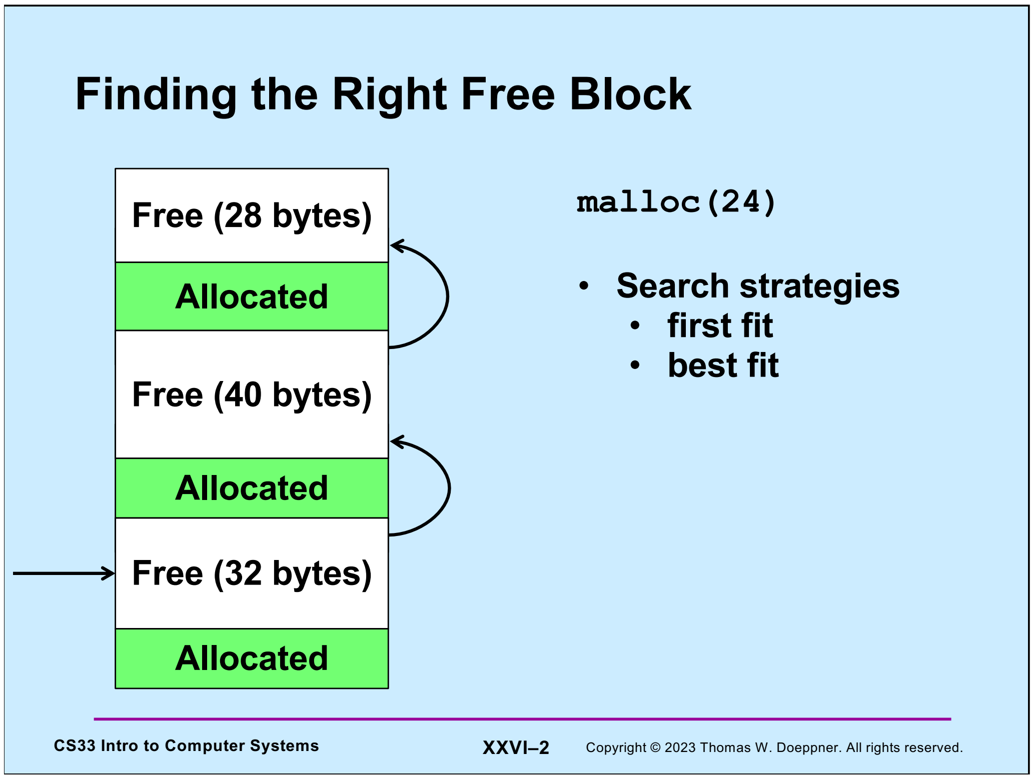
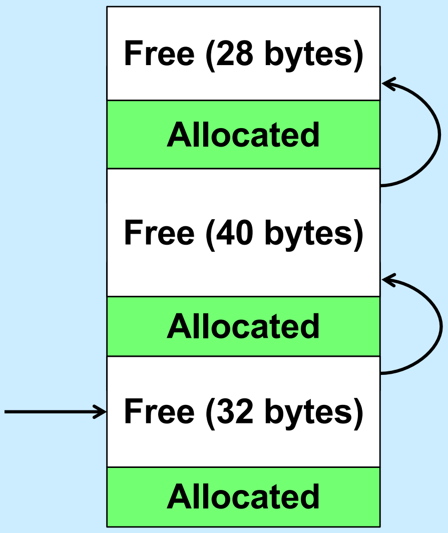
CS 33

Storage Allocation



Finding the Right Free Block



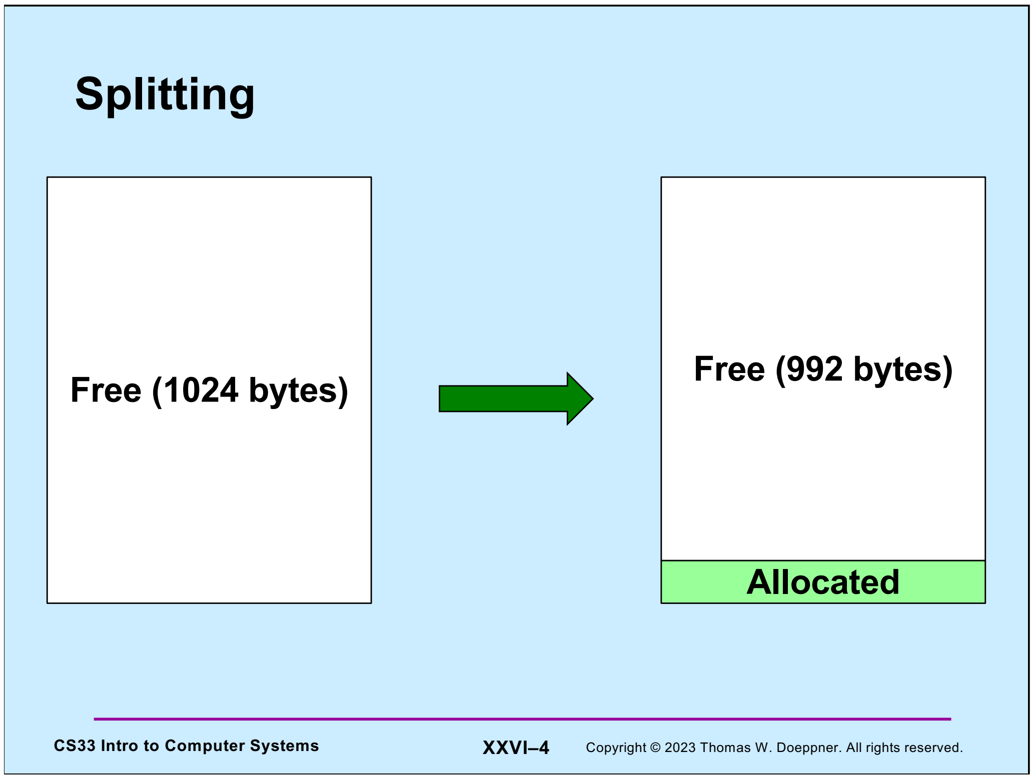
`malloc(24)`

- Search strategies
  - first fit
  - best fit

Let's assume we link together all the free blocks, as in the slide. If we'd like to allocate a block of a particular size, we need to find a free block of at least that size. What search strategy do we use to find it? An easy approach is to search, starting at the beginning of the list, until we find a block that's big enough, and use it (this is known as *first fit*). An alternative strategy, that perhaps might make better use of the available space, is to search through the entire list of free blocks and choose a block that's the smallest of those that are big enough (this is known as *best fit*).

A Problem

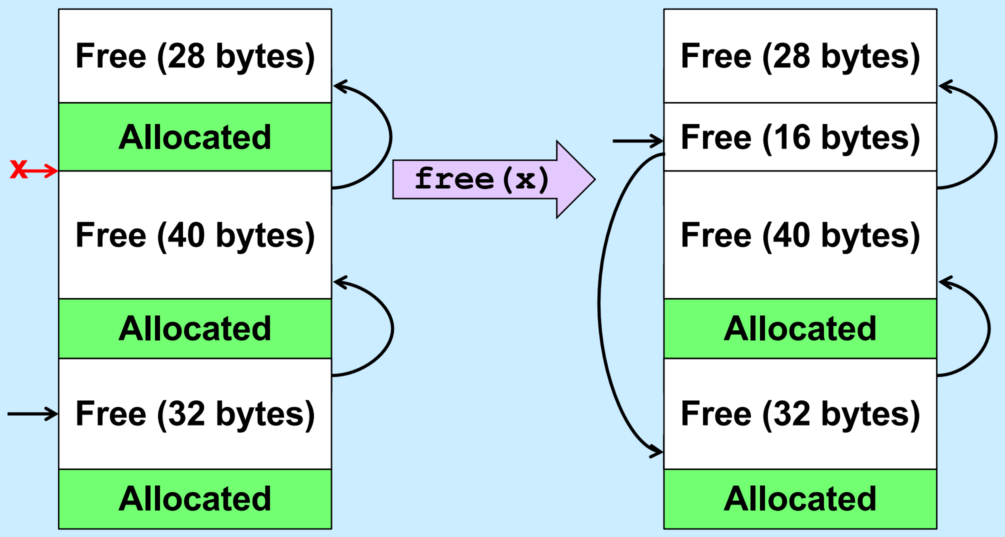
- A malloc request is for a block of 32 bytes
- The block found on the free list is 1024 bytes long
- Should malloc return a pointer to the entire 1024-byte block?



Splitting

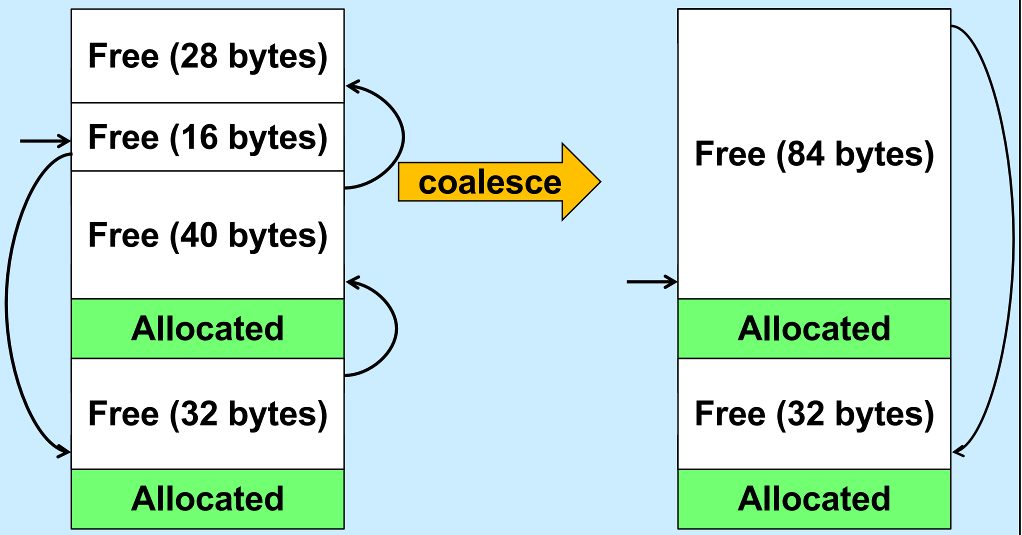
It makes no sense for malloc to return a block that's much larger than needed. Instead, it should split the block into two pieces: one piece is returned to the caller and is at least as large as was requested. The other piece is put back on the free list with an adjusted size.

Another Problem



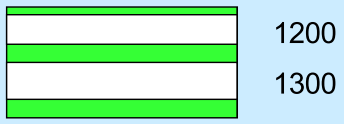
Here we've freed a block and end up with three free blocks in a row. The problem is that if we now attempt to allocate a block, say of size 52, we won't find a free block that's big enough, even though we clearly have enough space.

Coalescing



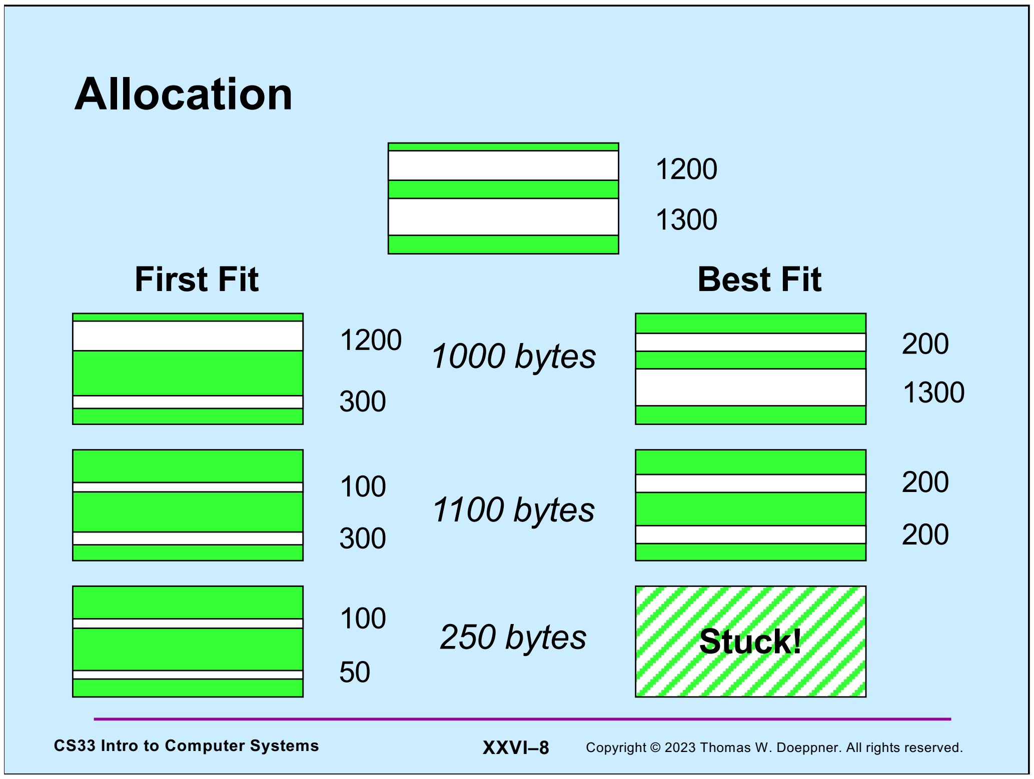
The solution is known as **coalescing**: when freeing a block, we look at adjacent blocks. if either or both adjacent blocks are free, we merge the newly freed block with its non-allocated neighbors to form a single free block whose size is the sum of the sizes of the blocks being coalesced.

Quiz 1



We have two free blocks of memory, of sizes 1300 and 1200 (appearing in that order). There are three successive requests to *malloc* for allocations of 1000, 1100, and 250 bytes. Which approach does best? (Hint: one of the two fails the last request.)

a) first fit
b) best fit

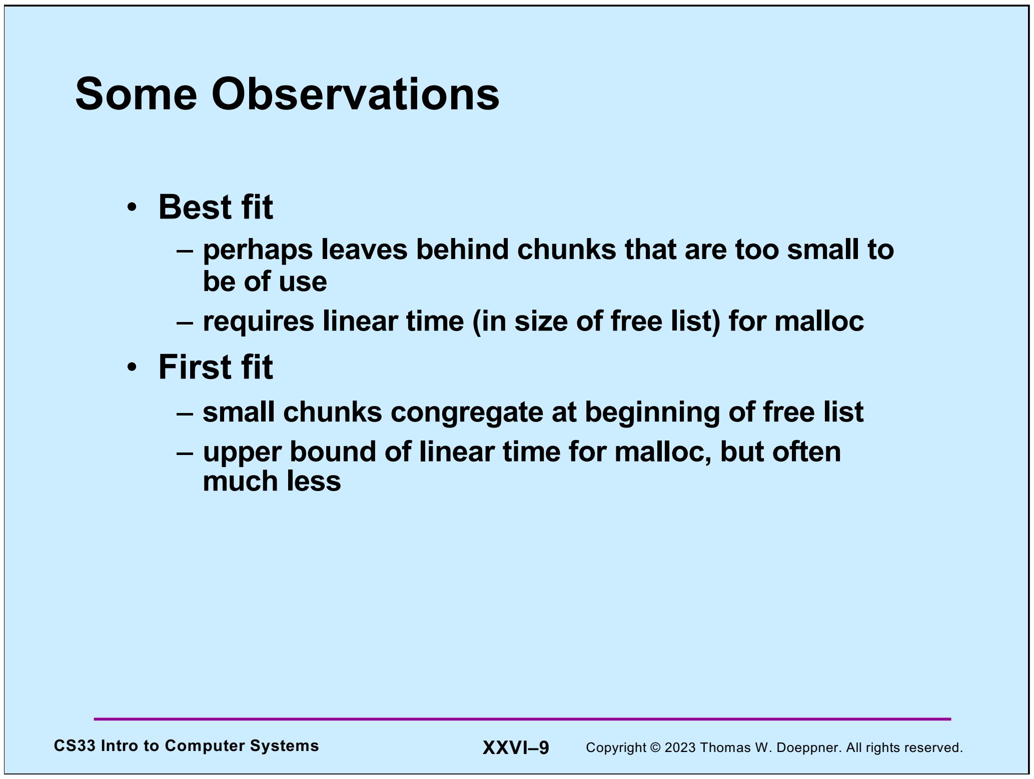


Allocation

Consider the situation in which we have one large pool of memory from which we will allocate (and to which we will liberate) variable-sized pieces of memory. Assume that we are currently in the situation shown at the top of the picture: two unallocated areas of memory are left in the pool — one of size 1300 bytes, the other of size 1200 bytes. We wish to process a series of allocation requests, and will try out two different algorithms. The first is known as **first fit** — an allocation request is taken from the first area of memory that is large enough to satisfy the request. The second is known as **best fit** — the request is taken from the smallest area of memory that is large enough to satisfy the request. On the principle that whatever requires the most work must work the best, one might think that best fit would be the algorithm of choice.

The picture illustrates a case in which first fit behaves better than best fit. We first allocate 1000 bytes. Under the first-fit approach (shown on the left side), this allocation is taken from the topmost region of free memory, leaving behind a region of 300 bytes of still unallocated memory. With the best-fit approach (shown on the right side), this allocation is taken from the bottommost region of free memory, leaving behind a region of 200 bytes of still-unallocated memory. The next allocation is for 1100 bytes. Under first fit, we now have two regions of 300 bytes and 100 bytes. Under best fit, we have two regions of 200 bytes. Finally, there is an allocation of 250 bytes. Under first fit this leaves behind two regions of 50 bytes and 100 bytes, but the allocation cannot be handled under best fit — neither remaining region is large enough.

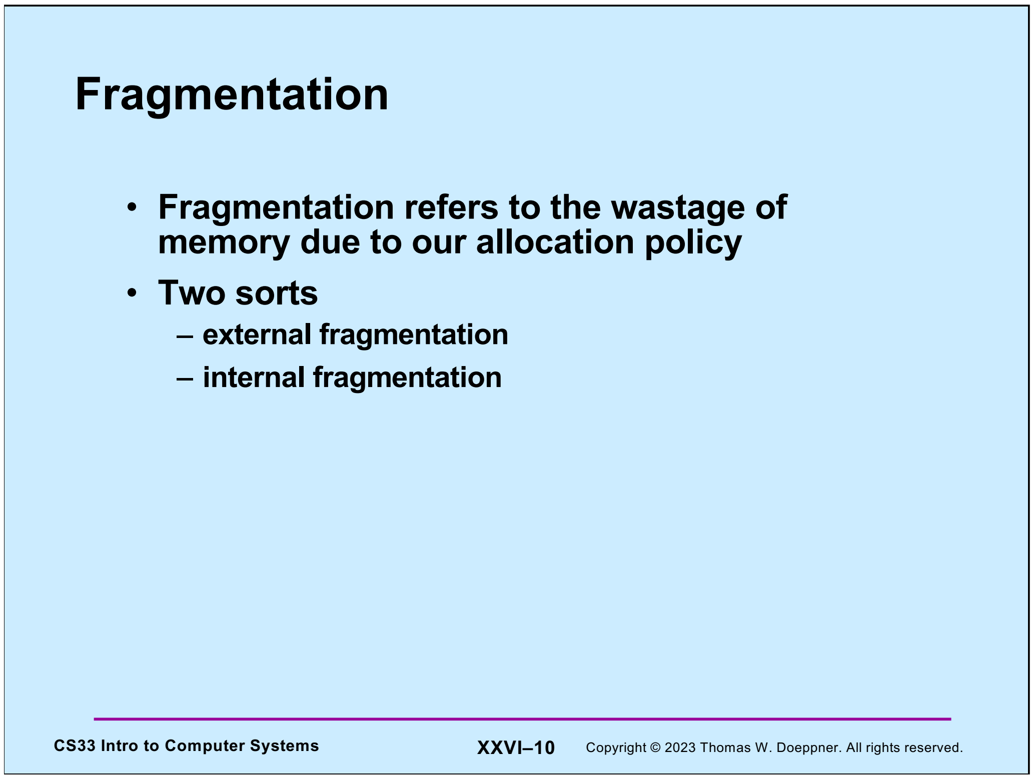
This example comes from the classic book, **The Art of Computer Programming, Vol. 1, Fundamental Algorithms**, by Donald Knuth.



Some Observations

- Best fit
  - perhaps leaves behind chunks that are too small to be of use
  - requires linear time (in size of free list) for malloc
- First fit
  - small chunks congregate at beginning of free list
  - upper bound of linear time for malloc, but often much less

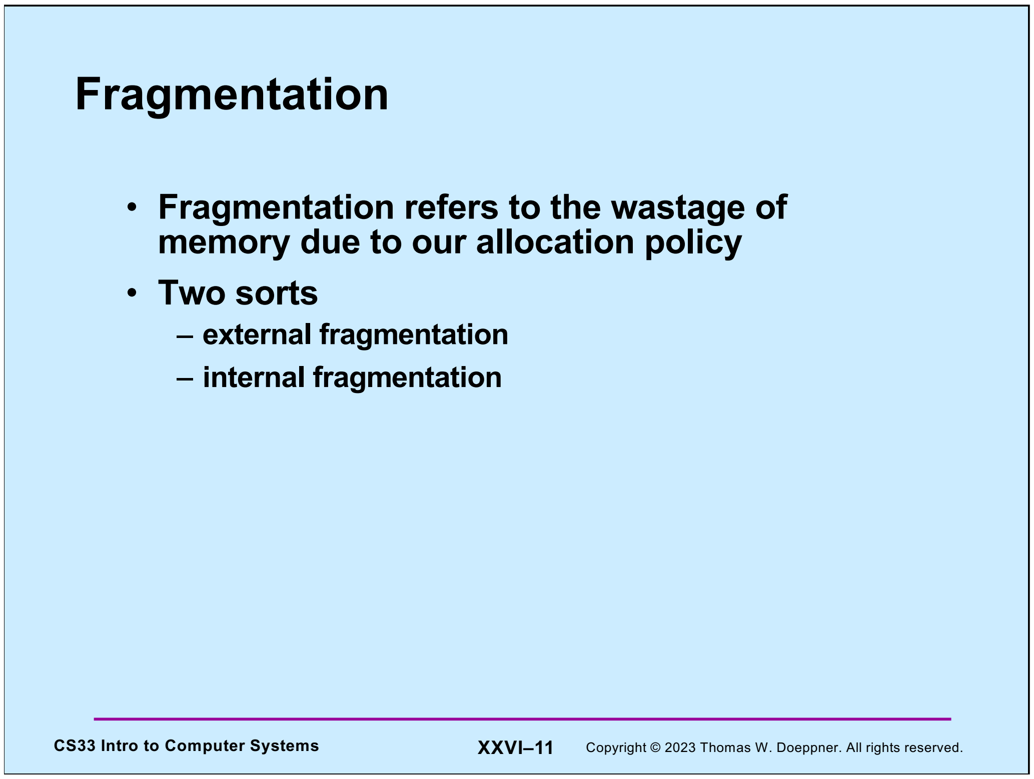
Neither first fit nor best fit is ideal. In practice, both work reasonably well in most situations. First fit has the advantage in that it doesn't always require looking at the sizes of all free blocks of memory.



Fragmentation

- Fragmentation refers to the wastage of memory due to our allocation policy
- Two sorts
  - external fragmentation
  - internal fragmentation

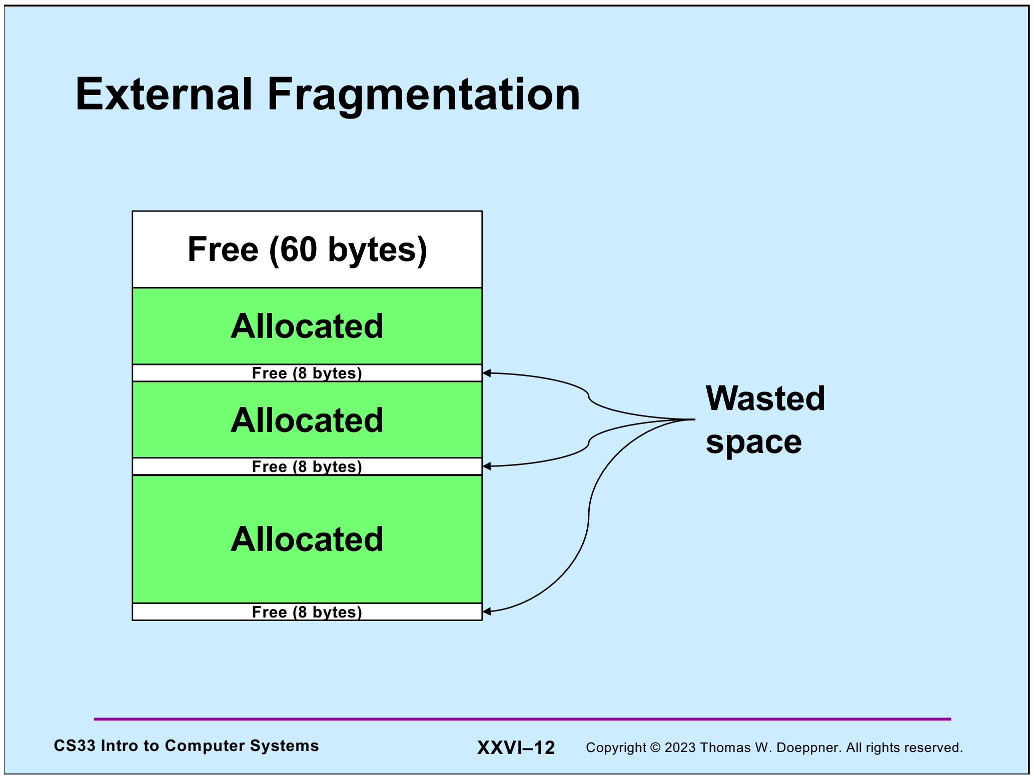
When we analyze the behavior of our storage-allocation approaches, we're concerned about **fragmentation** – how much storage is wasted.



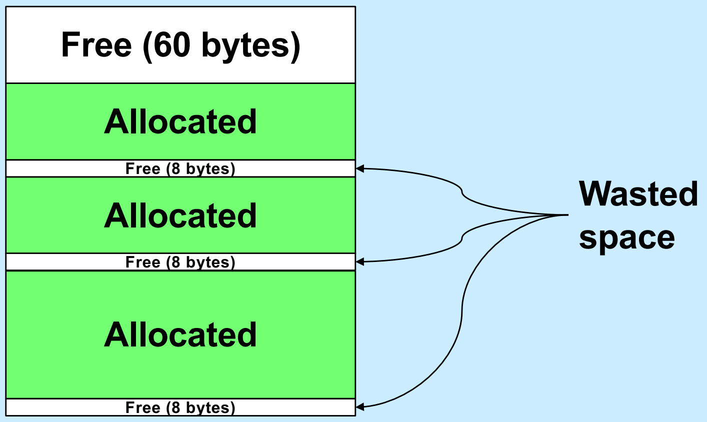
Fragmentation

- Fragmentation refers to the wastage of memory due to our allocation policy
- Two sorts
  - external fragmentation
  - internal fragmentation

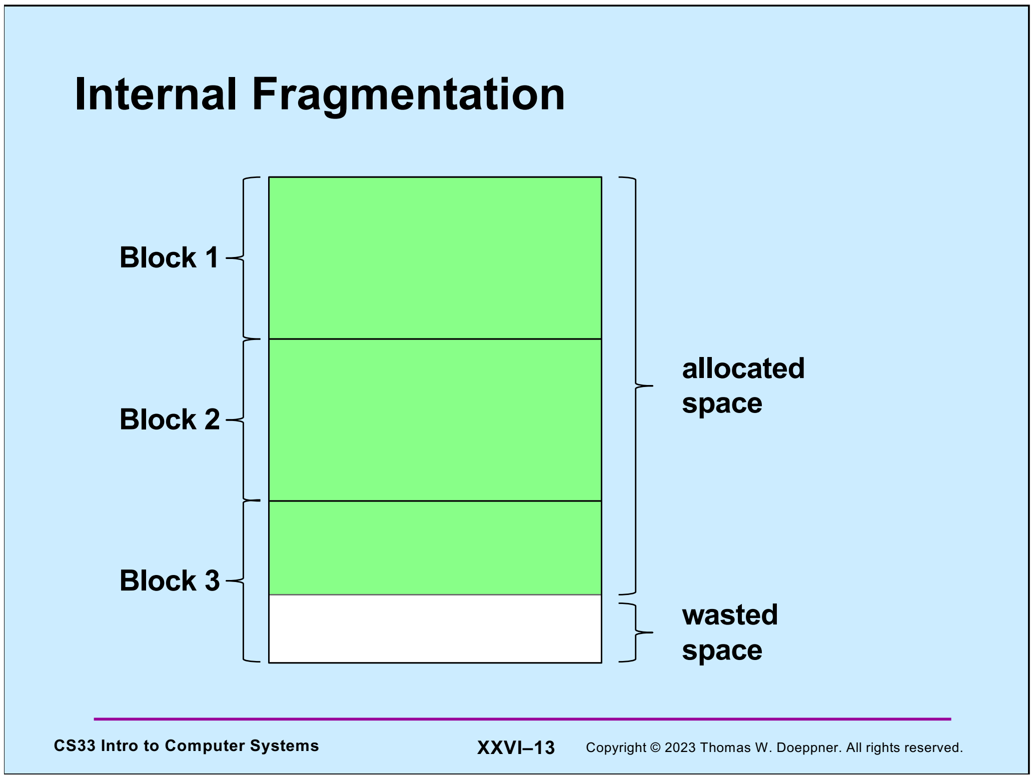
When we analyze the behavior of our storage-allocation approaches, we're concerned about **fragmentation** – how much storage is wasted.



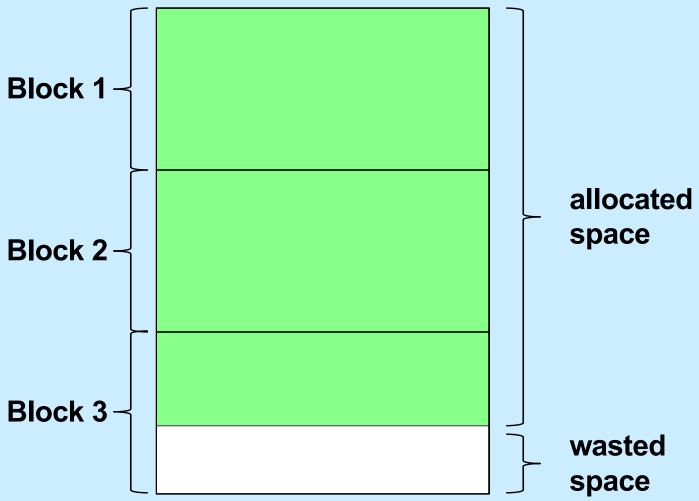
External Fragmentation



External fragmentation is when our allocation policy produces free blocks that are too small to be of use.

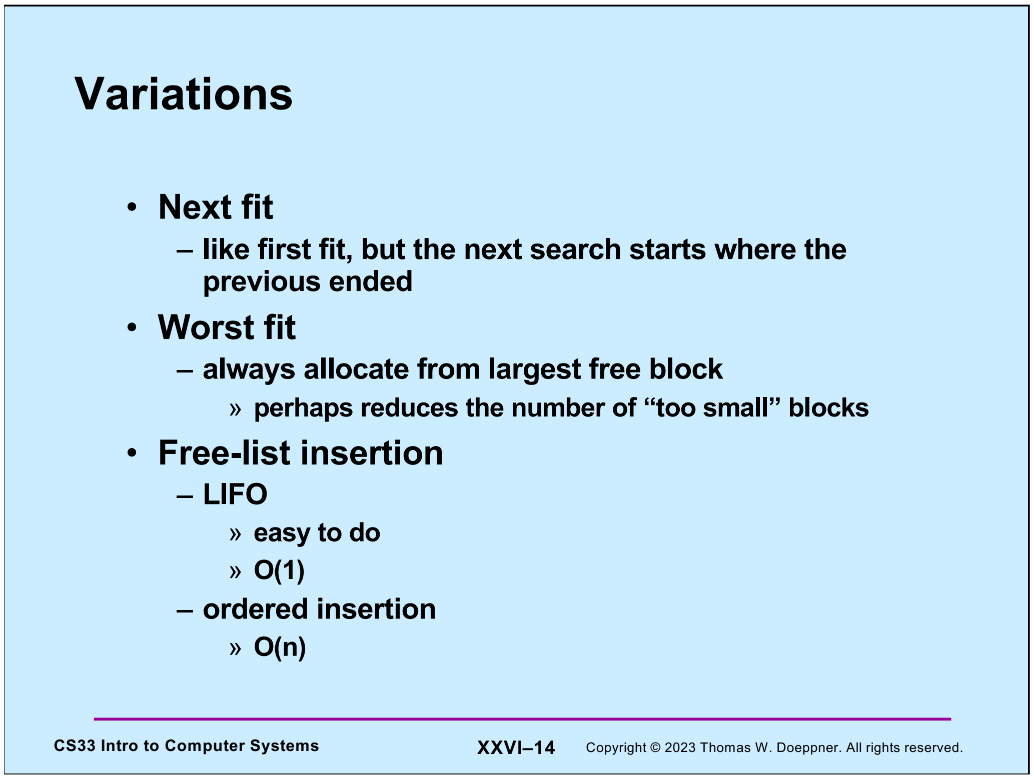


Internal Fragmentation



While this isn't important for this course, internal fragmentation occurs when memory is allocated in fixed-size blocks, say 4k bytes each. If we allocate space for a data structure whose size is not a multiple of the block size, the wasted space is said to be due to **internal fragmentation**.

Note that for the malloc project (coming out soon), we will do next fit with LIFO insertion.

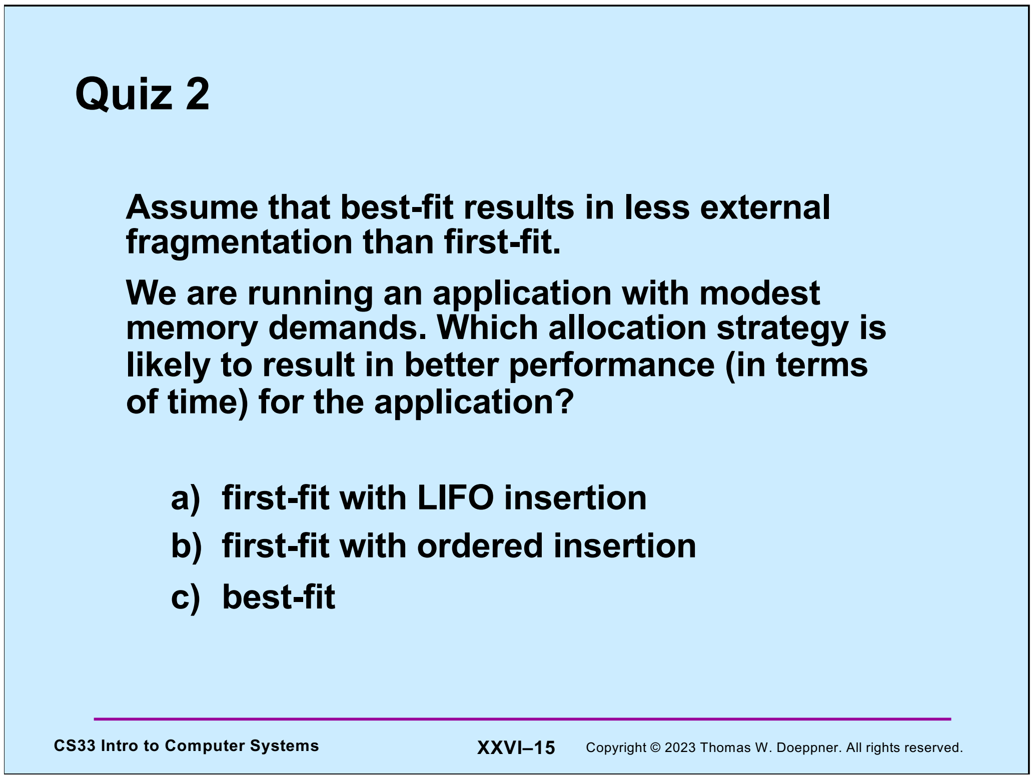


Variations

- **Next fit**
  - like first fit, but the next search starts where the previous ended
- **Worst fit**
  - always allocate from largest free block
    - » perhaps reduces the number of “too small” blocks
- **Free-list insertion**
  - LIFO
    - » easy to do
    - » O(1)
  - ordered insertion
    - » O(n)

LIFO (last in first out) insertion simply means that items are always inserted at the beginning of the free list. With ordered insertion, we keep the free list ordered by the size of the block (from smallest to largest). Note that LIFO insertion tends to put larger blocks at the beginning of the free list, which is good for first-fit allocation.

Note that for the malloc project (coming out soon), we will do first fit with LIFO insertion.



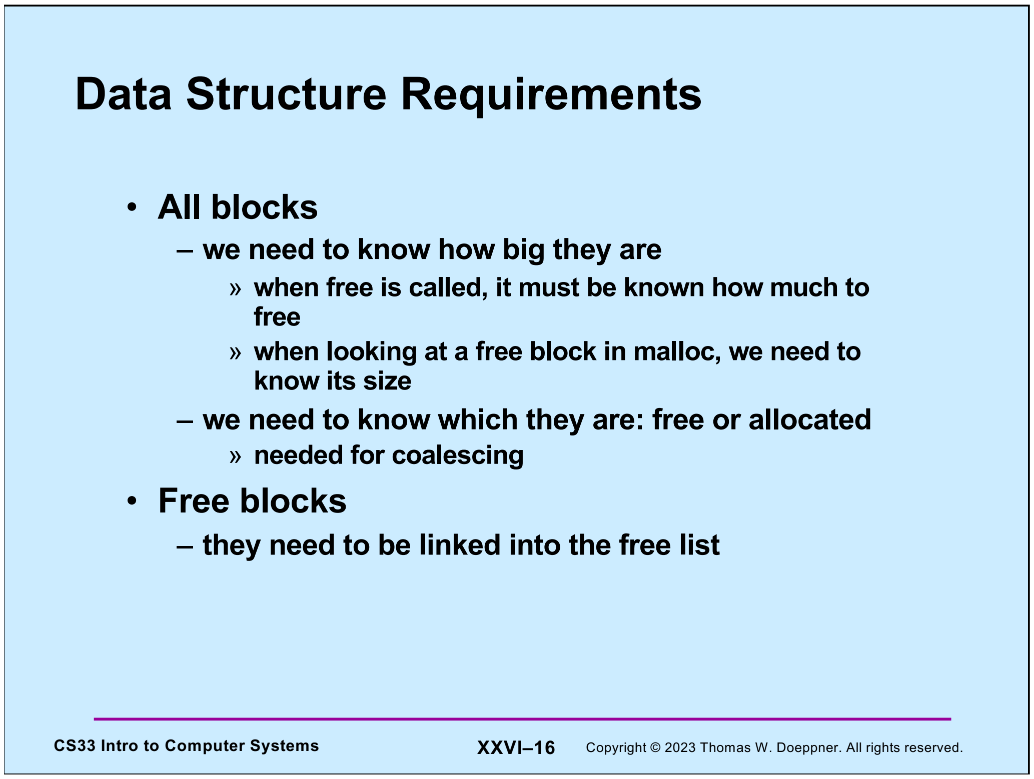
Quiz 2

Assume that best-fit results in less external fragmentation than first-fit.

We are running an application with modest memory demands. Which allocation strategy is likely to result in better performance (in terms of time) for the application?

a) first-fit with LIFO insertion
b) first-fit with ordered insertion
c) best-fit

By "modest memory demands", we mean that **malloc**, **free**, and related functions are not called frequently.

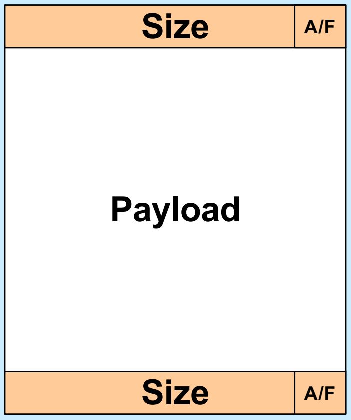


Data Structure Requirements

- All blocks
  - we need to know how big they are
    - » when free is called, it must be known how much to free
    - » when looking at a free block in malloc, we need to know its size
  - we need to know which they are: free or allocated
    - » needed for coalescing
- Free blocks
  - they need to be linked into the free list

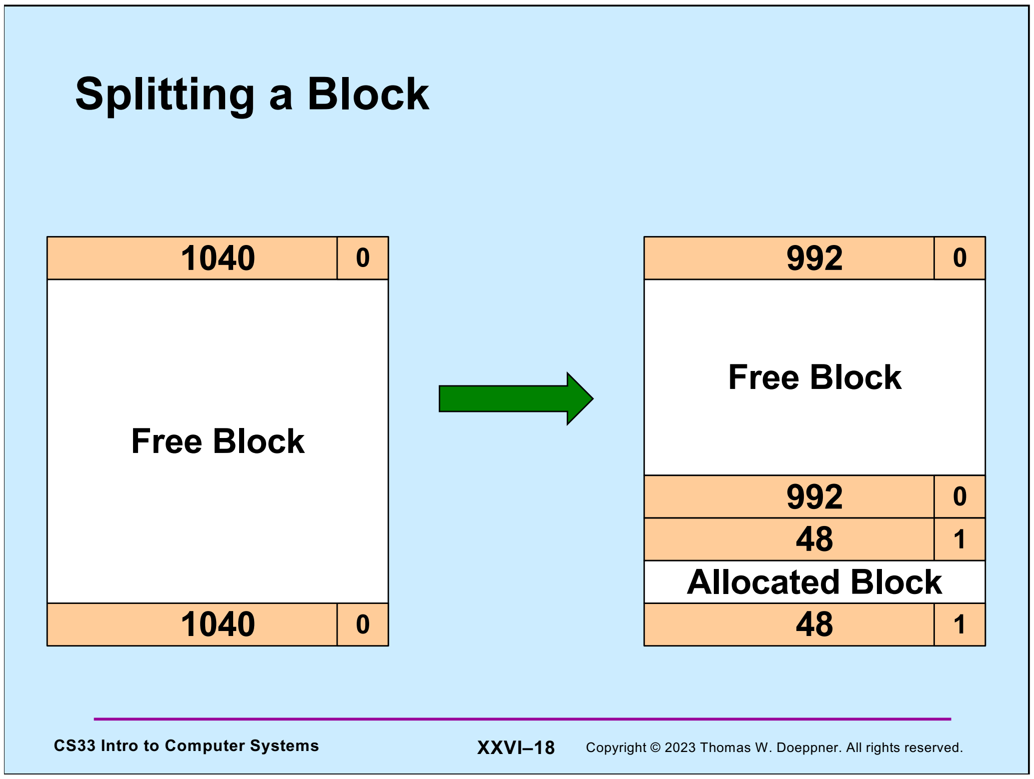
It's now time to design the data structures we need to represent our "heap" – the dynamic memory region.

Solution: Boundary Tags

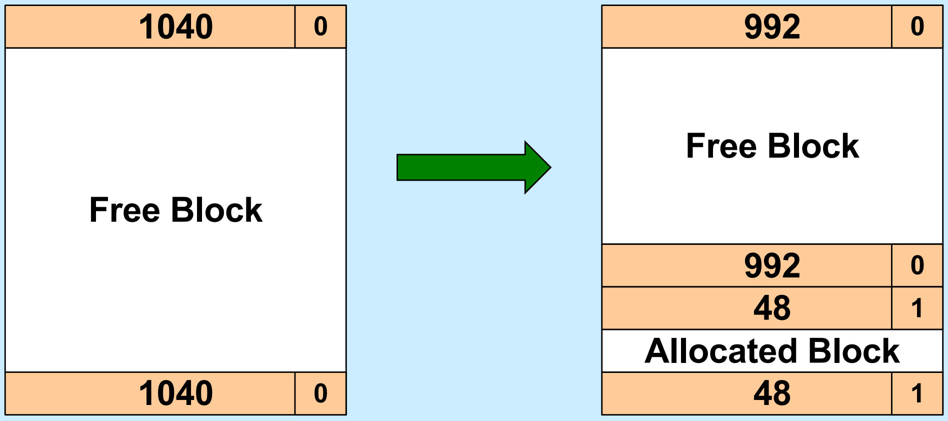


One solution (which we use in the malloc assignment) is the boundary tags approach. Here we have a fixed overhead for each block of memory (whether free or allocated) that indicates its size and whether it's free. So that we can determine if adjacent blocks are free (and what their sizes are), we put this information at each end of the block. The non-overhead portion of the block (which is available to hold data) is called the **payload**.

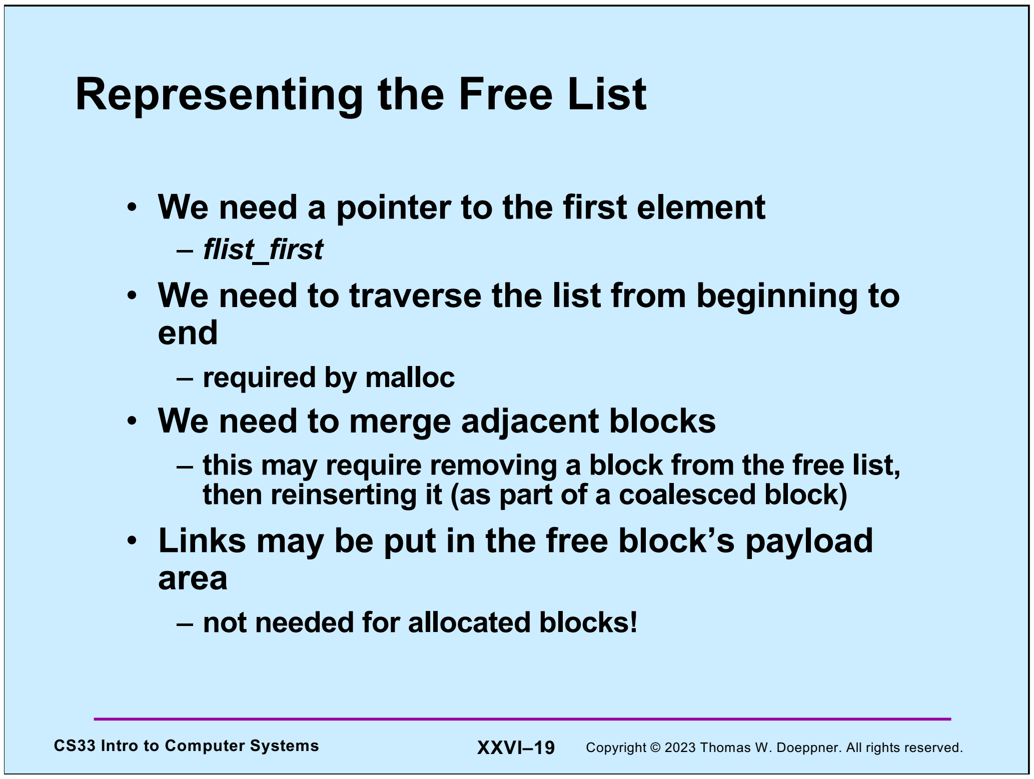
One could set the **size** to be the size of the entire block, or the size of just the payload – either way can be made to work. We find it more convenient for the **size** to be that of the entire block. Thus the size of the payload is **size** minus the amount of memory required to hold the boundary tags (in our implementation, each boundary tag (containing size and the allocated-or-free bit) is a **long**; thus the total amount of memory used for the boundary tags is 16 bytes).



Splitting a Block



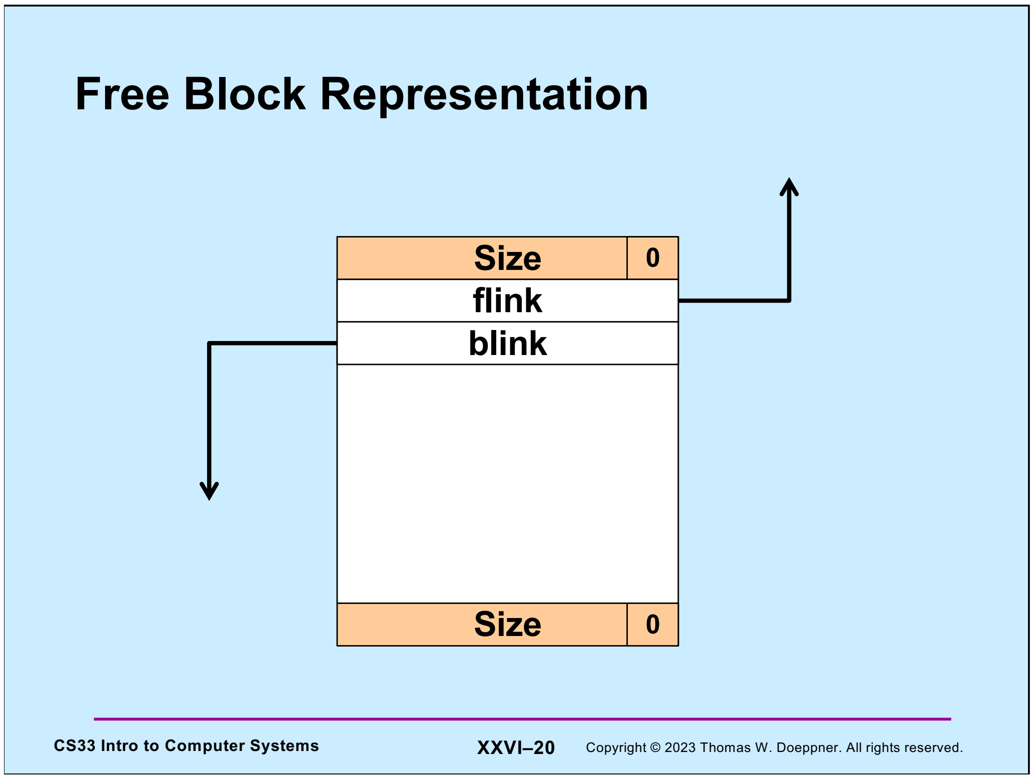
Splitting a block is straightforward. We take a block that was previously free and divide it into two blocks – an allocated block that's big enough to hold the storage request, and the remainder represented as a free block.



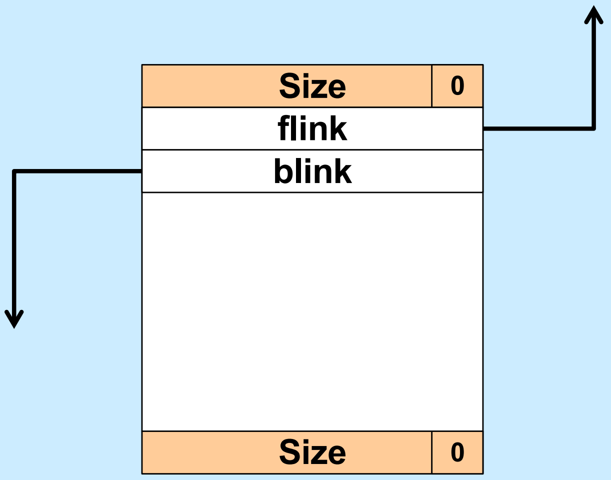
Representing the Free List

- We need a pointer to the first element
  - *flist_first*
- We need to traverse the list from beginning to end
  - required by malloc
- We need to merge adjacent blocks
  - this may require removing a block from the free list, then reinserting it (as part of a coalesced block)
- Links may be put in the free block's payload area
  - not needed for allocated blocks!

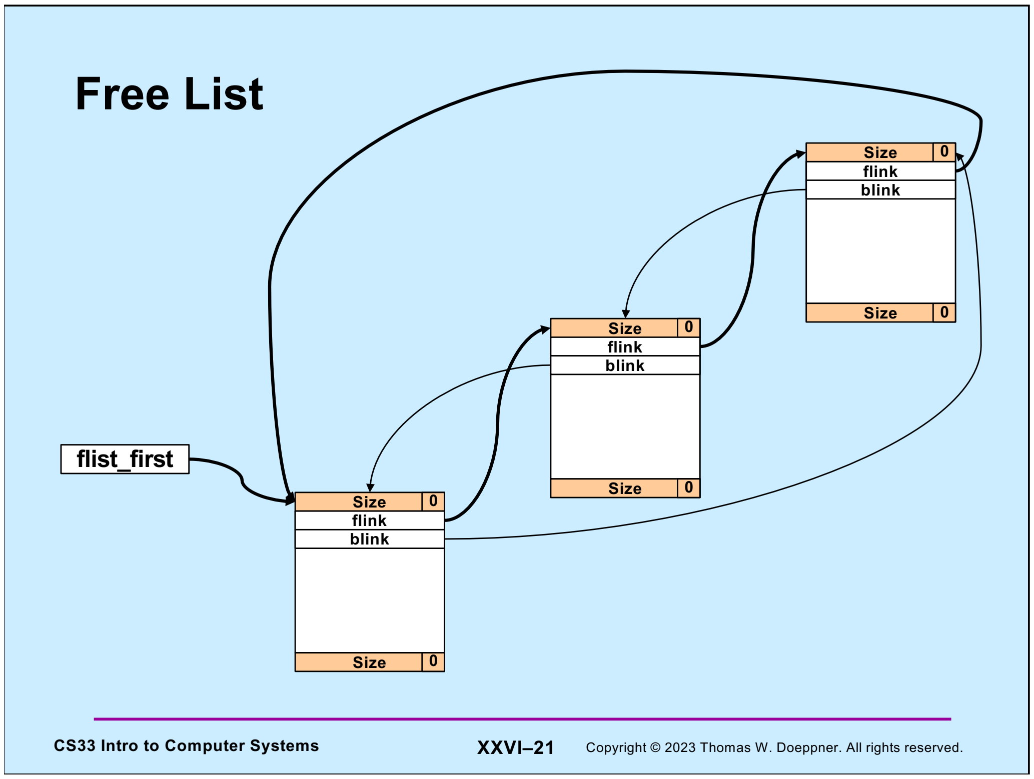
The global variable **flist_first** is a pointer to the first item in the free list (and is null if the free list is empty).



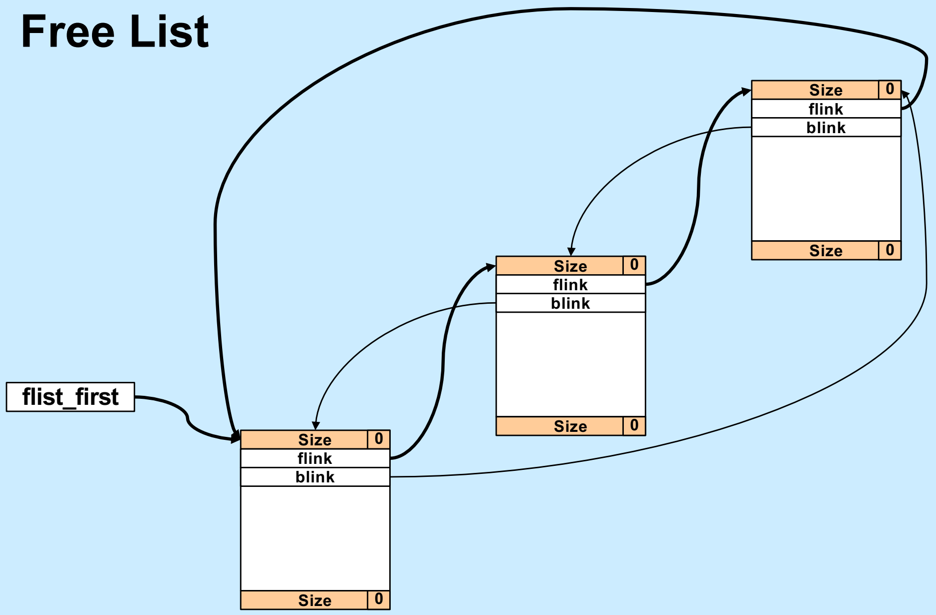
Free Block Representation



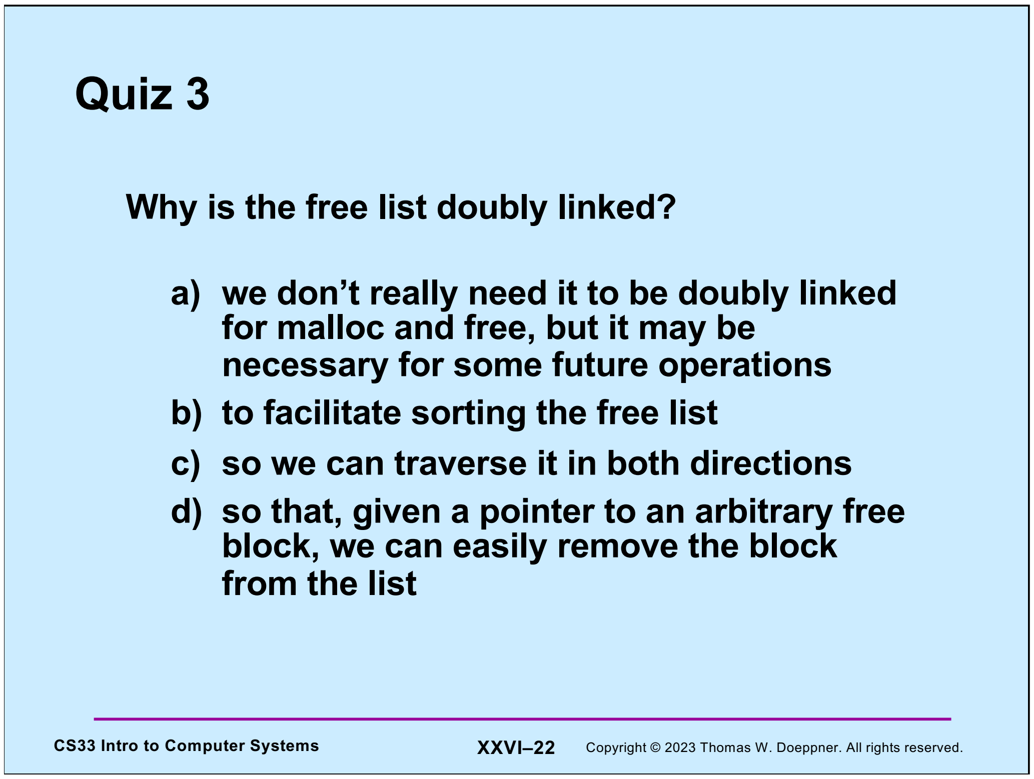
Here's our representation of a free block. Note that it has both a forward link (**flink**) and a backwards link (**blink**) – thus the free list is doubly linked.



Free List



The free list is a circular, doubly linked list.



Quiz 3

Why is the free list doubly linked?

a) we don’t really need it to be doubly linked for malloc and free, but it may be necessary for some future operations
b) to facilitate sorting the free list
c) so we can traverse it in both directions
d) so that, given a pointer to an arbitrary free block, we can easily remove the block from the list

If the course had a final exam, this question would definitely be on it. Make sure you understand the answer. It will come up again in the course (and count towards your grade!).

Quiz 4

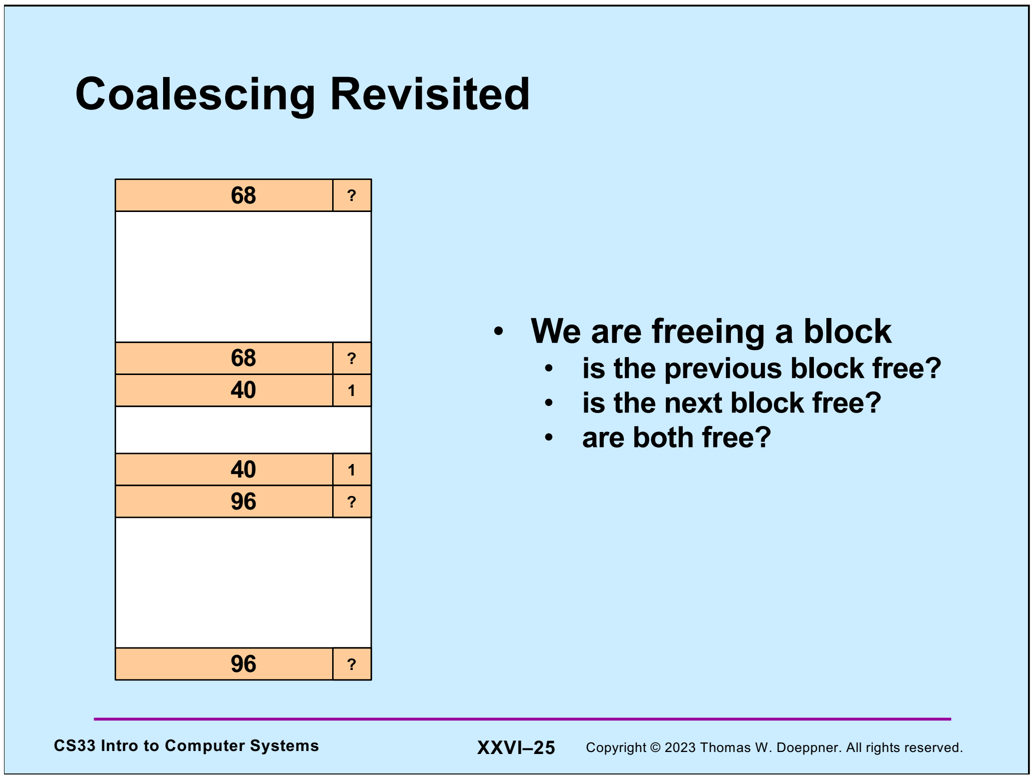
Why is the free list circular?

a) to facilitate implementing the next-fit search strategy
b) so that we don't have to special-case the the handling of the first and last list elements
c) both of the above
d) none of the above

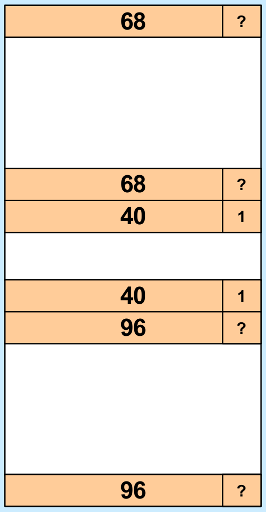
Heap ≠ Free List

- Heap
  - collection of all memory usable as dynamic storage: the dynamic portion of the address space
    - » both allocated and free
- Free list
  - those blocks of the heap that are free
    - » linked together (circular, doubly)

- Both important, but different
- Confusion: what does *next block* mean?
  - next adjacent block (next in heap)
  - next free block (next in free list)

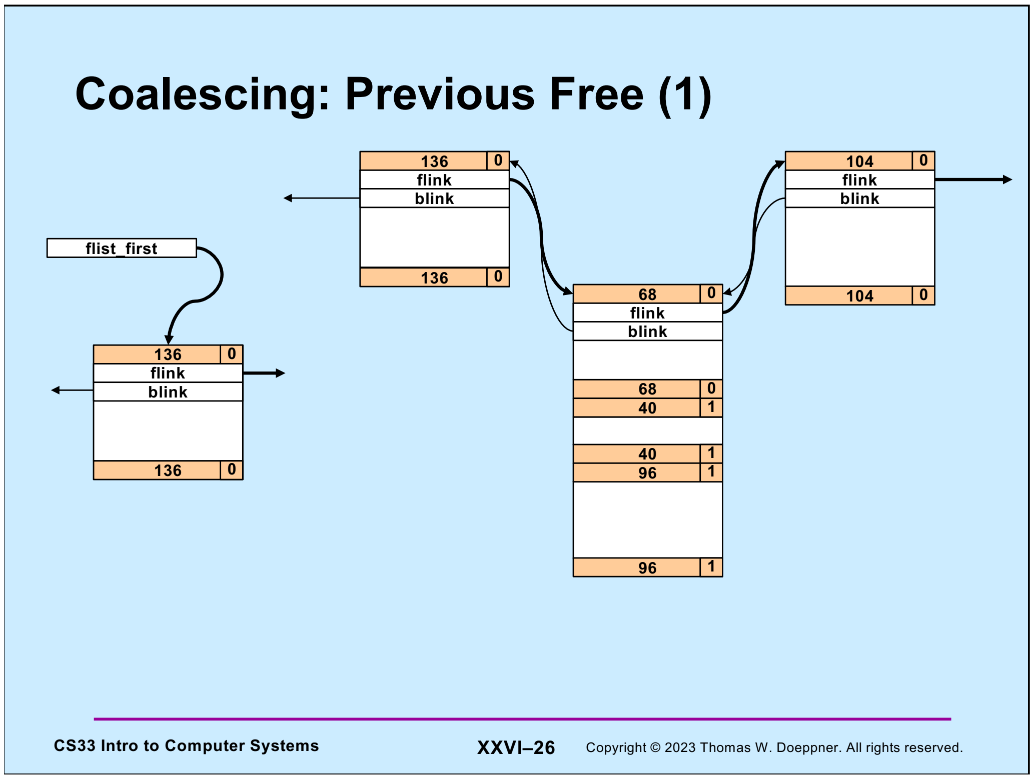


Coalescing Revisited

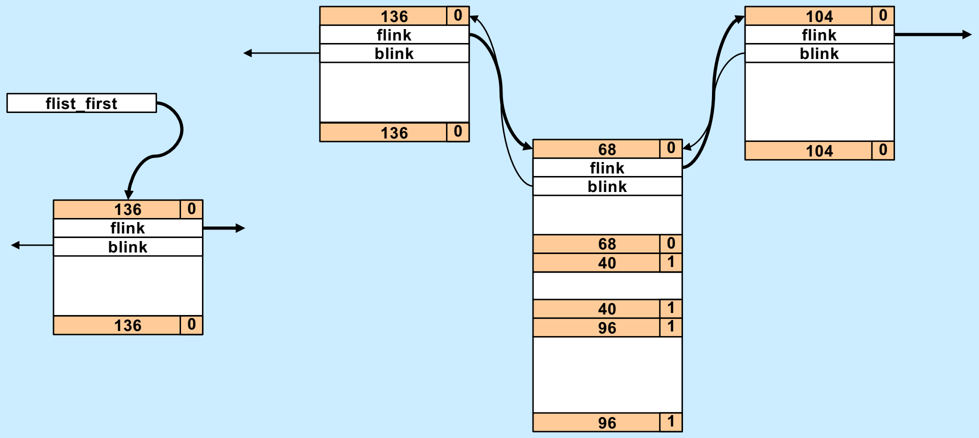


- We are freeing a block
  - is the previous block free?
  - is the next block free?
  - are both free?

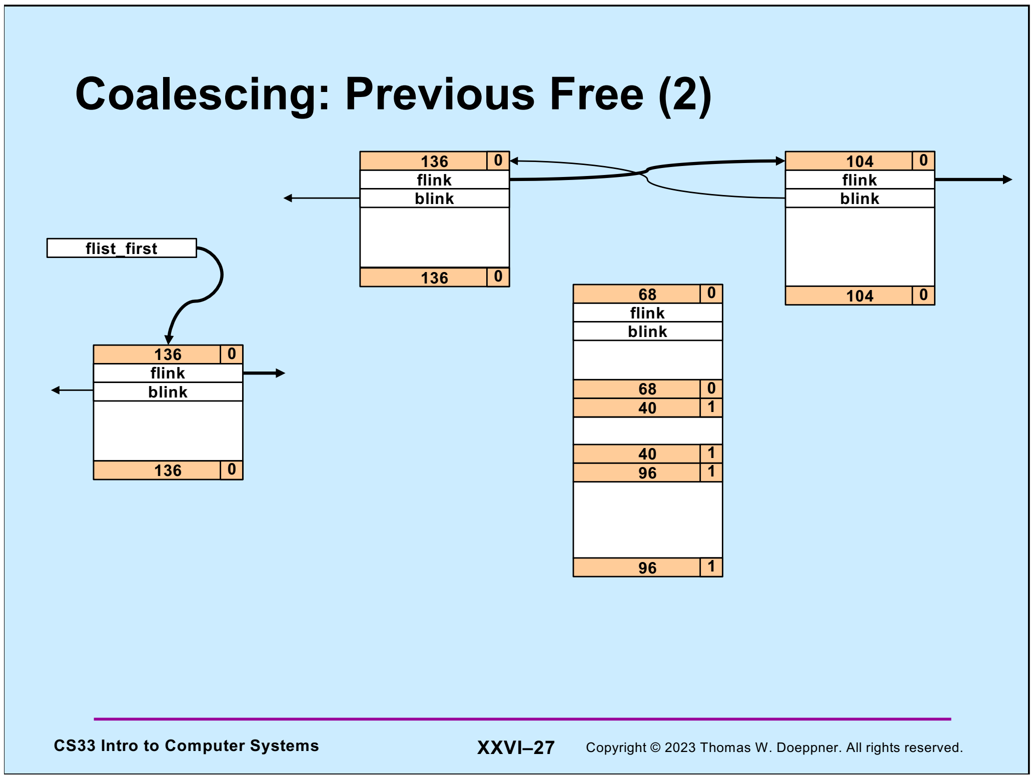
We now look at implementing the coalesce operation, given our data structures. Let's assume that we're about to free the middle block, of size 40. To handle coalescing, we need to know whether the previous block and the next block are free.



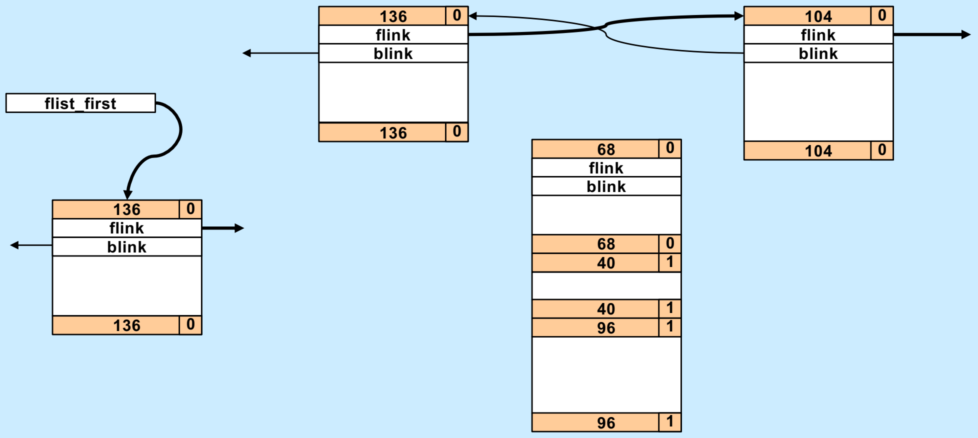
Coalescing: Previous Free (1)



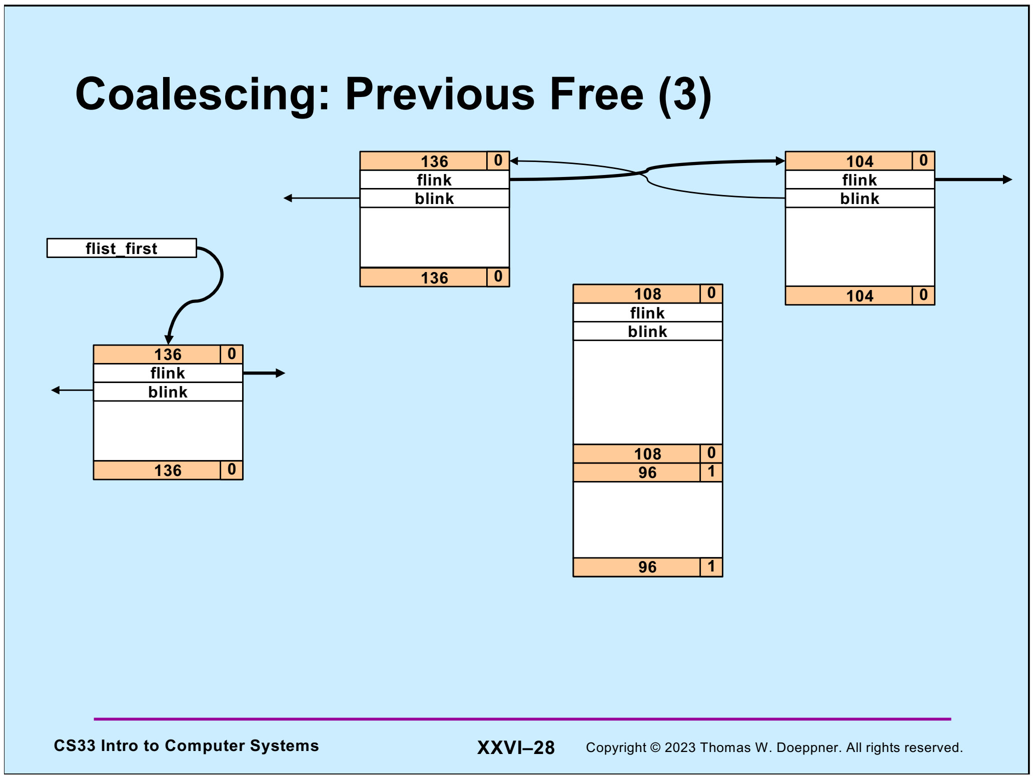
Suppose the previous block is free, but the next block is allocated. Thus, the previous block is in the free list. We'll assume it's not the first element of the free list, which is pointed to by **flist_first**.



Coalescing: Previous Free (2)

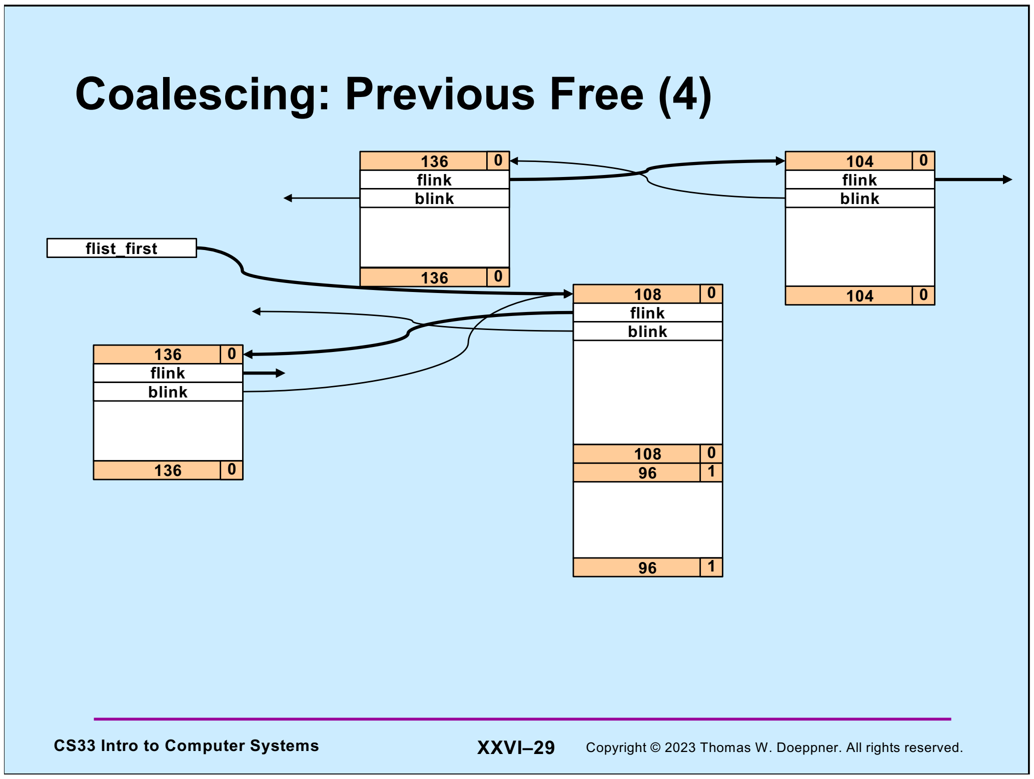


We first pull the previous block from the free list.

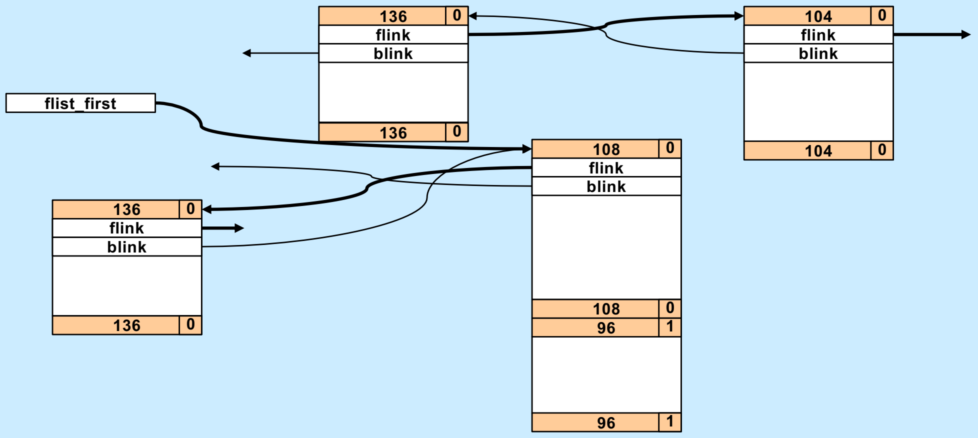


Coalescing: Previous Free (3)

We the merge the newly freed block with the previous block.



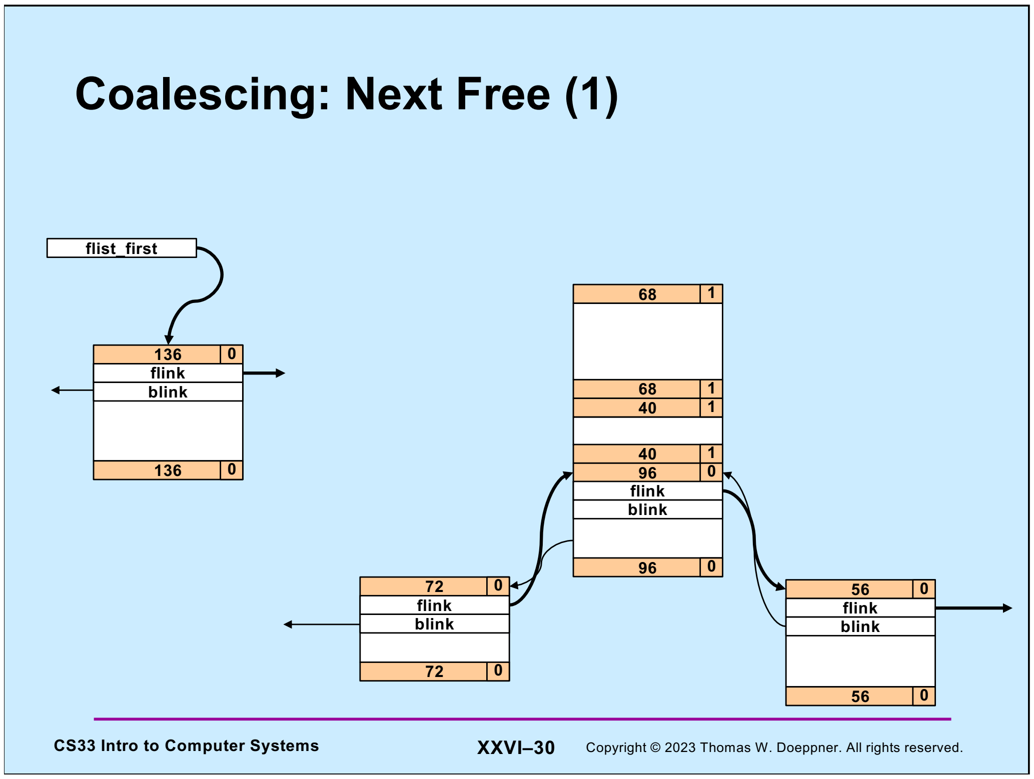
Coalescing: Previous Free (4)



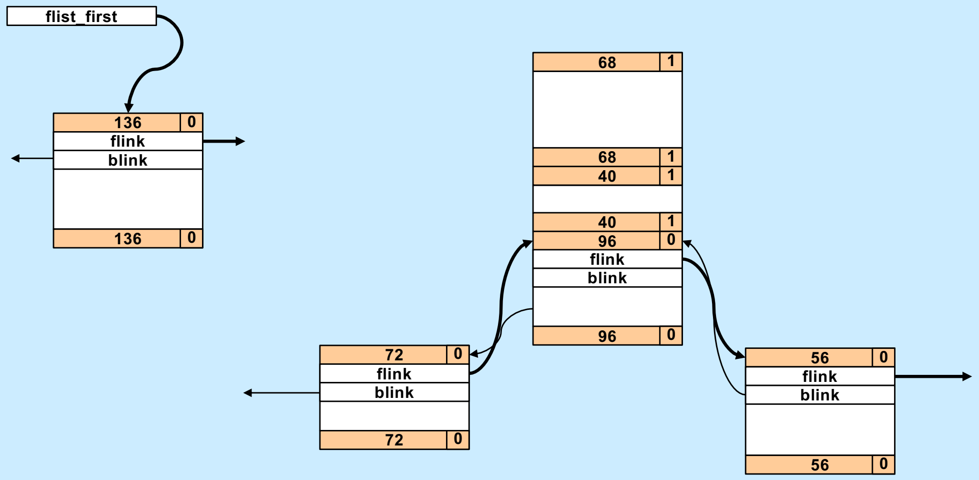
Finally, we add the merged free block to the beginning of the free list.

This, of course, is not the only way to do this. We could simply leave the previous block in the free list at its current position and increase its size so as to absorb the block being freed. This perhaps could be more efficient than what's shown in the slide, but it leads to some slight complications in the code. Feel free to do it either way in your own code.

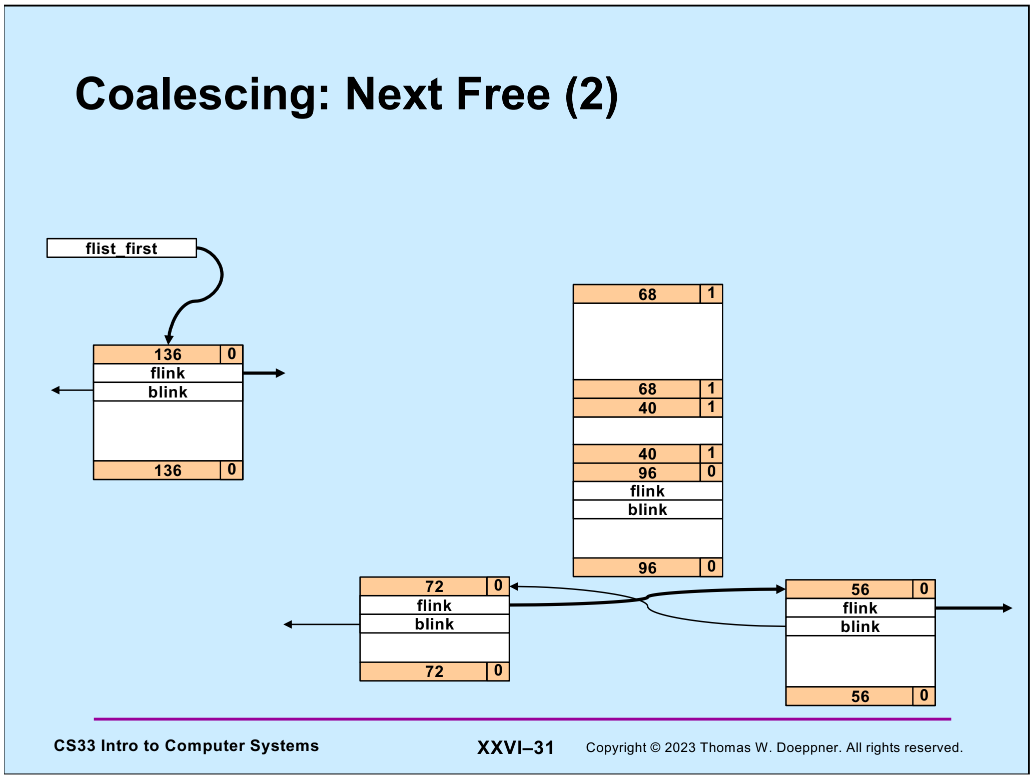
A potential advantage of implementing coalesce as done here is that it puts a potentially larger block at the beginning of the free list, possibly improving the performance of first fit.



Coalescing: Next Free (1)

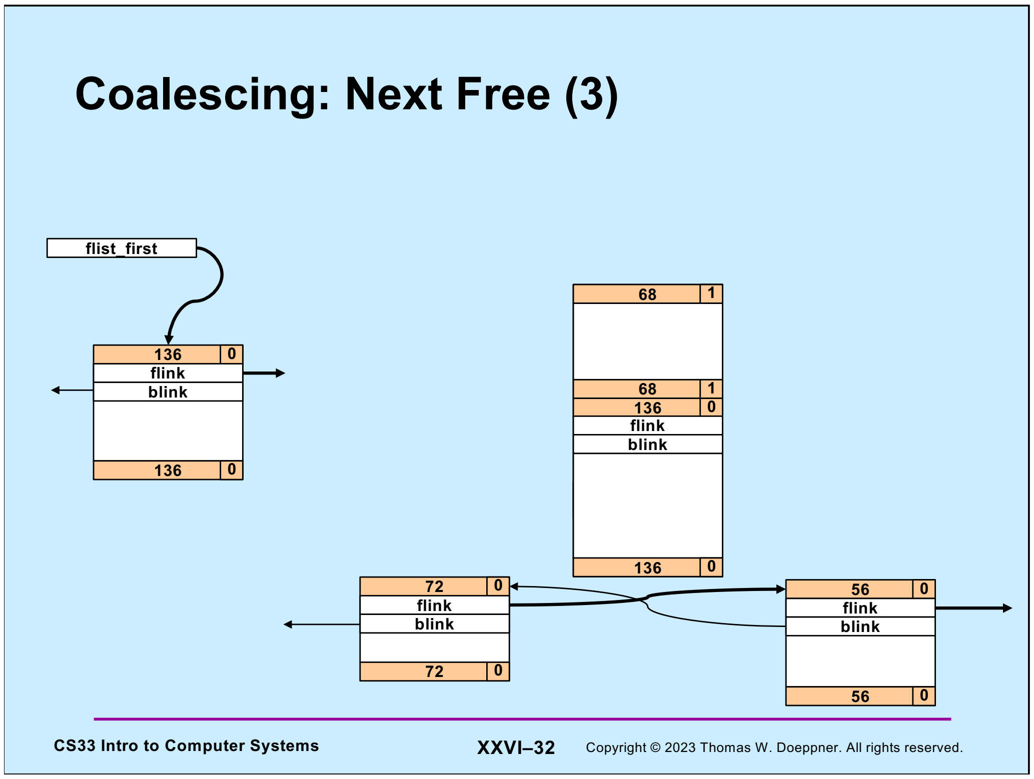


Here the previous block is allocated but the next block is free.

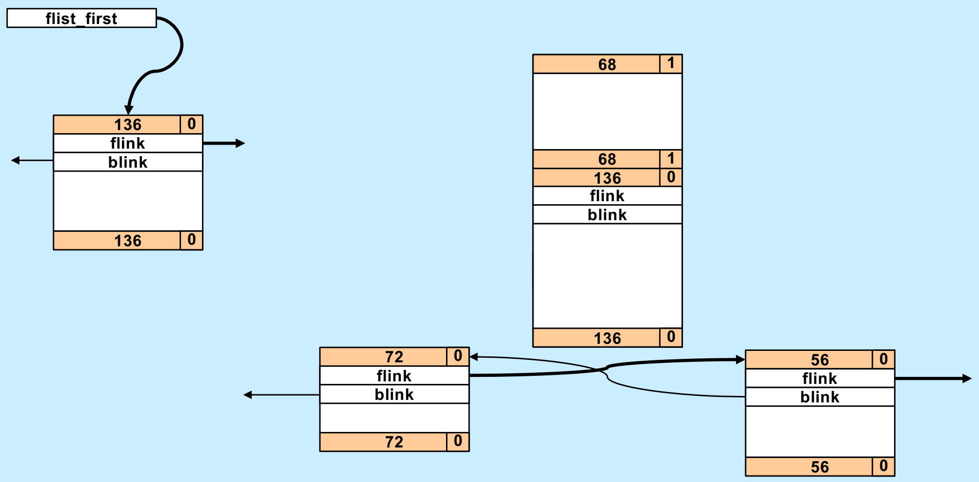


Coalescing: Next Free (2)

We first pull the next block from the free list.

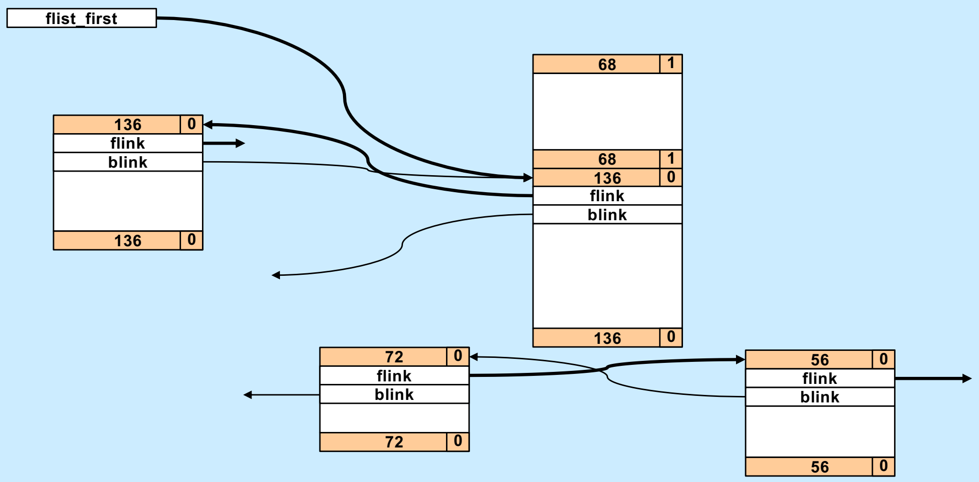


Coalescing: Next Free (3)



We then merge the block we're freeing with the next block.

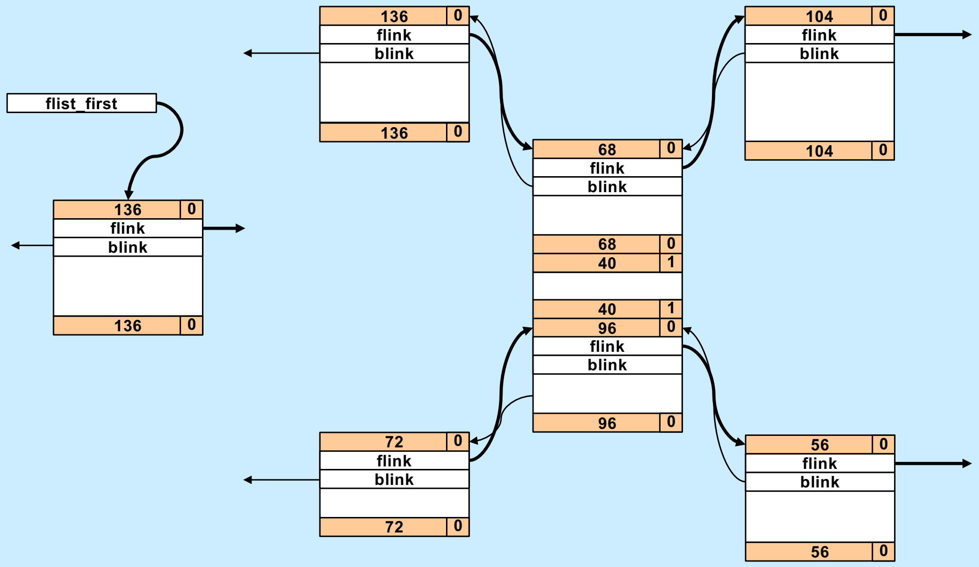
Coalescing: Next Free (4)



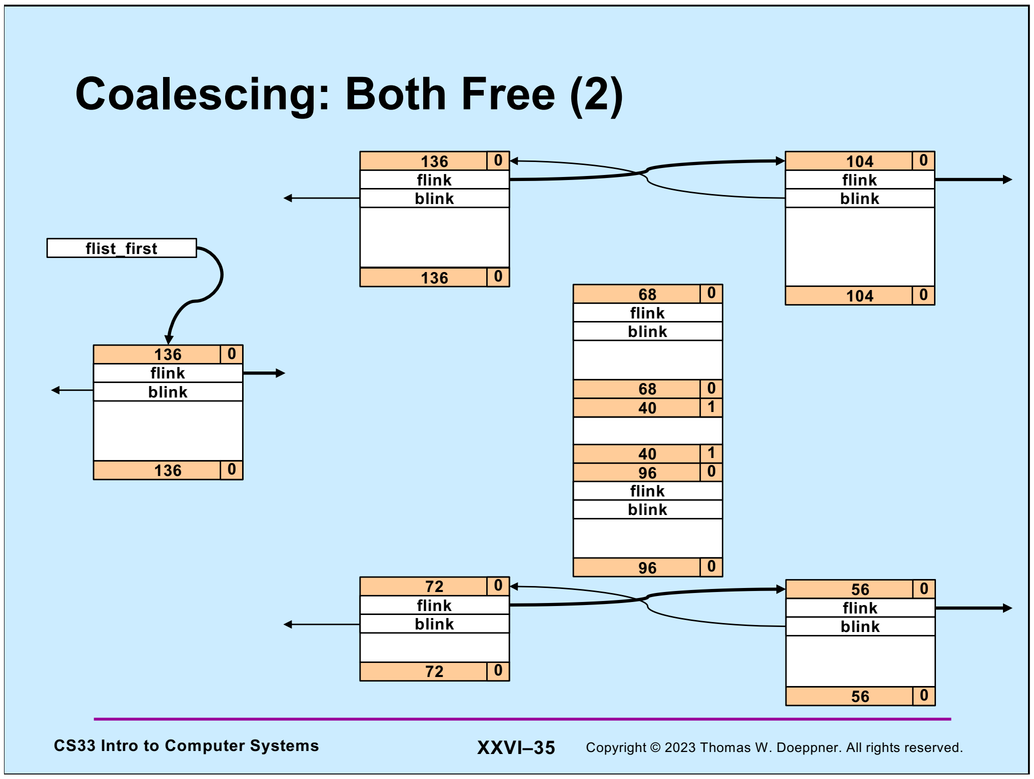
Finally, we insert the combined block into the beginning of the free list.

Again, there are other ways for doing this. In particular, one might simply replace the next block with the combined block, putting it into the free list where the next block was.

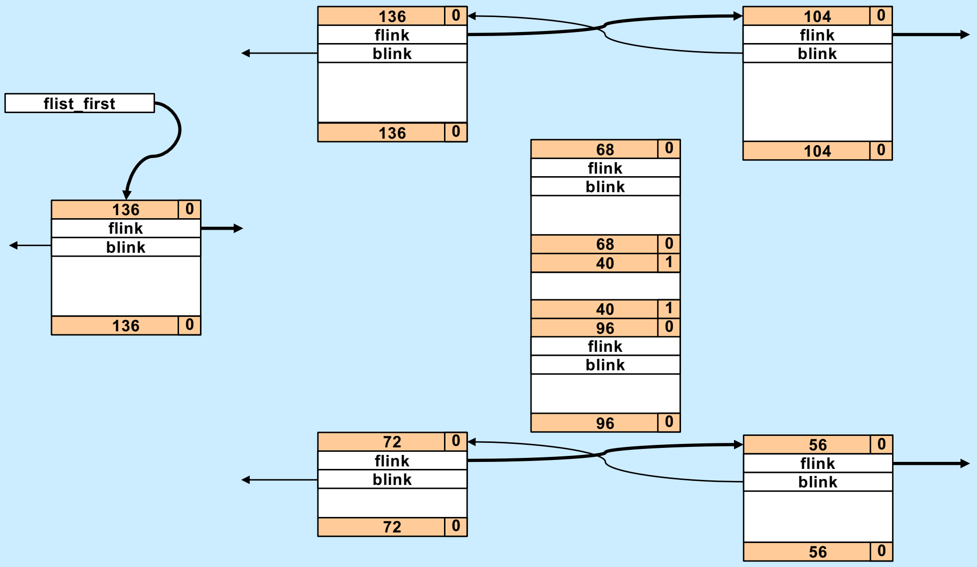
Coalescing: Both Free (1)



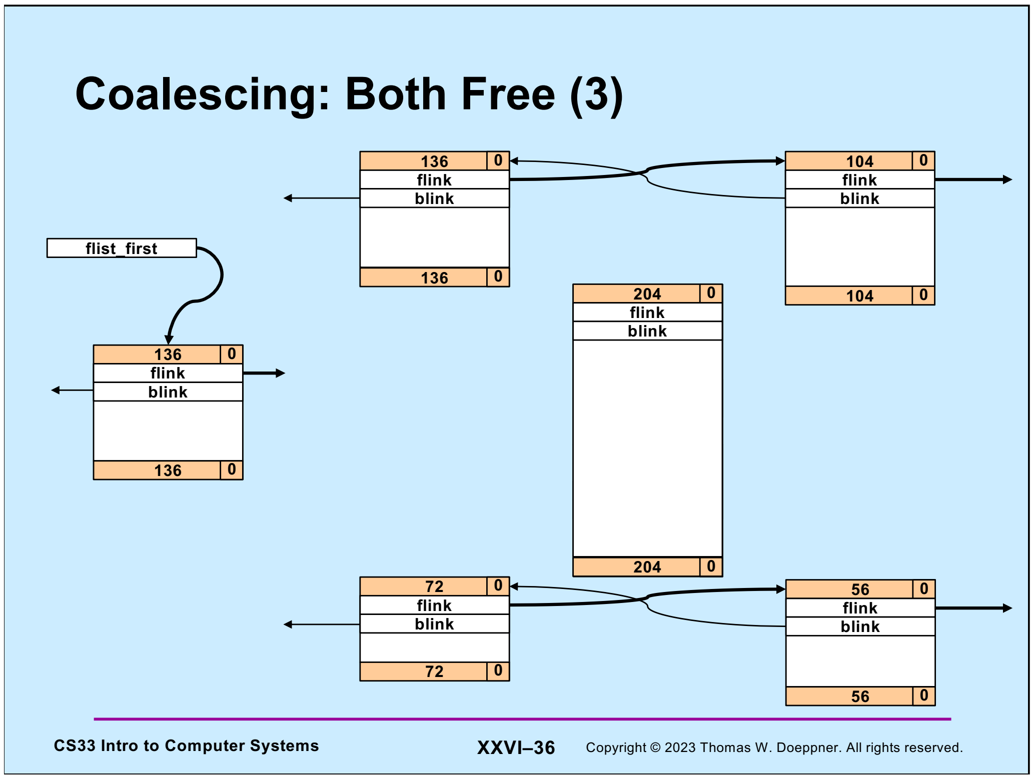
Finally, we have the case in which both the previous and the next blocks are free.



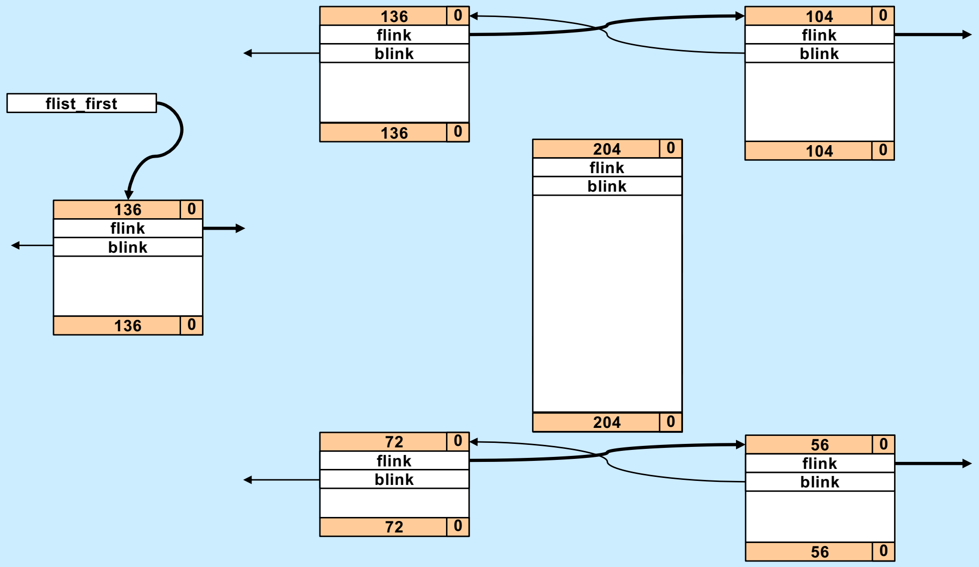
Coalescing: Both Free (2)



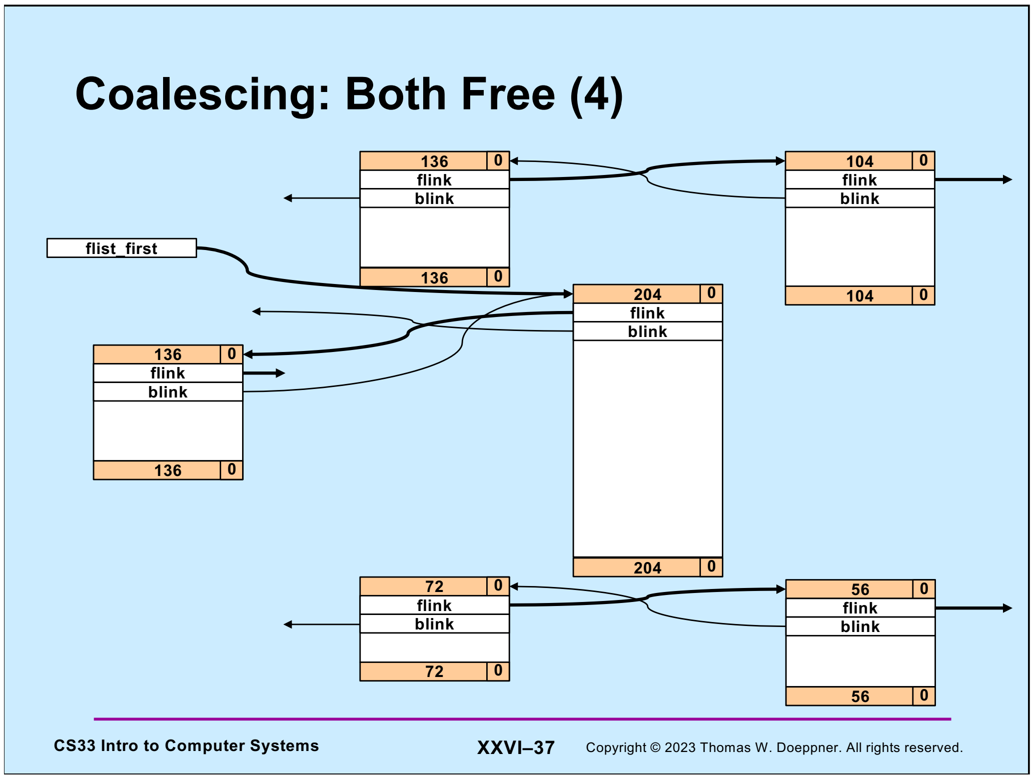
We remove both the prev and next blocks from the free list.



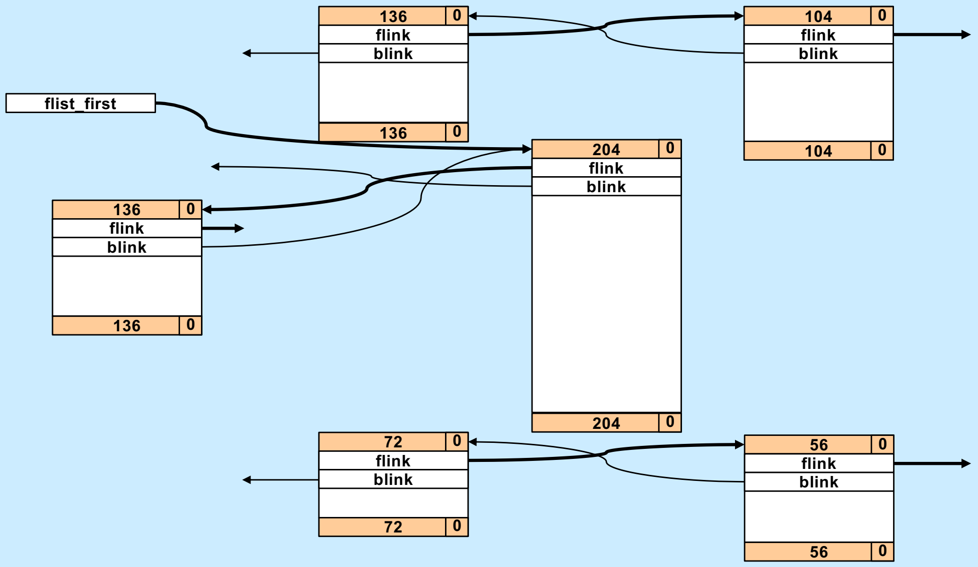
Coalescing: Both Free (3)



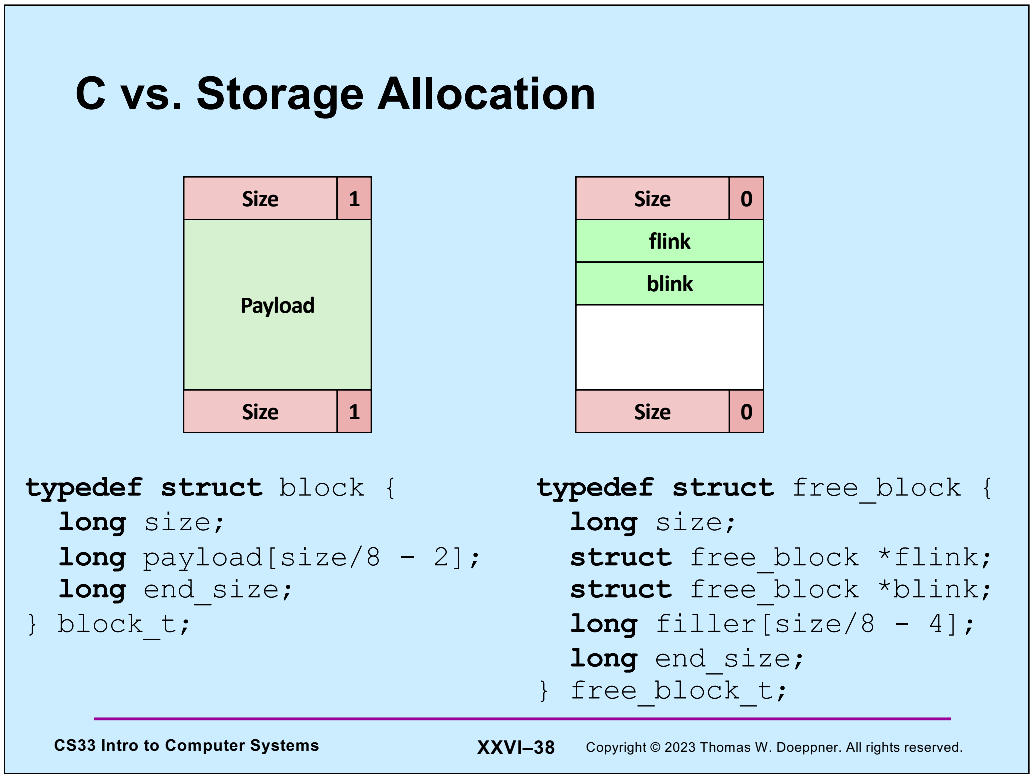
We merge the block we're freeing with the prev and next blocks.



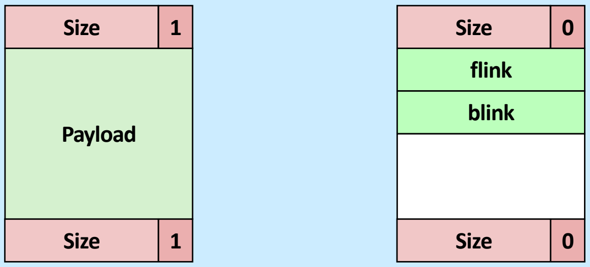
Coalescing: Both Free (4)



Finally we insert the combined block into the beginning of the free list.



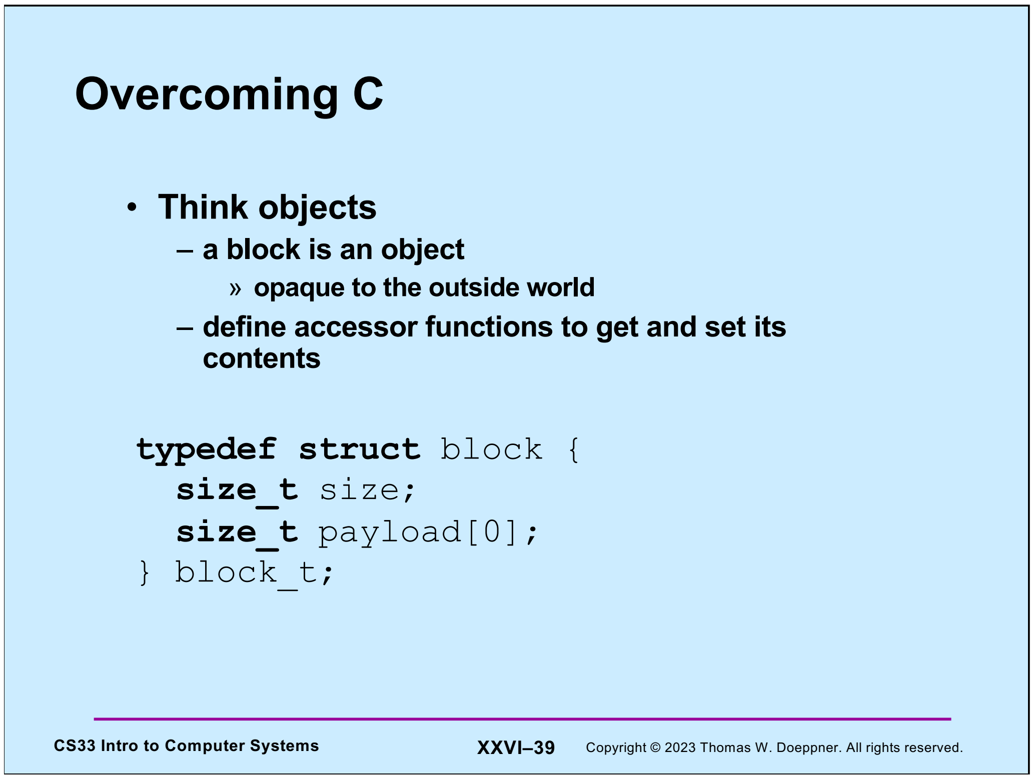
C vs. Storage Allocation



```
typedef struct block {
  long size;
  long payload[size/8 - 2];
  long end_size;
} block_t;
```

```
typedef struct free_block {
  long size;
  struct free_block *flink;
  struct free_block *blink;
  long filler[size/8 - 4];
  long end_size;
} free_block_t;
```

What we might like to be able to do in C is expressed on the slide. Unfortunately, C does not allow such variable-sized arrays. Another concern is the allocated flag, which we'd like to be included in the size fields.

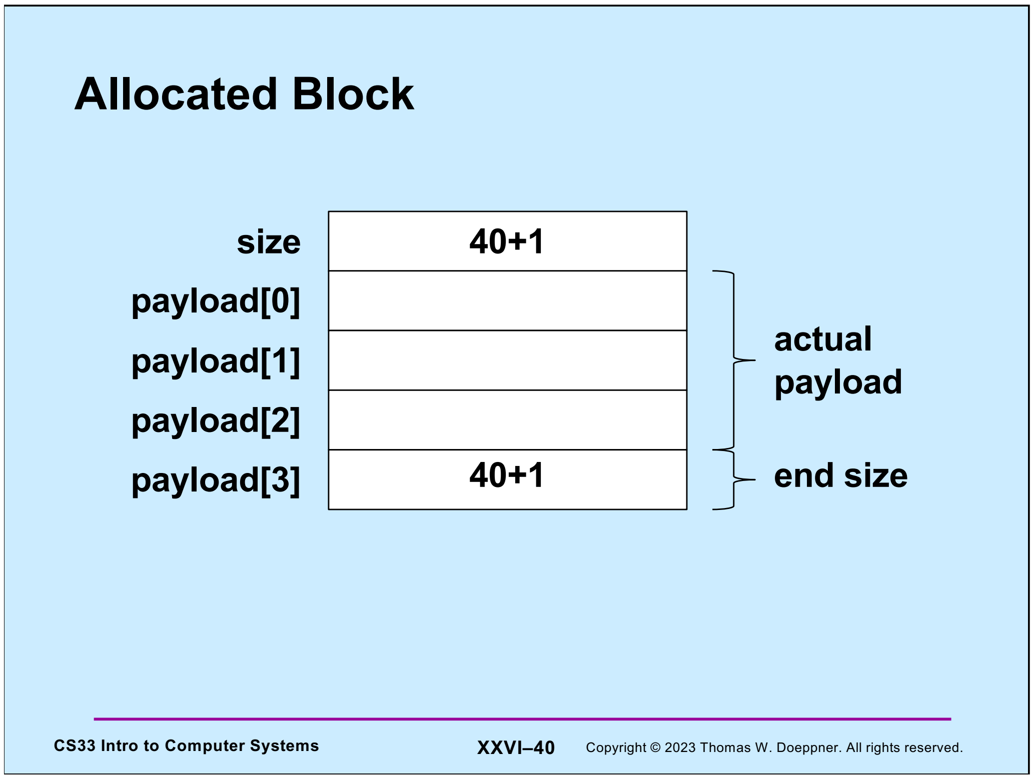


Overcoming C

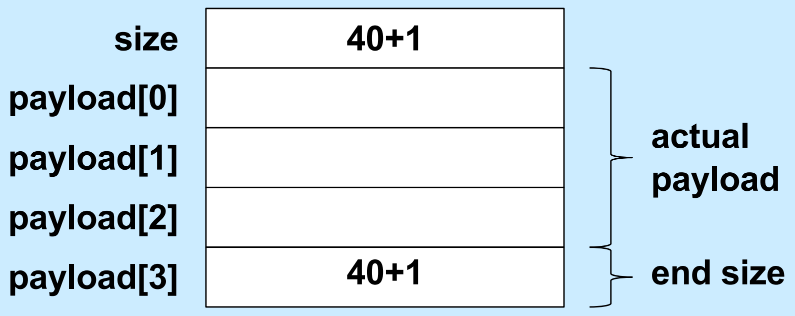
- **Think objects**
  - **a block is an object**
    - » **opaque to the outside world**
  - **define accessor functions to get and set its contents**

```
typedef struct block {
  size_t size;
  size_t payload[0];
} block_t;
```

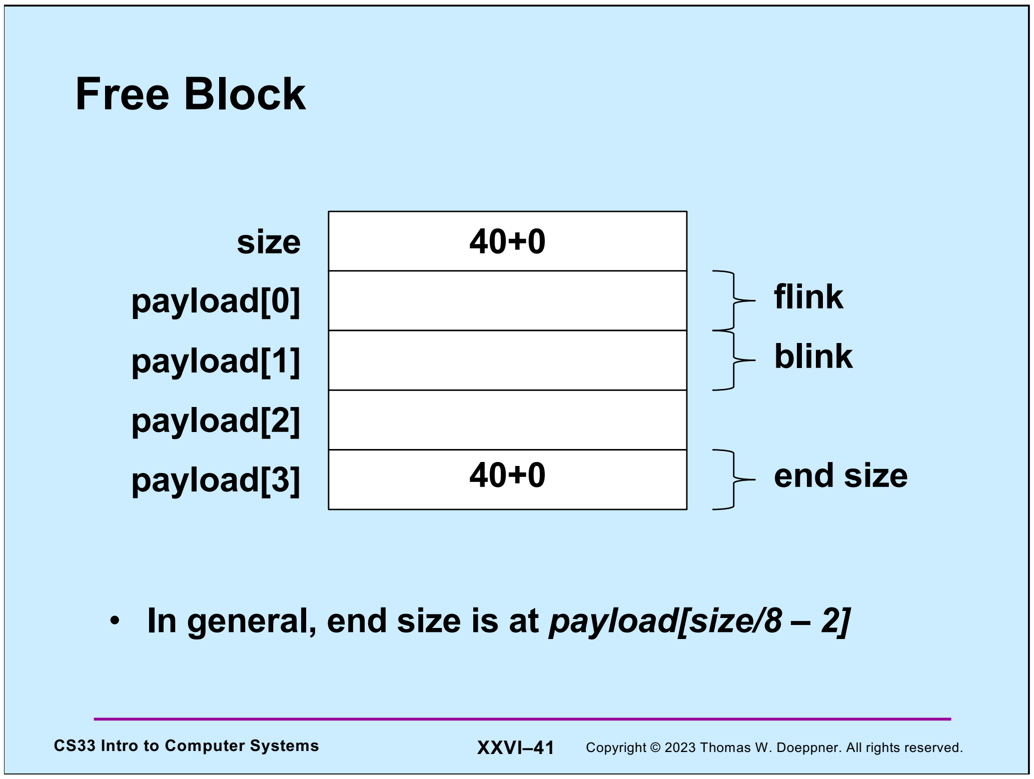
Putting a zero for the dimension of payload is a way of saying that we do not know a priori how big payload will be, so we give it an (arbitrary) size of 0. Note that sizeof(size_t) is 8 (i.e., **size_t** is a typedef for a **long**).



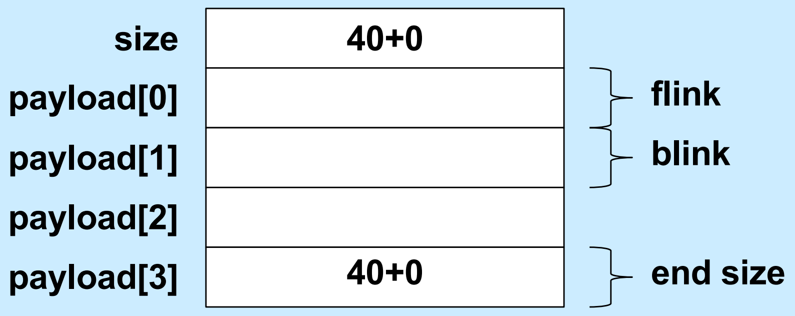
Allocated Block



In this example we have an allocated block of 40 bytes. Its size and end size fields have their low-order bits set to one to indicate that the block is allocated. (Since each element of payload is 8 bytes long, the entire allocated block, including tags, is 40 bytes long.)

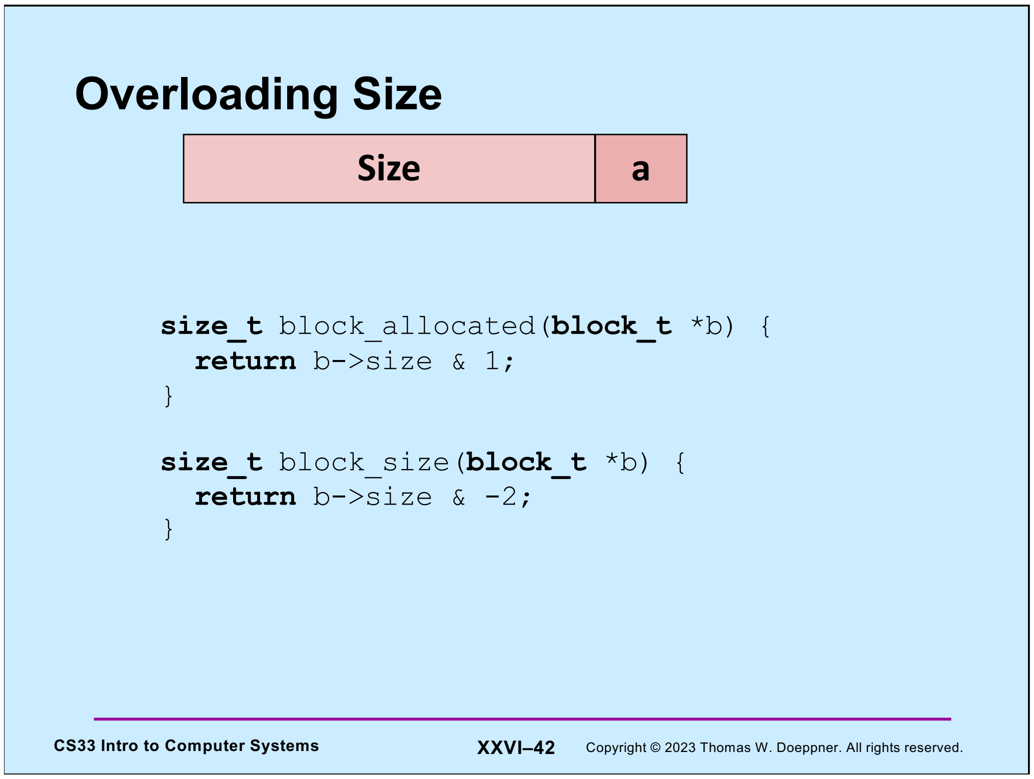


Free Block

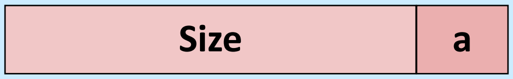


- In general, end size is at *payload[size/8 – 2]*

For a free block, the size fields contain the exact size of the block: the allocated bits are zeroes. The first two elements of payload are the flink and blink pointers, respectively.



Overloading Size

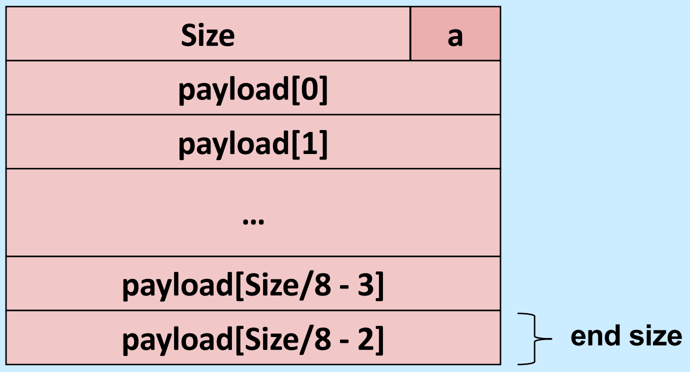


```
size_t block_allocated(block_t *b) {
  return b->size & 1;
}

size_t block_size(block_t *b) {
  return b->size & -2;
}
```

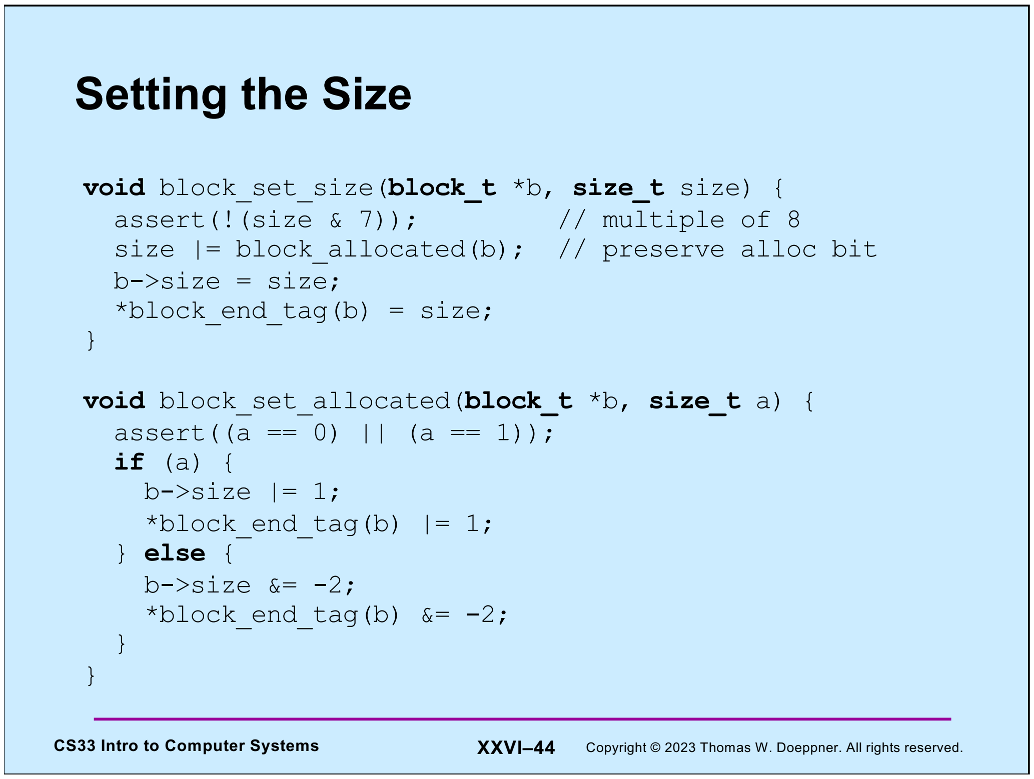
If we assume that the size of a block is always even (in practice it's probably a multiple of 4 or 8), then we can assume the least significant bit is zero and use that bit position to represent the allocated flag.

End Size



```
size_t *block_end_tag(block_t *b) {
  return &b->payload[b->size/8 - 2];
}
```

The block_end_tag function returns the address of a block's end tag, given the address of the beginning of the block (where its front tag is).

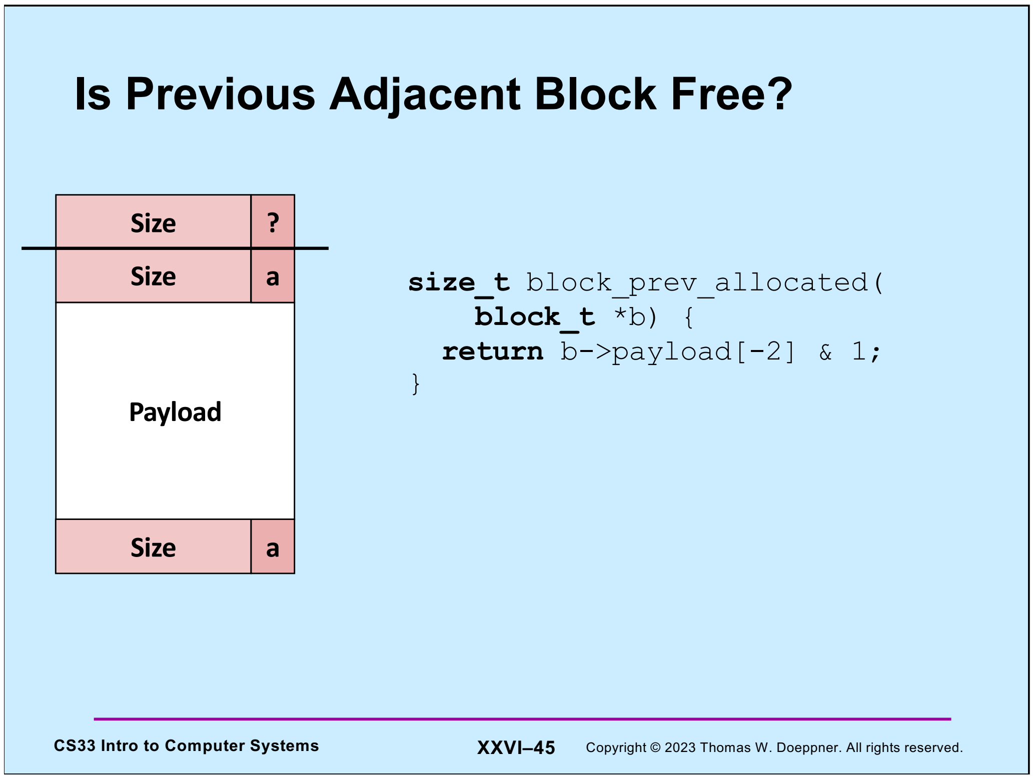


Setting the Size

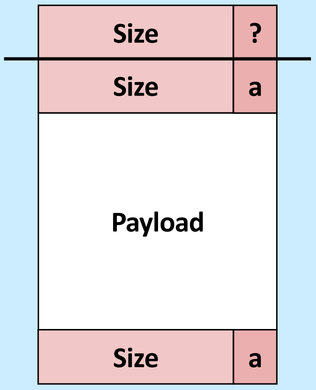
```
void block_set_size(block_t *b, size_t size) {
  assert(!(size & 7));         // multiple of 8
  size |= block_allocated(b);  // preserve alloc bit
  b->size = size;
  *block_end_tag(b) = size;
}

void block_set_allocated(block_t *b, size_t a) {
  assert((a == 0) || (a == 1));
  if (a) {
    b->size |= 1;
    *block_end_tag(b) |= 1;
  } else {
    b->size &= -2;
    *block_end_tag(b) &= -2;
  }
}
```

Here we have functions for setting both tags of a block.

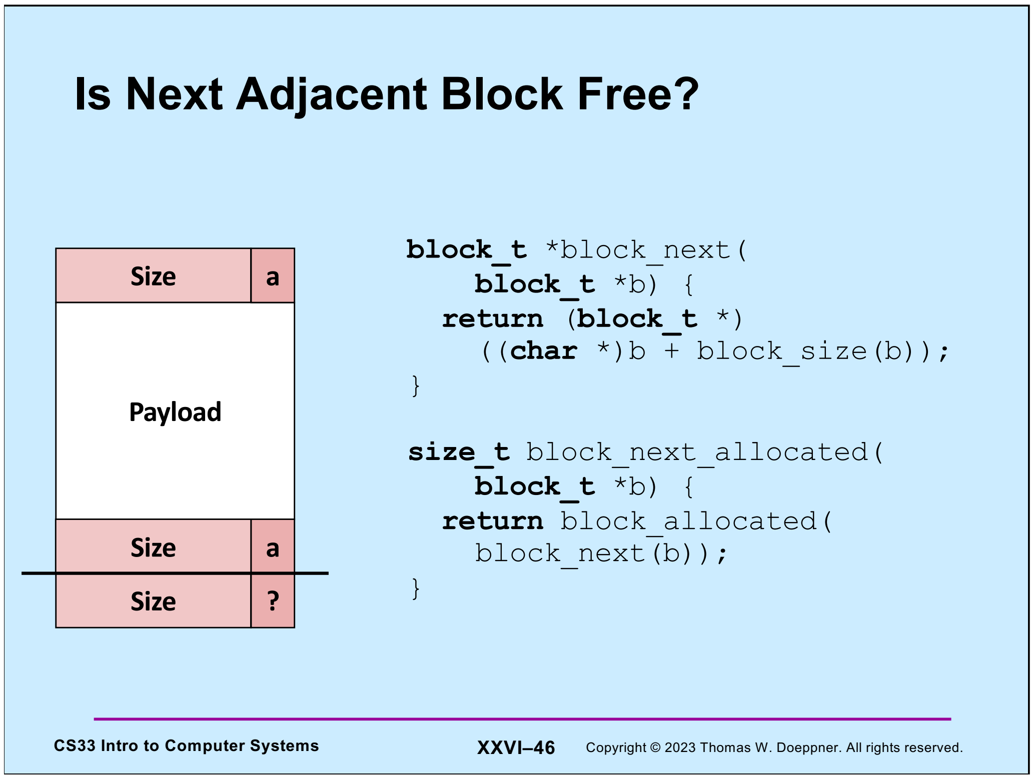


Is Previous Adjacent Block Free?

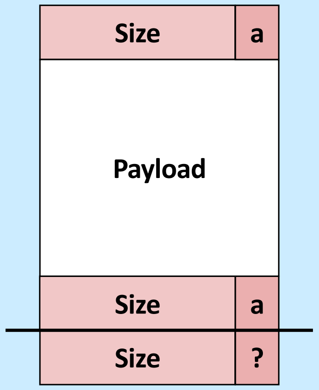


```
size_t block_prev_allocated(
    block_t *b) {
  return b->payload[-2] & 1;
}
```

We take advantage of the boundary-tags approach to determine if the previous block is free.



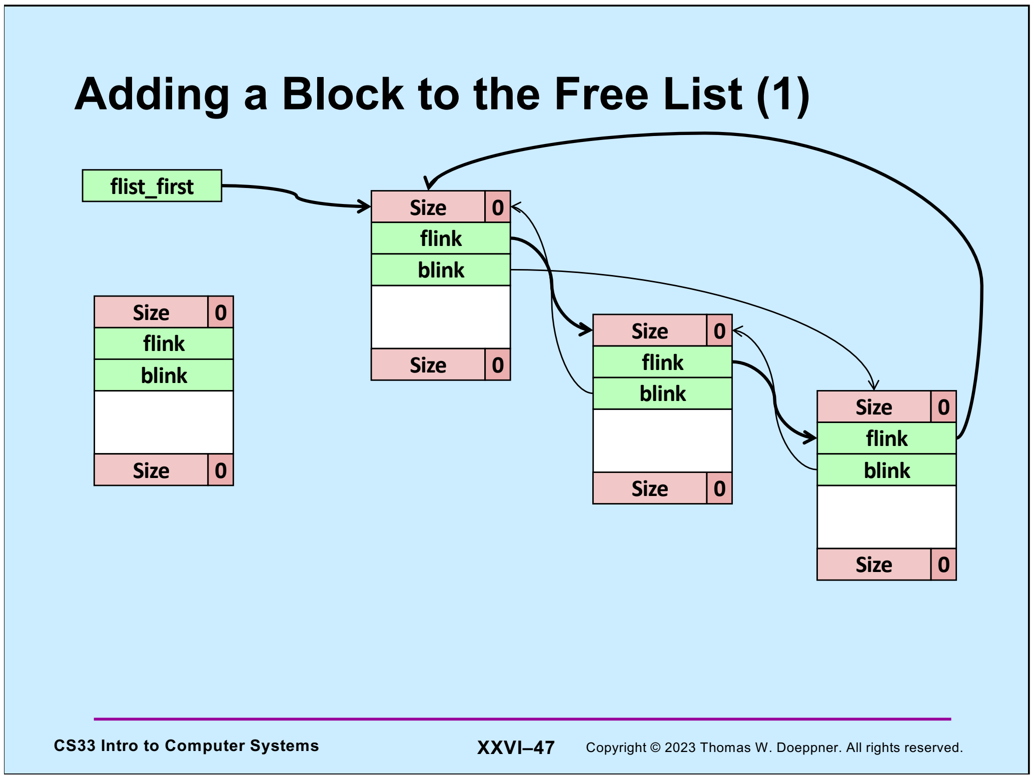
Is Next Adjacent Block Free?



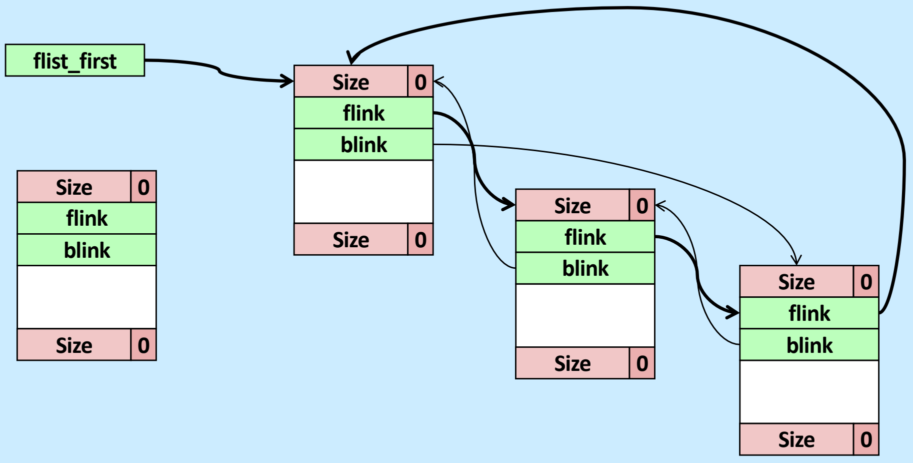
```
block_t *block_next(
    block_t *b) {
  return (block_t *)
    ((char *)b + block_size(b));
}

size_t block_next_allocated(
    block_t *b) {
  return block_allocated(
    block_next(b));
}
```

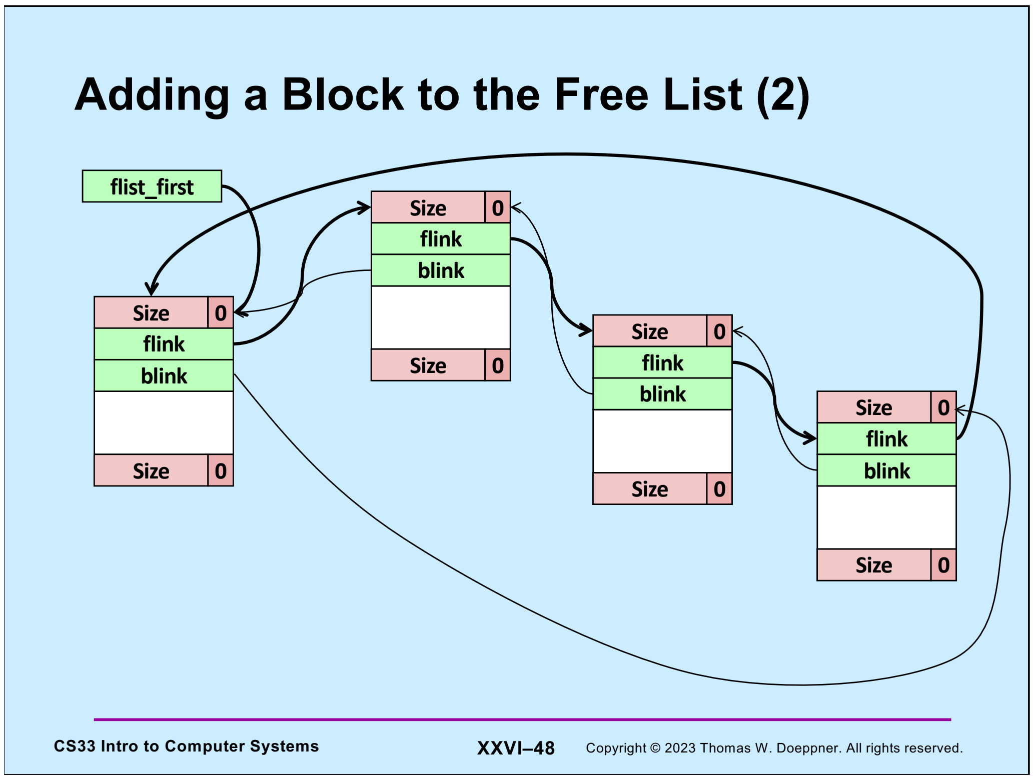
Similarly, we can determine if the next block is free.



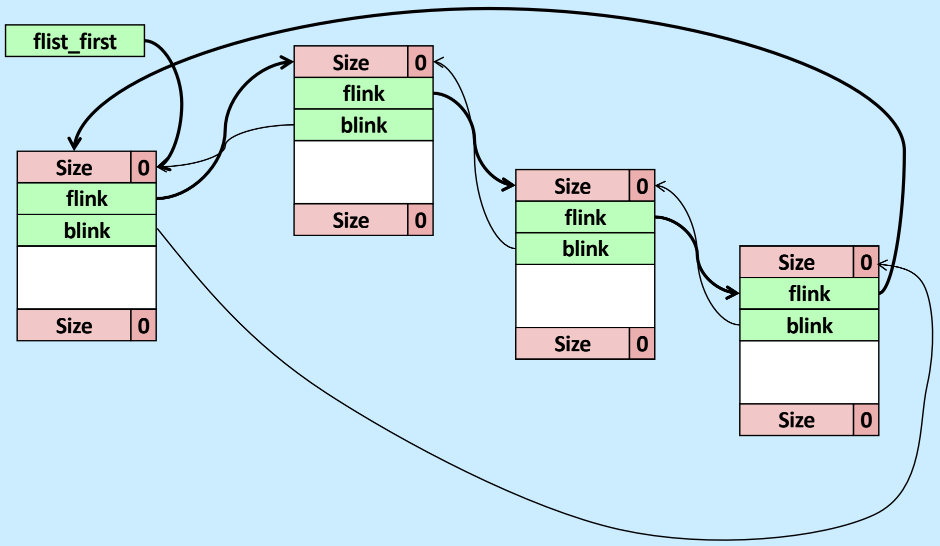
Adding a Block to the Free List (1)



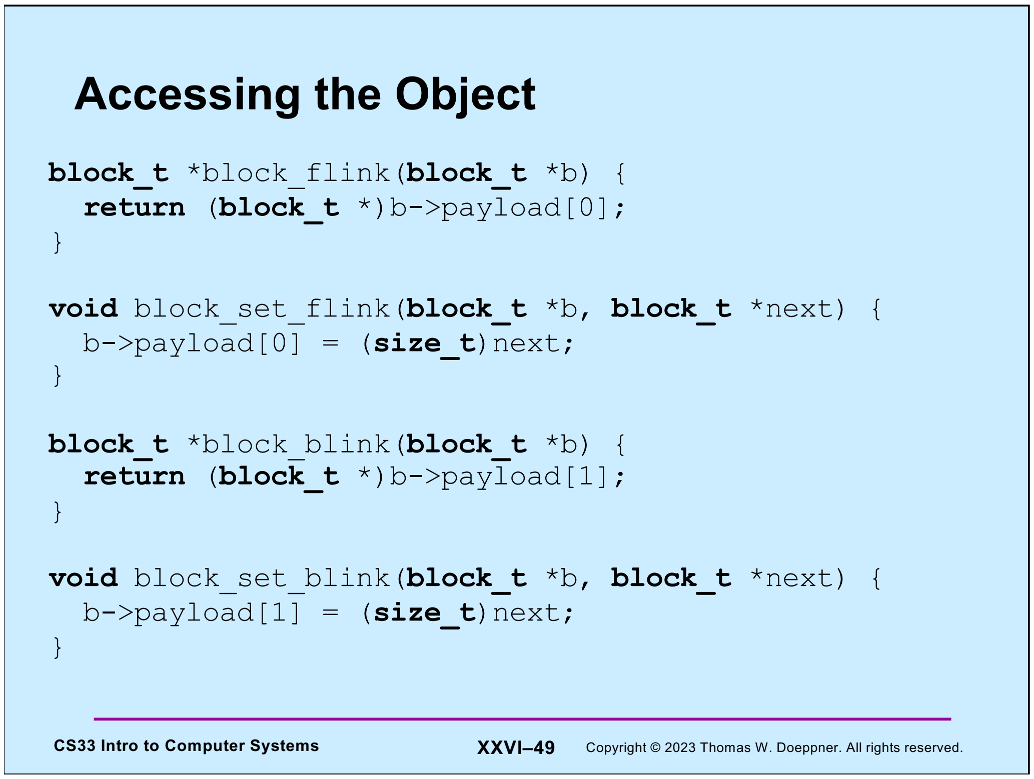
An important operation is to add a block to the beginning of the free list. We start with a picture of the free list.



Adding a Block to the Free List (2)



Here's what it looks like after we add the block.



Accessing the Object

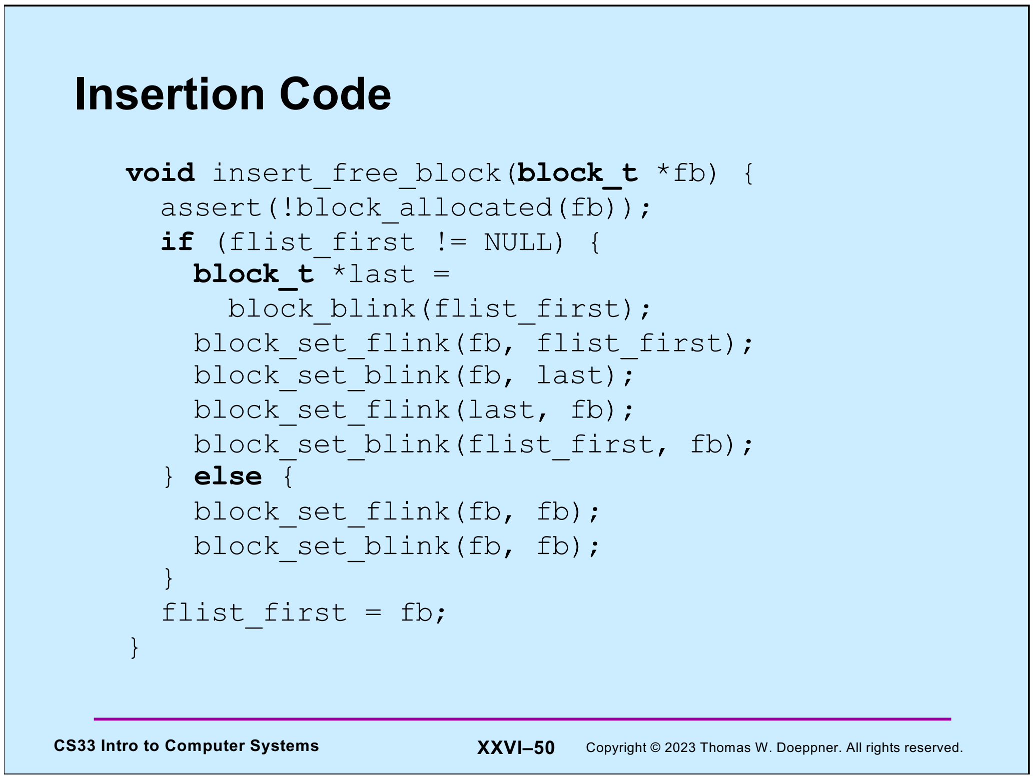
```
block_t *block_flink(block_t *b) {
  return (block_t *)b->payload[0];
}

void block_set_flink(block_t *b, block_t *next) {
  b->payload[0] = (size_t)next;
}

block_t *block_blink(block_t *b) {
  return (block_t *)b->payload[1];
}

void block_set_blink(block_t *b, block_t *next) {
  b->payload[1] = (size_t)next;
}
```

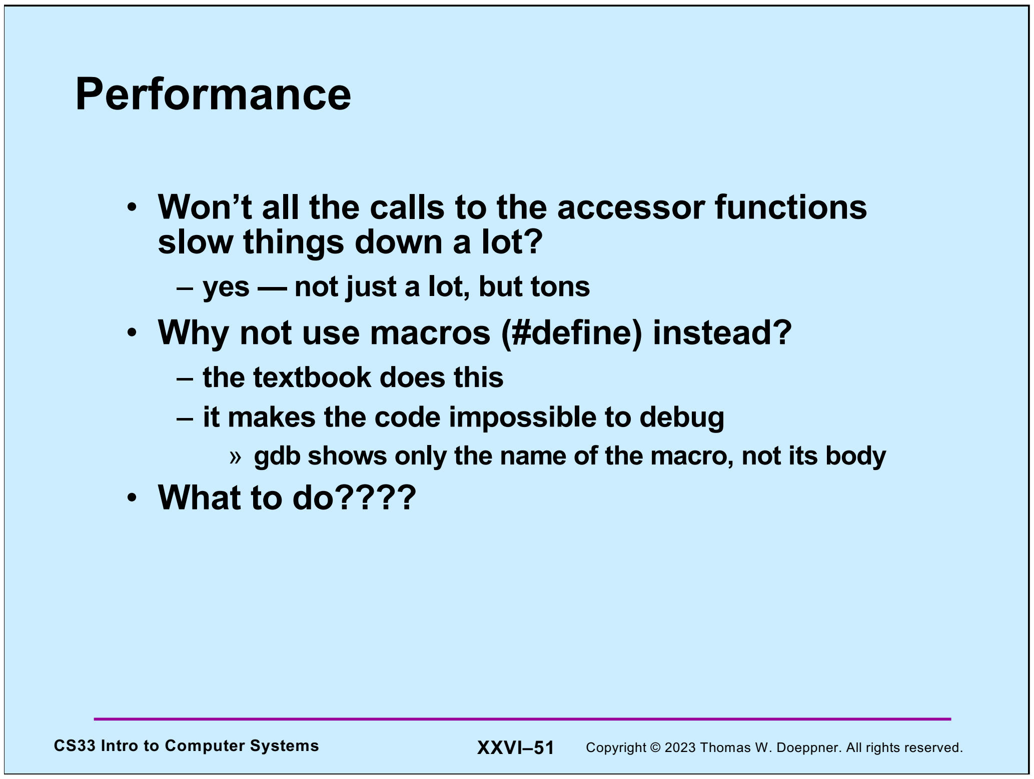
Here are a few more simple functions we need to access and set fields of blocks.



Insertion Code

```
void insert_free_block(block_t *fb) {
  assert(!block_allocated(fb));
  if (flist_first != NULL) {
    block_t *last =
      block_blink(flist_first);
    block_set_flink(fb, flist_first);
    block_set_blink(fb, last);
    block_set_flink(last, fb);
    block_set_blink(flist_first, fb);
  } else {
    block_set_flink(fb, fb);
    block_set_blink(fb, fb);
  }
  flist_first = fb;
}
```

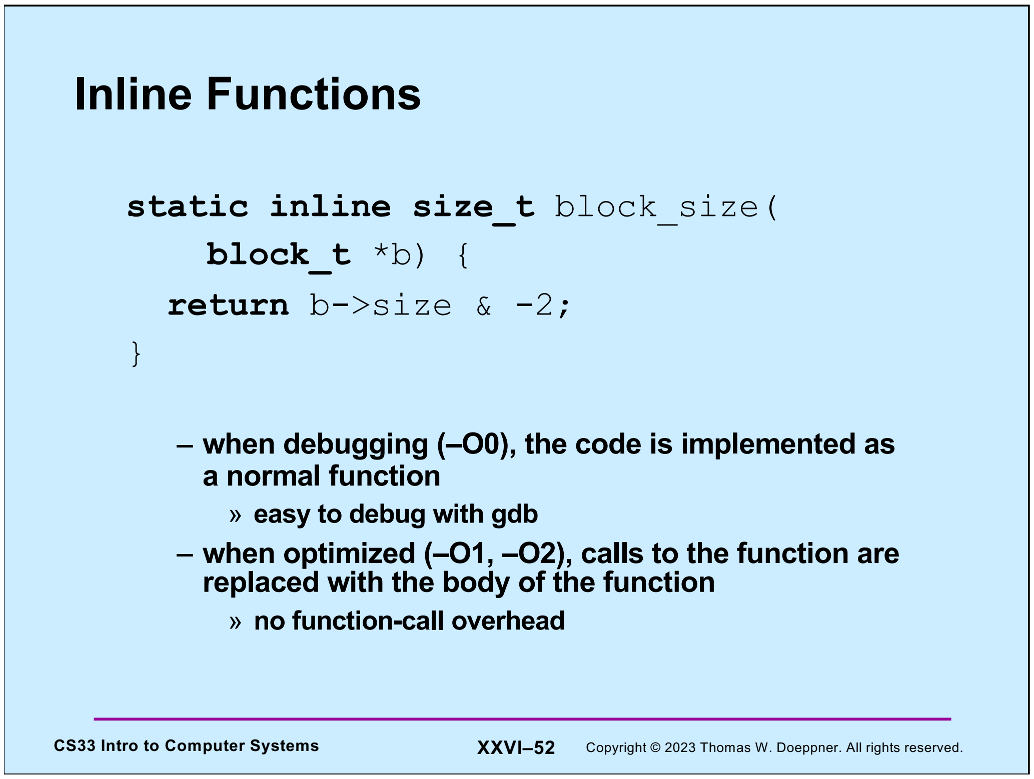
Using our functions, here's the code to insert a block at the beginning of the free list.



Performance

- Won't all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - » gdb shows only the name of the macro, not its body
- What to do????

We've used a lot of functions without thinking about their effect on performance. While we know that the overhead of a function call is not great, there is still some overhead that might best be eliminated.



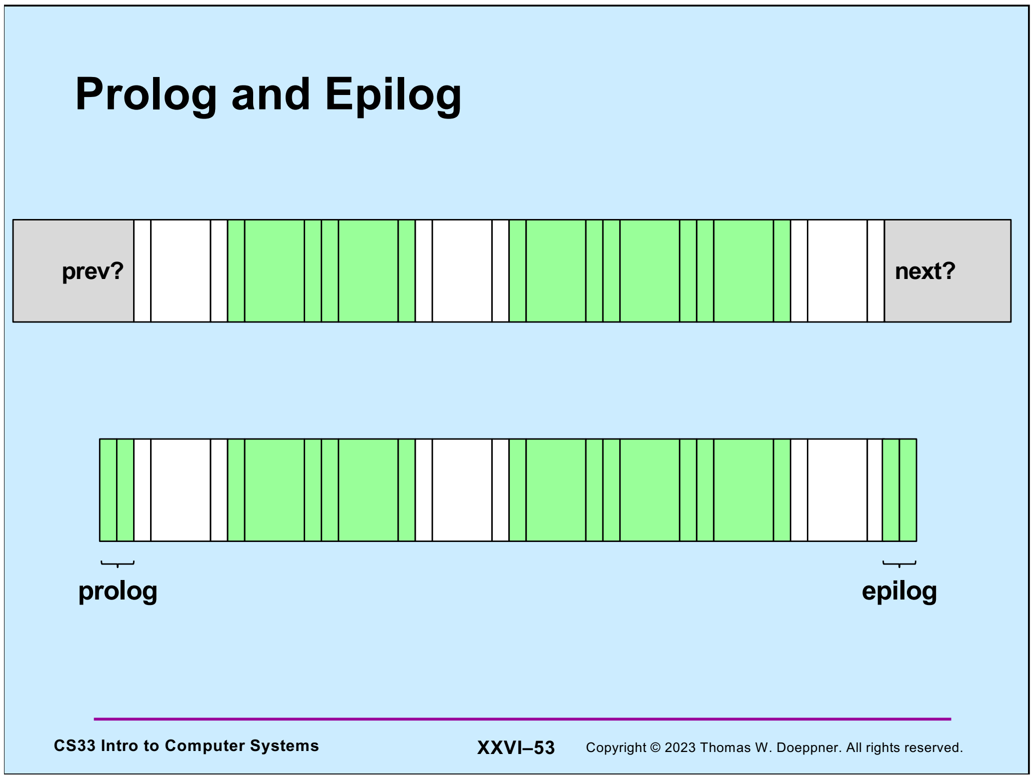
Inline Functions

```
static inline size_t block_size(
    block_t *b) {
  return b->size & -2;
}
```

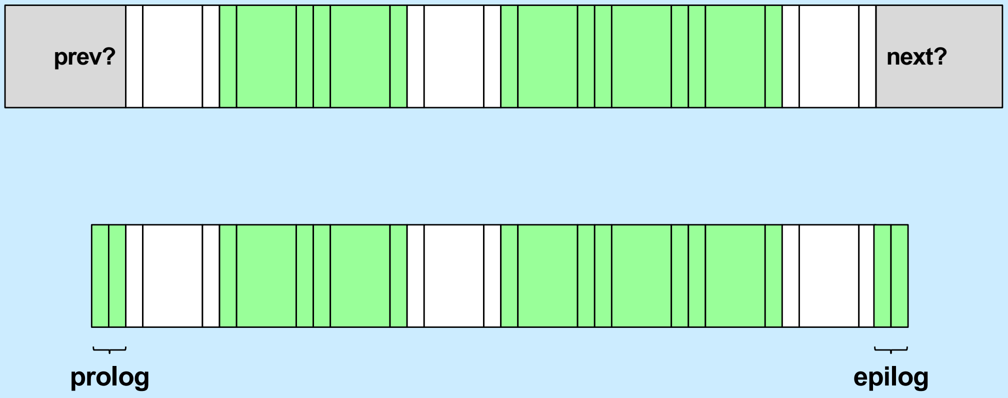
- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead

If we declare a function to be **inline**, the C compiler is instructed to replace calls to the function with its actual code (unless –O0 is specified). Thus, inlined functions have no function-call overhead (though they do increase the total size of our code, which does come at some cost).

Note that inline functions are declared to be *static*. This makes it possible to have two .c files that use an inline function, with one compiled with –O0 and the other perhaps with –O1 – since the function is static, it can be different in the two source files.



Prolog and Epilog



The slide shows our heap, with allocated blocks shown in green and free blocks in white. At either end of each block are its tags.

An issue that comes up when implementing malloc/free is dealing with the first and last blocks, whether they are allocated or free. What is the **prev** block relative to the first block? What is the **next** block relative to the last block? Having to special-case the first and last blocks can help make your code unnecessarily complicated. To avoid these complications, we use **prolog** and **epilog** blocks. These are blocks of minimum size (containing just two tags and no payload) that are marked **allocated**. They are on either end of the list. Since they're marked allocated, when a check is made of the **prev** block relative to the first real block, it will always appear to be allocated, and similarly with the **next** block relative to the last real block.

Thus, the initial heap might consist of three blocks: the prolog, a block representing the initial free space, and an epilog. Of course, when the heap is expanded by calling **sbrk**, the epilog must be moved to the new end of the heap.