-list allocation



- Free objects kept on list
- Allocate by taking an object from the free list
- Object must be at least large enough

Splitting vs overallocation



Header must specify sizeof(O+W)

Fragmentation

- Unused space between used objects
- Cannot allocate objects larger than largest fragment!
- With unmoving objects, can only avoid fragmentation, not prevent

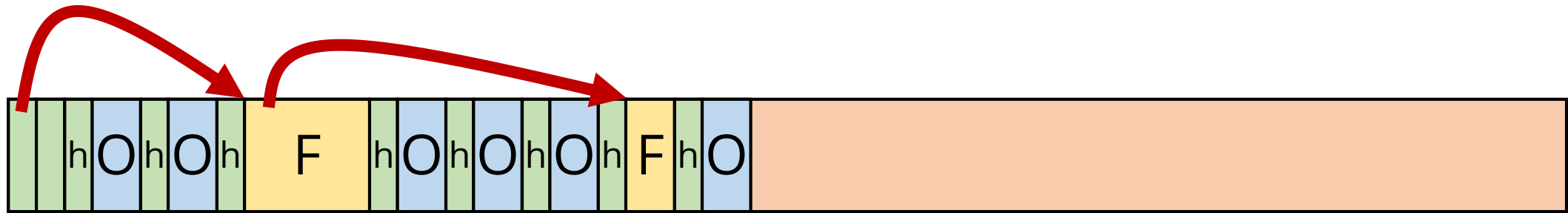
External vs internal fragments

- External fragmentation:
 - Unused space *between* objects (freeing and splitting)
- Internal fragmentation:
 - Unused space *within* objects (overallocation or padding/alignment issues)

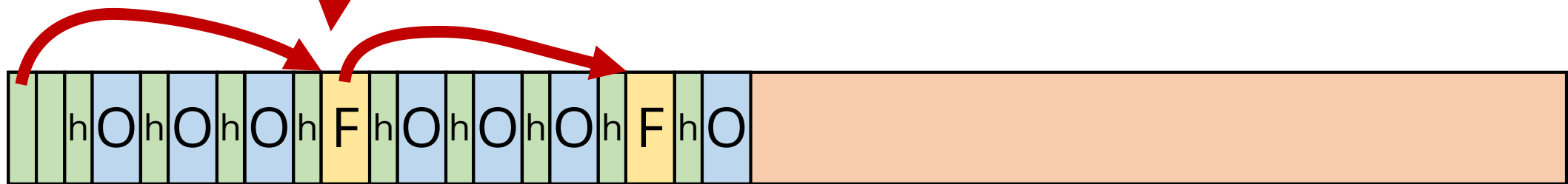
Choose wisely

- Free object of wrong size → splitting
- Alternatively, search through list for object of right size
- But searching through list takes time

First-fit



Always choose first, split if necessary



Aside: List order

- In manual memory management, list could be in any order
- In GC, list is created by sweep phase
- Typically ordered low memory to high memory

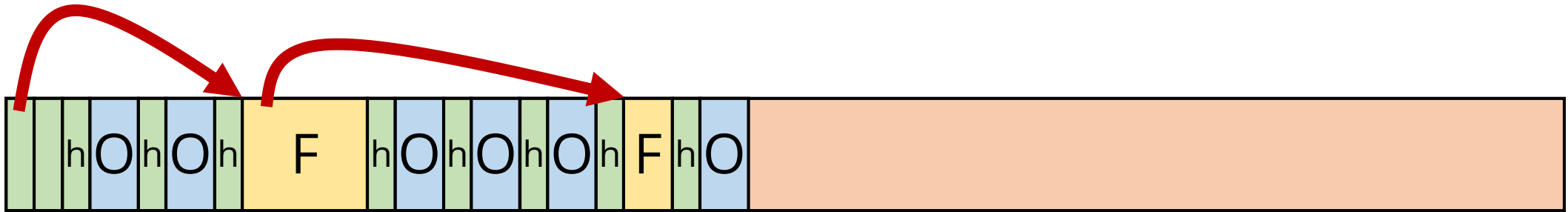
First-fit

- Fast to find objects (“expected” $O(1)$)
- Unusable fragments cluster at beginning
- Maximize memory reuse (cache!)

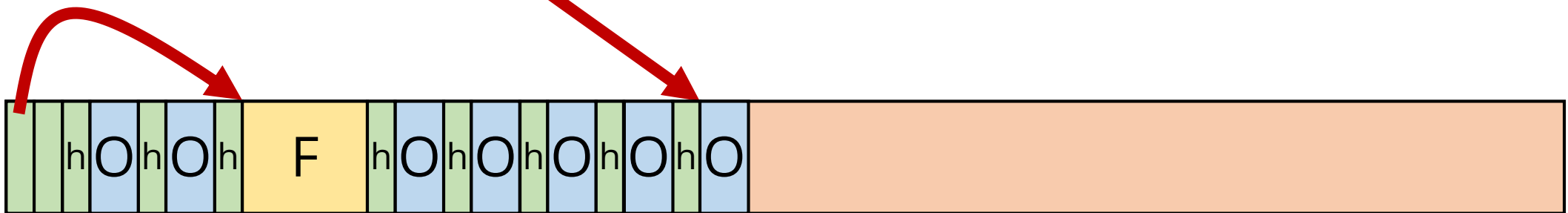
Next-fit

- Use a circular linked list
- Less skipping over fragments
- Less reuse

Best-fit



Always choose smallest \geq needed size



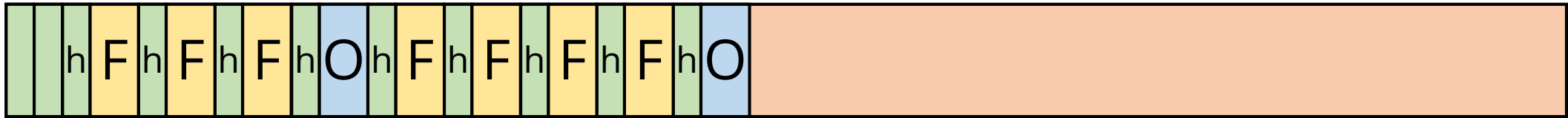
Best-fit

- Intuitive “best” fit
- Usual loads have perfect reuse
- On some loads, *worst* case: Unusable tiny fragments!
- Fragments don't cluster

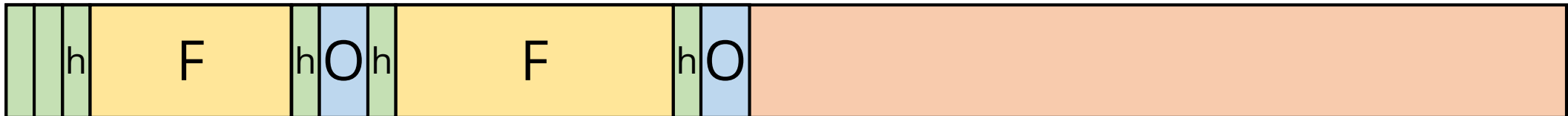
Which-fit?

- No fitting algorithm is always best
- With many similar-size objects, all the same!
- Cache locality is important

Coalescence



In free-list, all separate objects
In reality, two big blocks of space



To coalesce, must have sorted free-list
GC can do that!... but doesn't necessarily

Coalescence

- GC may provide unsorted free list
- To coalesce, must sort list
- When to sort/coalesce?
 - When fitting algo sucks too much, or
 - during GC

Breather

- Bump-pointer great when possible
- Free-list: First-, next-, best-fit
- Best not always the best
- Coalesce to avoid waste

Back to free-lists

- All this because lists suck!
- So, make a sorted tree of free objects
- Consequence: Free object definition changes

Free objects

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

```
struct FreeObject {  
    struct FreeObject *left;  
    struct FreeObject *right;  
};
```

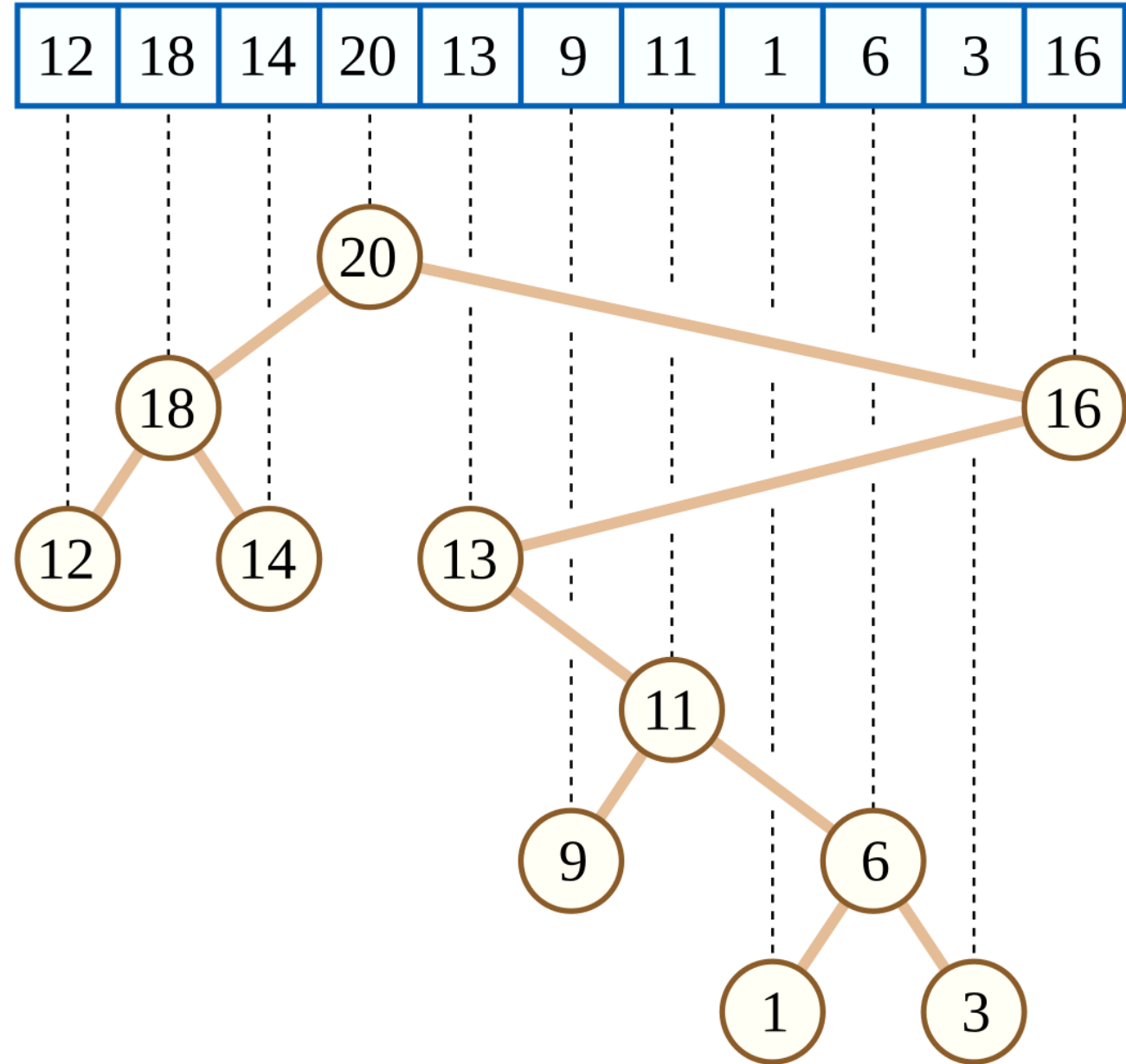
```
struct Pool {  
    struct FreeObject *freeObjectsRoot;  
    void *freeSpace;  
};
```

How to sort

- Sort tree depending on fit algo:
 - First-fit/next-fit: Sorted by address
 - Best-fit: Sorted by size
- For best-fit, simple sorted tree OK
- For first-fit/next-fit, Cartesian tree

Cartesian tree

- Tree sorted by address
- Max-heap property on size
- Walk by address fast, walk by size fast enough



Segregated-fits

- We've assumed one free-list
- Why not multiple free-lists?
- Create size categories with free-lists per size

Segregated-fits



- Static (usually) number of size categories
- Last category is “big”, for bigger objects
- *Always* round up allocation to nearest size category!
- Creates internal fragmentation

Splitting in segregated-fits

- Allocation from “big” is usual
- Splitting necessary to move objects to lower category
- Must split into category-sized objects
- If such a split is impossible, must overallocate
- More internal fragmentation!

Worth it?

- Segregated-fit is much more complicated
- Creates internal fragmentation
- (Nearly) Eliminates external fragmentation
- Faster allocation

Segregated blocks

- Can get small objects by splitting big ones...
- Or, allocate objects of different sizes in different pools
- Eliminates external fragmentation
- Can remove size from object header

Bitmapped-fits

- Free objects must be coalesced
- Instead, put freeness info aside, at beginning of pool
- Need one bit per granule of pool
- Bit 0 = free, 1 = used

Bitmapped-fits

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

```
struct Pool {  
    unsigned char freeMap  
        [POOL_SIZE/ALLOCATION_GRANULE/8];  
    void *freeSpace;  
};
```


Bitmapped-fits

- Some wasted space in pool
- Coalescence is automatic
- No explicit list/tree of free objects

Locality

- Locality matters!
- With no caches, best-fit always wins
- With cache, first-fit often wins
- Objects allocated at same time often used at same time

Overhead

- Faster to fill all empty space before using free-list
- But, wastes memory, poor locality
- “Wilderness preservation”: Don’t tread on anything you don’t need to!

Concurrency

- Synchronized allocation is death
- Thread-local pools/free-lists
- Only synchronize when acquiring new space
- (Another reason to allocate from free-list first)

Summary

- Bump-pointer great when possible
- Free-list options:
 - First-fit, next-fit, best-fit (Tree, not list)
 - Segregated-fit
 - Bitmapped-fit
- Locality and concurrency → Better to use free-lists than free space