CS107, Lecture 16
More Managing The Heap; Optimization

Reading: B&O 9.9, 9.11

Learning Goals

- Learn about the unique aspects of the explicit free list allocator design
- Understand the process of coalescing free blocks and how it helps performance
- Understand the process of in-place realloc and how it helps performance

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- **Break:** Announcements
- In-Place Realloc
- Optimization

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- **Break:** Announcements
- In-Place Realloc
- Optimization

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- A heap allocator is like a hotel concierge that manages its hotel rooms!

Heap Allocator Functions

```
void *malloc(size_t size);
```

Returns a pointer to a block of heap memory of at least size bytes, or NULL if an error occurred.

```
void free(void *ptr);
```

Frees the heap-allocated block starting at the specified address.

```
void *realloc(void *ptr, size_t size);
```

Changes the size of the heap-allocated block starting at the specified address to be the new specified size. Returns the address of the new, larger allocated memory region.

Heap Allocator Requirements

A heap allocator must…

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay
5. Return addresses that are 8-byte-aligned (must be multiples of 8).

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

These are seemingly conflicting goals – for instance, it may take longer to better plan out heap memory use for each request.

Heap allocators must find an appropriate balance between these two goals!

Bump Allocator

- A **bump allocator** is a heap allocator design that simply allocates the next available memory address upon an allocate request, and does nothing on a free request.
- Throughput: each **malloc** and **free** execute only a handful of instructions:
  - It is easy to find the next location to use
  - Free does nothing!
- Utilization: we use each memory block at most once. No freeing at all, so no memory is ever reused. ☹
- We provide a bump allocator implementation as part of assign7 as a code reading exercise.

Bump Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(24);
free(b);
void *d = malloc(4);
```

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

AVAILABLE

Bump Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(24);
free(b);
void *d = malloc(4);
```

Variable	Value
a	0x10

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34
a + padding		AVAILABLE							

Bump Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(24);
free(b);
void *d = malloc(4);
```

Variable	Value
a	0x10
b	0x18

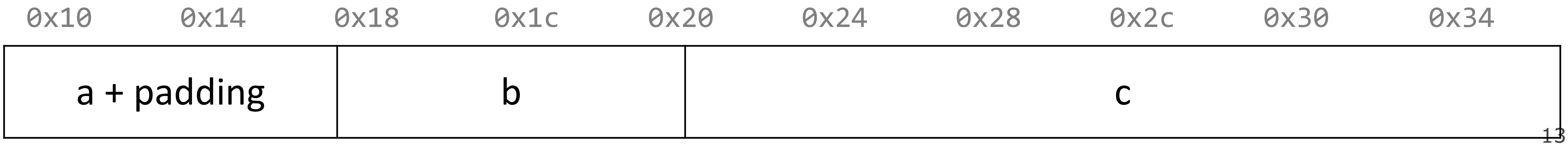
0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34

a + padding	b	AVAILABLE

Bump Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(24);
free(b);
void *d = malloc(4);
```

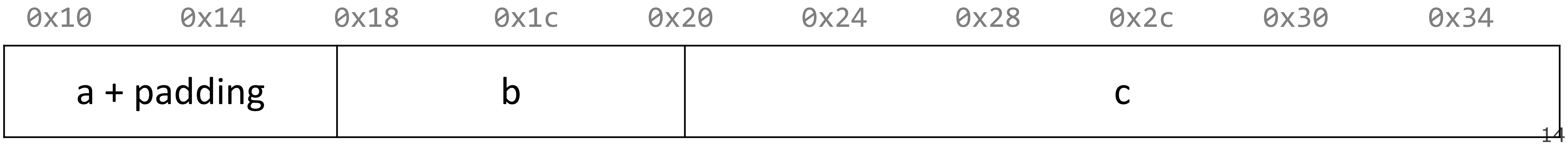
Variable	Value
a	0x10
b	0x18
c	0x20



Bump Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(24);
free(b);
void *d = malloc(4);
```

Variable	Value
a	0x10
b	0x18
c	0x20



Bump Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(24);
free(b);
void *d = malloc(4);
```

Variable	Value
a	0x10
b	0x18
c	0x20
d	NULL

0x10 0x14 0x18 0x1c 0x20 0x24 0x28 0x2c 0x30 0x34

a + padding	b	c

Summary: Bump Allocator

- A bump allocator is an extreme heap allocator – it optimizes only for **throughput**, not **utilization**.
- Better allocators strike a more reasonable balance. How can we do this?

Questions to consider:

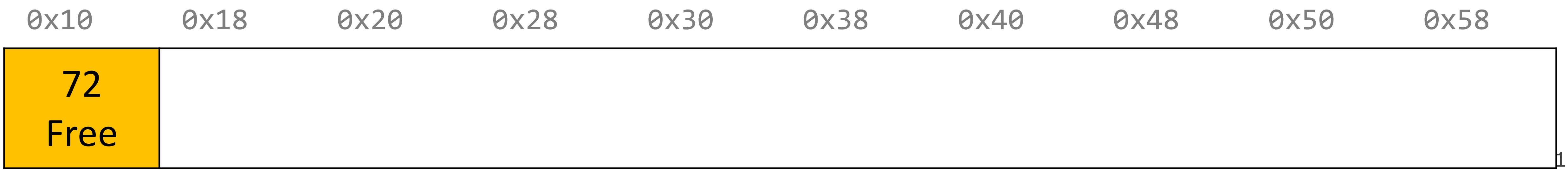
1. How do we keep track of free blocks?
2. How do we choose an appropriate free block in which to place a newly allocated block?
3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
4. What do we do with a block that has just been freed?

Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated or free.
- When we allocate a block, we look through the blocks to find a free one, and we update its header to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free.
- The header should be 8 bytes (or larger).

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```



Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	56 Free							

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

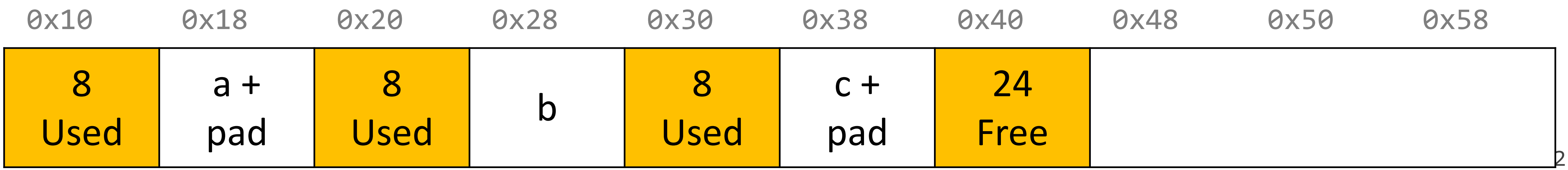
Variable	Value
a	0x18
b	0x28

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Used	b	40 Free					

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38



Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Free	b	8 Used	c + pad	24 Free			

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	8 Used	d	8 Used	c + pad	24 Free			

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Free			

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28
e	0x48

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Used	e		

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28
e	0x48

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	d	8 Used	c + pad	24 Used	e		

Representing Headers

How can we store both a size and a status (Free/Allocated) in 8 bytes?

Int for size, int for bytes? **no! malloc/realloc use size_t for sizes!**

Key idea: block sizes will *always be multiples of 8*. (Why?)

- Least-significant 3 bits will be unused!
- *Solution:* use one of the 3 least-significant bits to store free/allocated status

For assignment 7, you may use this approach, or another approach, but remember that header sizes affect utilization!

Implicit Free List Allocator

- How can we choose a free block to use for an allocation request?
  - **First fit:** search the list from beginning each time and choose first free block that fits.
  - **Next fit:** instead of starting at the beginning, continue where previous search left off.
  - **Best fit:** examine every free block and choose the one with the smallest size that fits.
- Notice if possible we use only a chunk that we need of a larger free block.
- What are the pros/cons of this approach?
  - Con: Headers use extra memory in each block
  - Pro: Can reuse blocks
  - Con: must search entire heap for free blocks

Revisiting Our Goals

Questions we considered:

1. How do we keep track of free blocks? **Using headers!**
2. How do we choose an appropriate free block in which to place a newly allocated block? **Iterate through all blocks.**
3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block? **Try to make the most of it!**
4. What do we do with a block that has just been freed? **Update its header!**

Assignment 7: Implicit Allocator

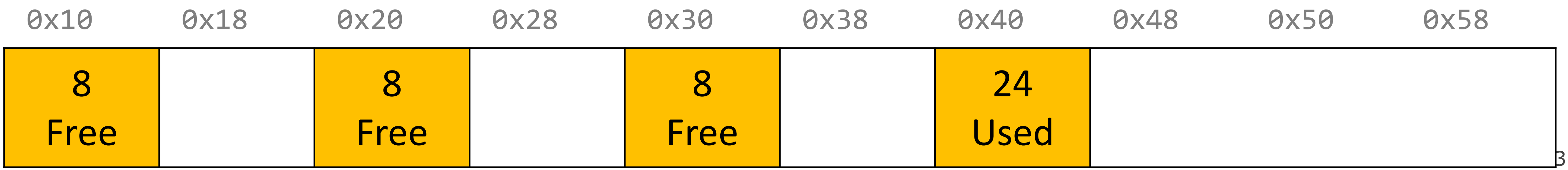
- **Must have** headers that track block information (size, status in-use or free) - we recommend using headers larger than the 4 byte headers specified in the book, as this makes it easier to satisfy the alignment constraint and store information.
- **Must have** free blocks that are recycled and reused for subsequent malloc requests if possible
- **Must have** a malloc implementation that searches the heap for free blocks via an implicit list (i.e. traverses block-by-block).

- **Does not need to** have coalescing of free blocks
- **Does not need to** support in-place realloc

Coalescing

```
void *e = malloc(24);   // returns NULL!
```

You do not need to worry about this problem for the implicit allocator, but this is a requirement for the *explicit* allocator! (More about this later).



In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

In-Place Realloc

```c
void *a = malloc(4);
void *b = realloc(a, 8);
```

Variable	Value
a	0x18

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	a + pad	56 Free							

In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

Variable	Value
a	0x10
b	0x28

The implicit allocator can always move memory to a new location for a realloc request. The *explicit* allocator must support in-place realloc (more on this later).

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free	a + pad	8 Used	b	40 Free					

Summary: Implicit Allocator

- An implicit allocator is a more efficient implementation that has reasonable **throughput** and **utilization** due to its recycling of blocks.

Can we do better?

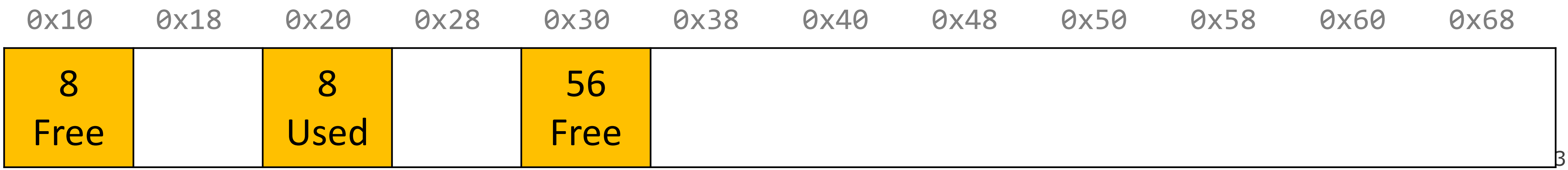
1. Can we avoid searching all blocks for free blocks to reuse?
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during realloc?

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- **Break:** Announcements
- In-Place Realloc
- Optimization

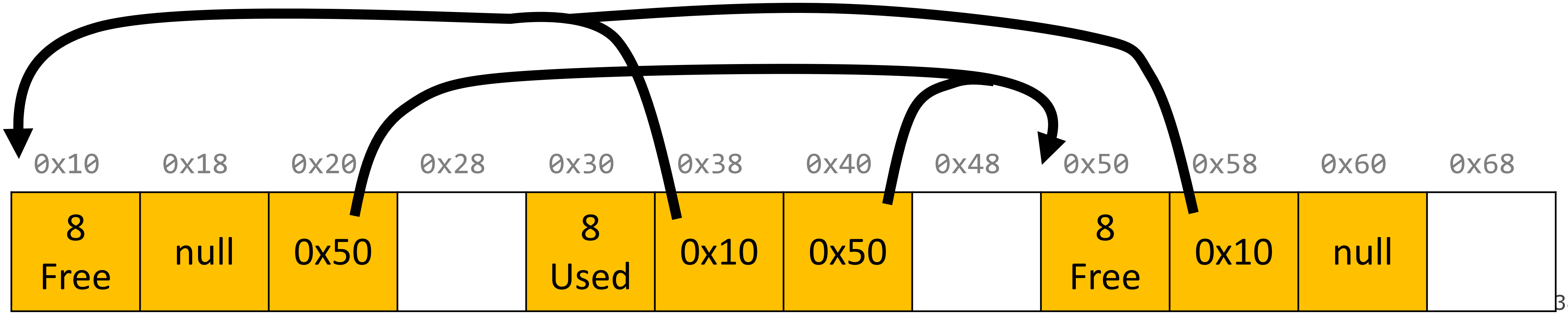
Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the next free block and a pointer to the previous free block.



Can We Do Better?

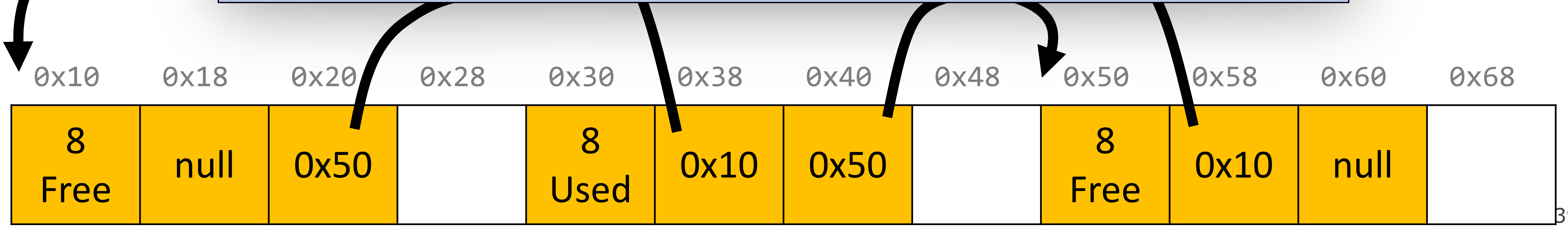
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the next free block and a pointer to the previous free block.



Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the next free block and a pointer to the previous free block.

This is inefficient – it triples the size of *every* header, when we really just need to jump from one free block to another. And even if we just made free headers bigger, it's complicated to have *two* different header sizes.



Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the next free block and a pointer to the previous free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**

Can We Do Better?

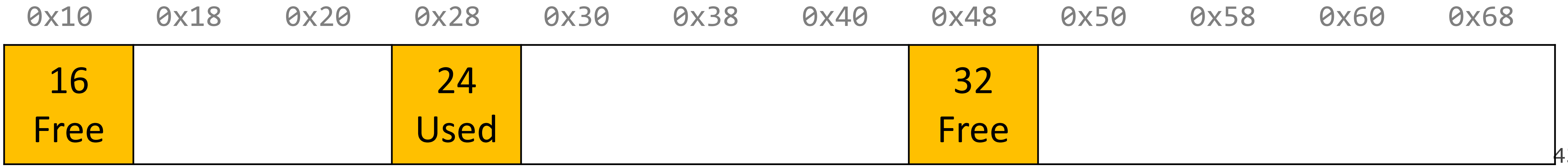
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the next free block and a pointer to the previous free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure?

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the next free block and a pointer to the previous free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure? *More difficult to access in a separate place – prefer storing near blocks on the heap itself.*

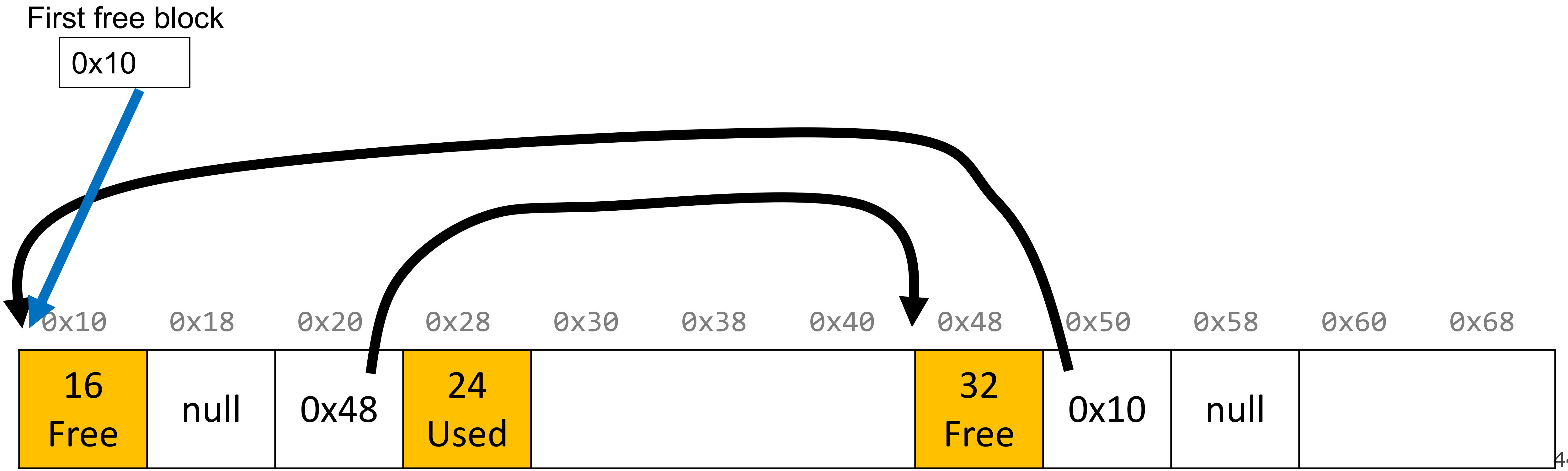
Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>



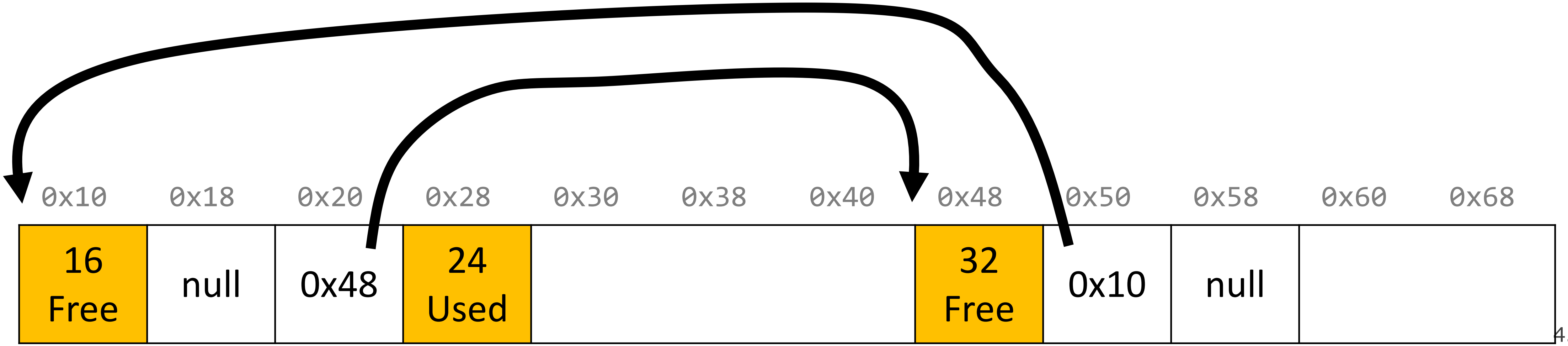
Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the first 16 bytes of each free block's payload!



Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the first 16 bytes of each free block's payload!
- This means each payload must be big enough to store 2 pointers (16 bytes). So we must require that for every block, free and allocated. (why?)

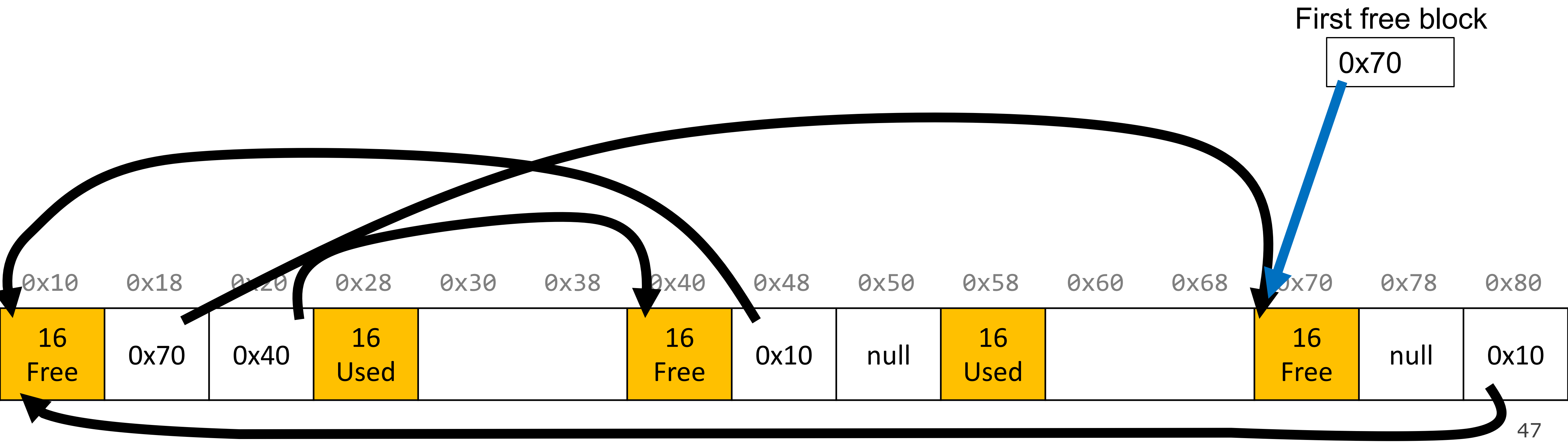


Explicit Free List Allocator

- This design builds on the implicit allocator, but also stores pointers to the next and previous free block inside each free block's payload.
- When we allocate a block, we look through just the free blocks using our linked list to find a free one, and we update its header and the linked list to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free, and update the linked list.

Explicit Free List Allocator

- Note that the doubly-linked list *does not have to be in address order*.
- You should build up your linked list as efficiently as possible (e.g. where is it most efficient to add to a linked list?)
- When you allocate a block, you must update your linked list pointers.



Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during realloc?

Revisiting Our Goals

Can we do better?

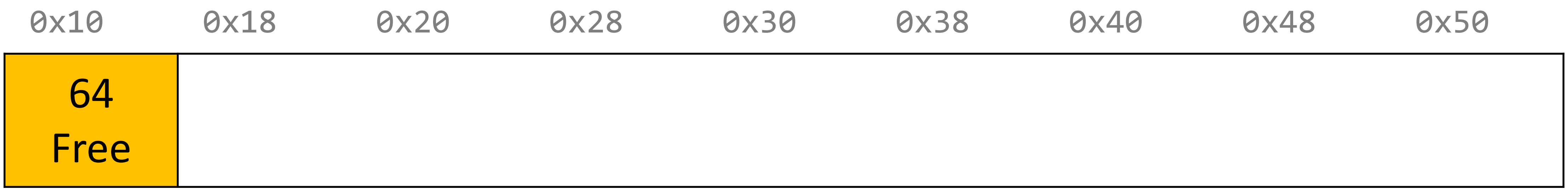
1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. **Can we merge adjacent free blocks to keep large spaces available?**
3. Can we avoid always copying/moving data during realloc?

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- **Break:** Announcements
- In-Place Realloc
- Optimization

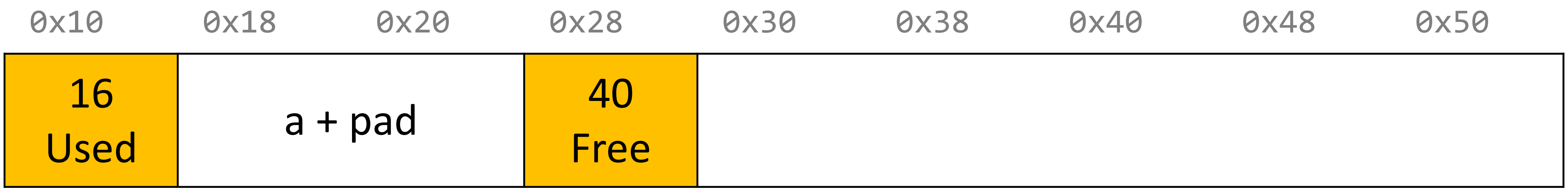
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



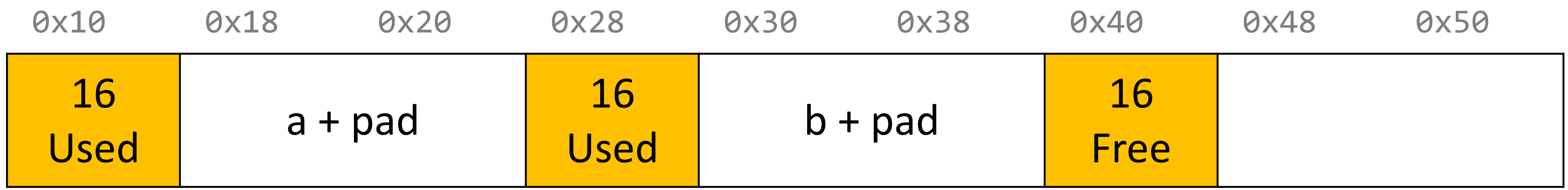
Coalescing

```c
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



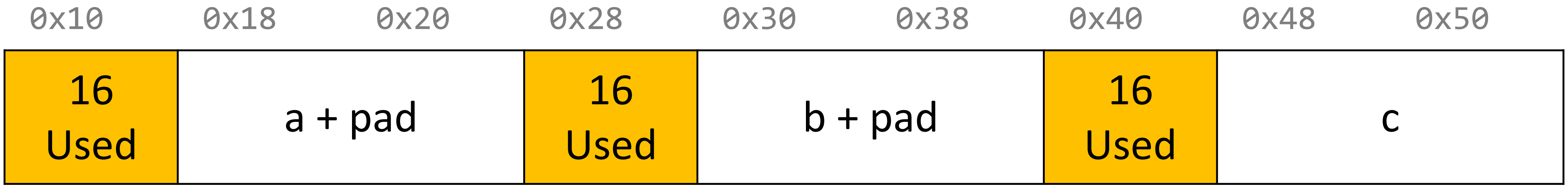
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



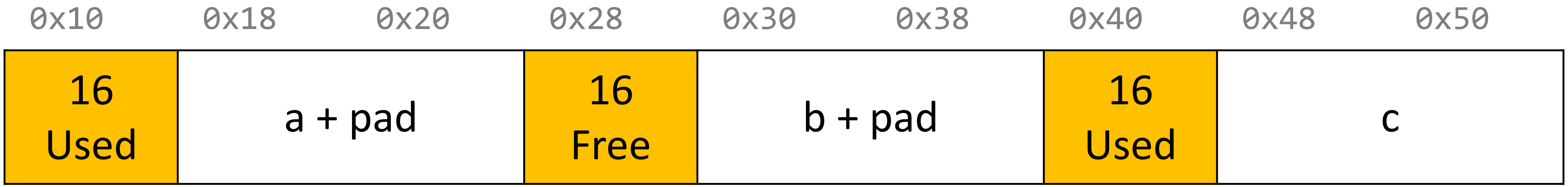
Coalescing

```c
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



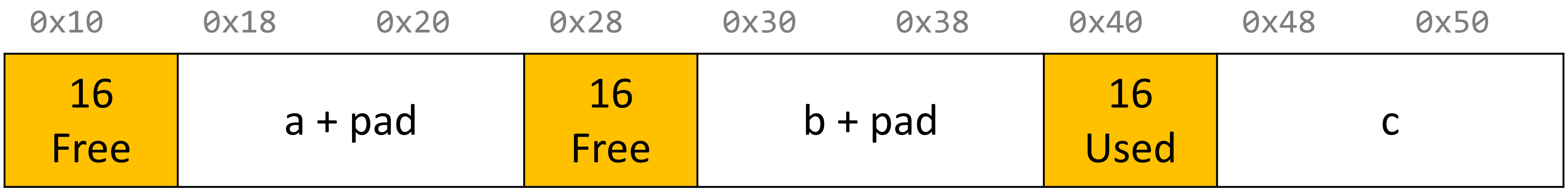
Coalescing

```c
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

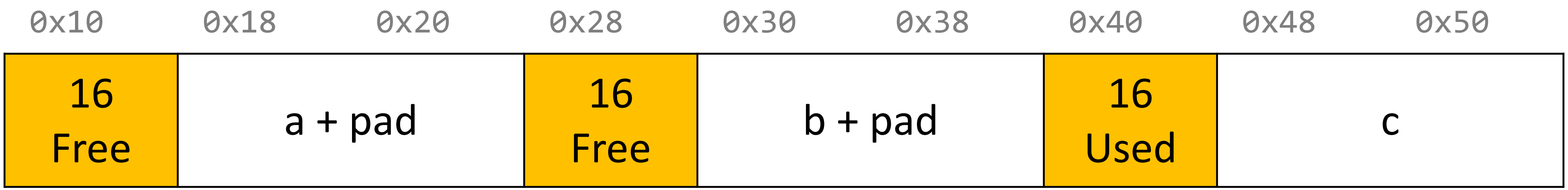


Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

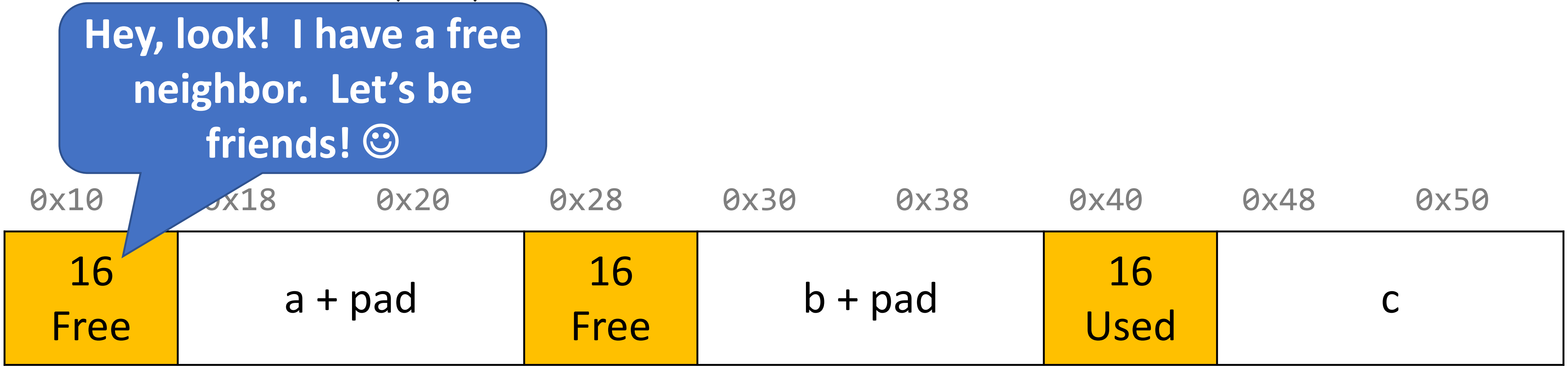
We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

We'd like to be able to merge adjacent free blocks back together. How can we do this?



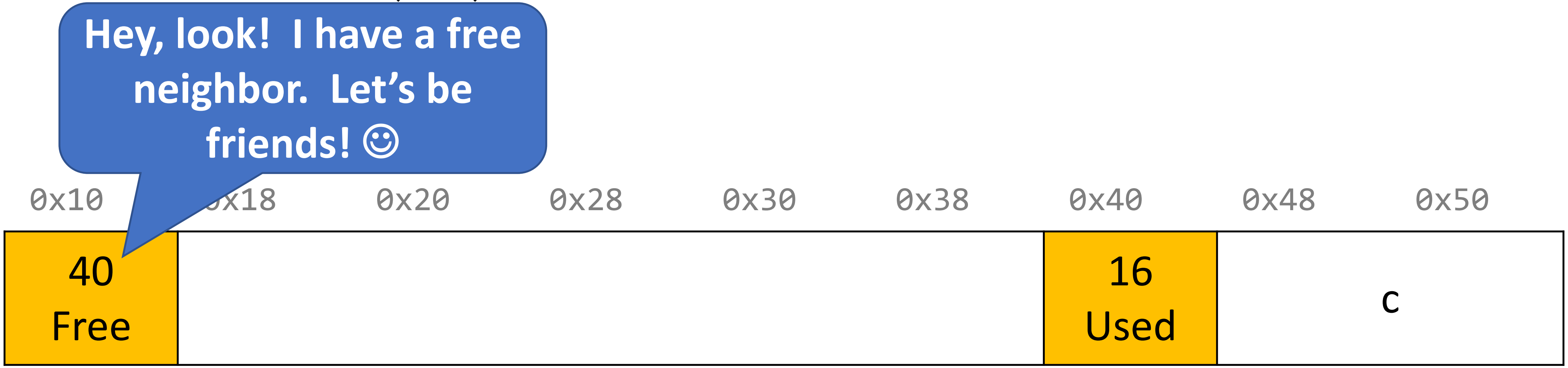
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

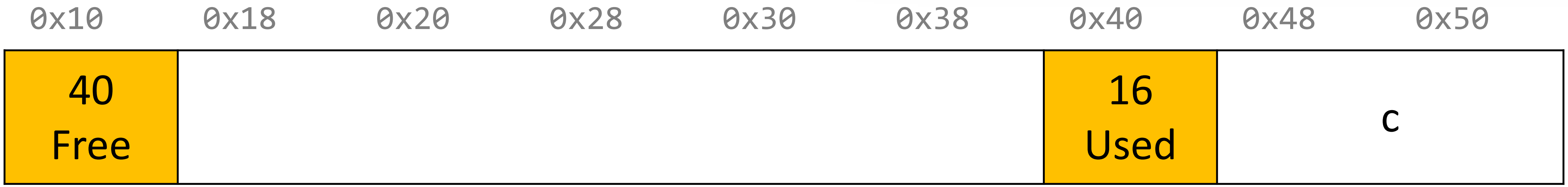


Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

The process of combining adjacent free blocks is called *coalescing*.

For your explicit heap allocator, you should coalesce if possible when a block is freed. **You only need to coalesce the most immediate right neighbor.**



Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on free().**
3. Can we avoid always copying/moving data during realloc?

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on free().**
3. **Can we avoid always copying/moving data during realloc?**

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- Break: Announcements
- In-Place Realloc
- Optimization

Announcements

Clarifications on assign7 (recent additions to FAQ)

What can/can't I assume about the validity of the parameters to myinit?

You may assume that the heap starting address is aligned to the ALIGNMENT constant. You should not assume anything about the heap size, such as that it is a multiple of ALIGNMENT (though it is actually fine if it is not), or that it is large enough for the heap allocator to use.

Does our code have to work for any alignment, or just 8?

Your code should work for 8 and any value that is a factor of 8, but does not have to work for other alignment values, such as multiples of 8. That being said, you should still use the ALIGNMENT constant rather than hardcoding the value where possible.

Announcements

Microsoft recently open-sourced Windows Calculator!

- Great case study of tradeoffs of accurate numeric computation (floats!)
- Can browse the code on GitHub (online code sharing website) – code from as far back as 1995!

From Ars Technica article:

“The actual calculations are performed by this ancient code. Calculator's mathematics library is built using rational numbers (that is, numbers that can be expressed as the ratio of two integers). Where possible, it preserves the exact values of the numbers it is computing, falling back on Taylor series expansion when an approximation to an irrational number is required. Poking around the change history shows that the very earliest iterations of Windows Calculator, starting in 1989, didn't use the rational arithmetic library, instead using floating point arithmetic and the much greater loss of precision this implies.”

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on free().**
3. **Can we avoid always copying/moving data during realloc?**

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- Break: Announcements
- **In-Place Realloc**
- Optimization

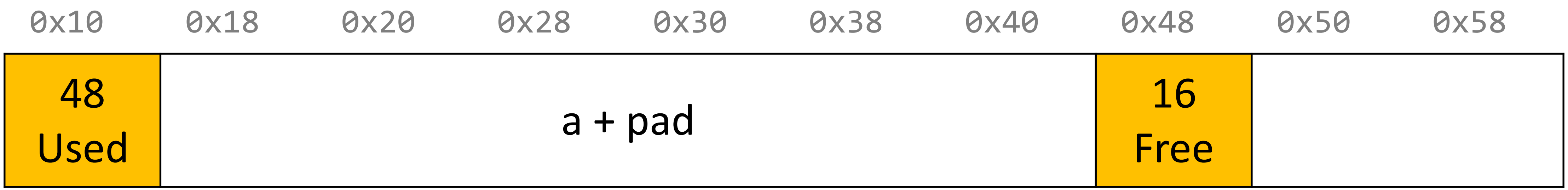
Realloc

- For the implicit allocator, we didn't worry too much about realloc. We always moved data when they requested a different amount of space.
  - Note: realloc can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.

Realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 48);
```

a’s earlier request was too small, so we added padding. Now they are requesting a larger size we can satisfy with that padding! So realloc can return the same address.

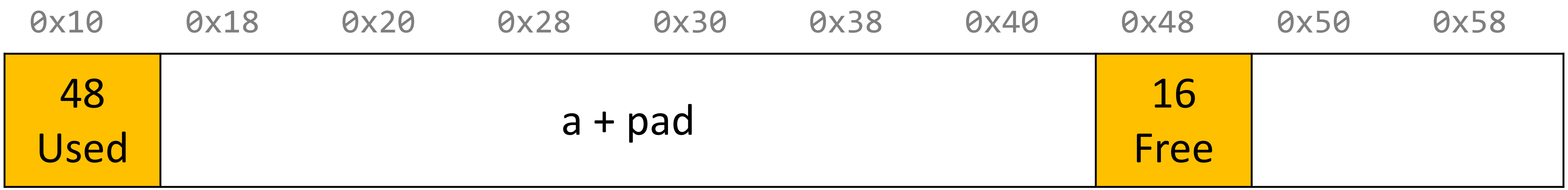


Realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a realloc is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

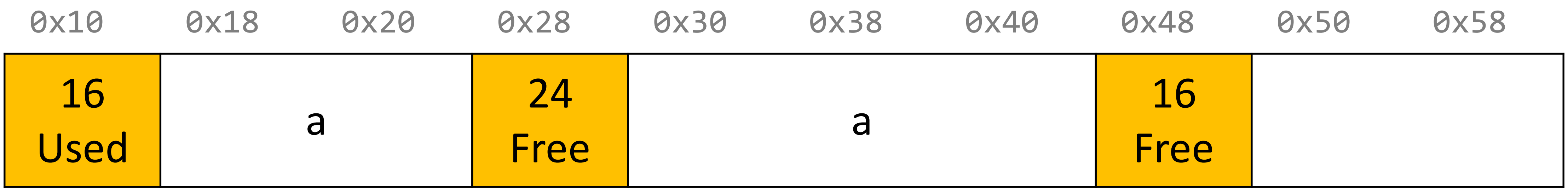


Realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a realloc is requesting to shrink, we can still use the same starting address.

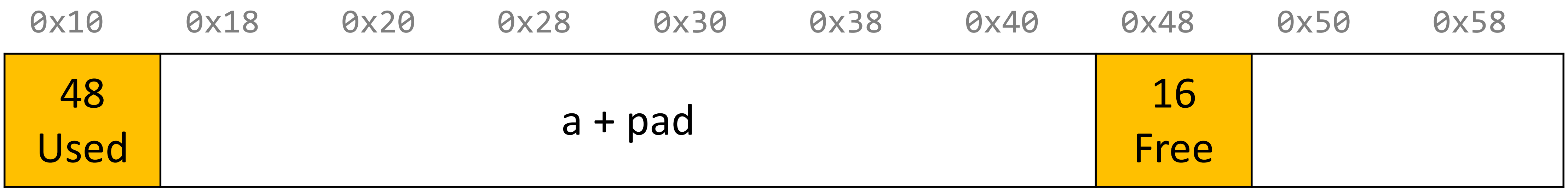
If we can, we should try to recycle the now-freed memory into another freed block.



Realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

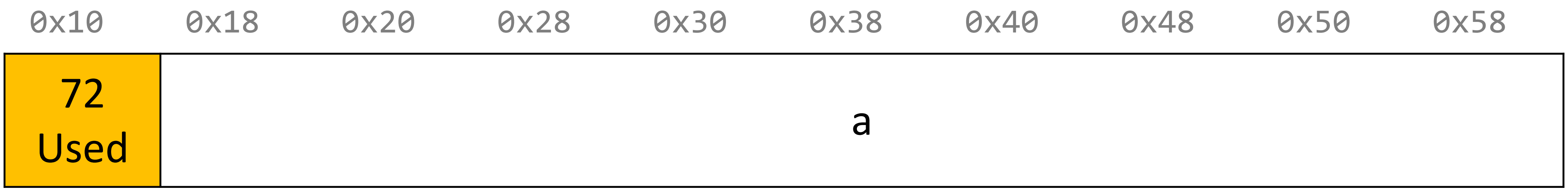


Realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

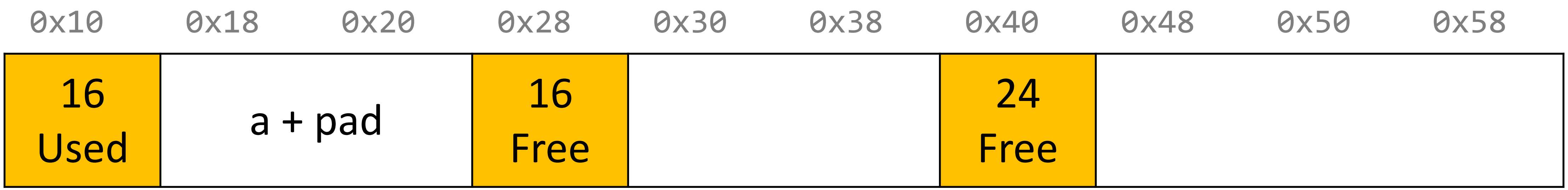
Now we can still return the same address.



Realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

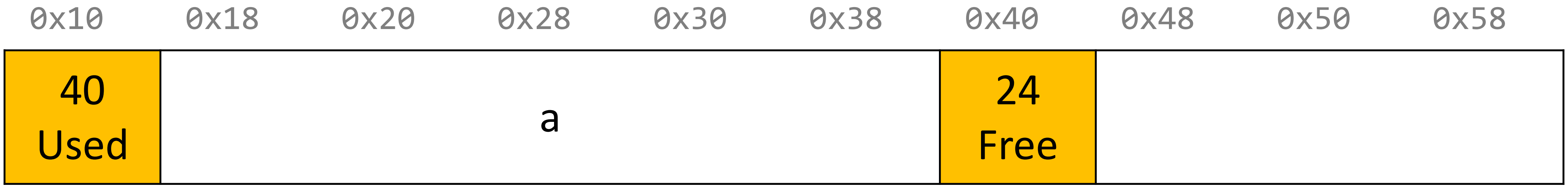
You should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

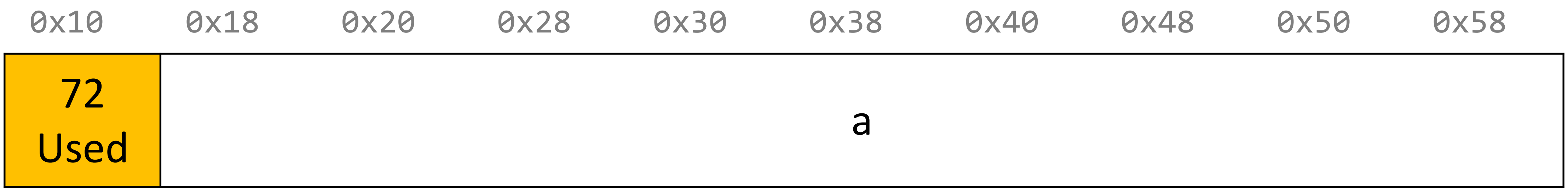
You should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

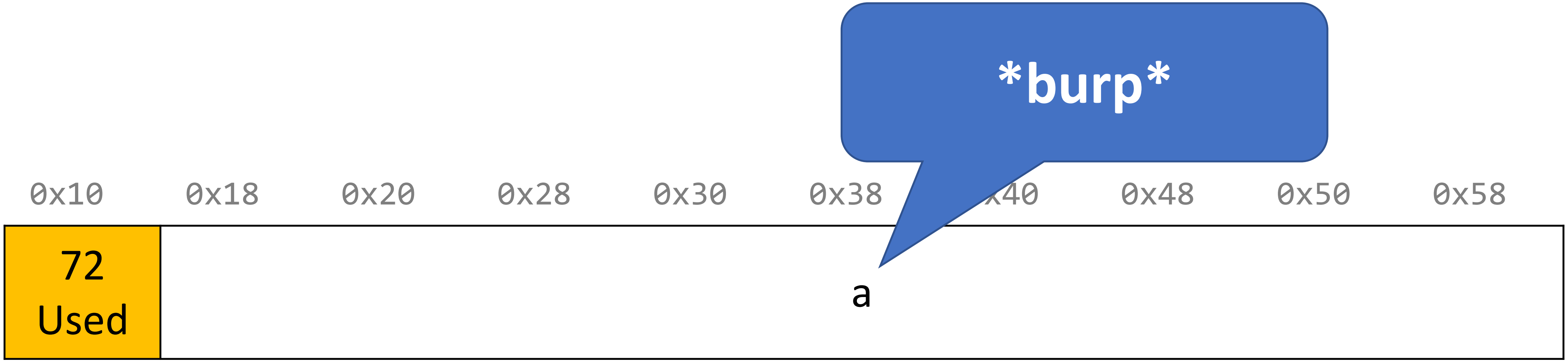
You should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

You should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc

- For the implicit allocator, we didn't worry too much about realloc. We always moved data when they requested a different amount of space.
  - Note: realloc can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.
- If you can't do an in-place realloc, then you should move the data elsewhere.

Assignment 7: Explicit Allocator

- **Must have** headers that track block information (size, status in-use or free) – you can copy from your implicit version
- **Must have** an explicit free list managed as a doubly-linked list, using the first 16 bytes of each free block's payload for next/prev pointers.
- **Must have** a malloc implementation that searches the explicit list of free blocks.
- **Must** coalesce a free block in free() whenever possible with its immediate right neighbor.
- **Must** do in-place realloc when possible. Even if an in-place realloc is not possible, you should still absorb adjacent right free blocks as much as possible until you either can realloc in place, or can no longer absorb and must realloc elsewhere.

Plan For Today

- Recap: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- Break: Announcements
- In-Place Realloc
- **Optimization**

Optimization

- Optimization is the task of making your program faster or more efficient with space or time. You've seen explorations of efficiency with Big-O notation!
- *Targeted, intentional* optimizations to alleviate bottlenecks can result in big gains. But it's important to only work to optimize where necessary.

Optimization

Most of what you need to do with optimization can be summarized in 3 easy steps:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things thing a lot, or on big inputs, make the primary algorithm's Big-O cost reasonable
3) **Let gcc do its magic from there**

GCC Optimization

- Today, we'll be comparing two levels of optimization in the gcc compiler:
  - `gcc –O0  //mostly just literal translation of C`
  - `gcc –O2  //enable nearly all reasonable optimizations`
  - (we use –Og, like –O0 but with less needless use of the stack)

- There are other custom and more aggressive levels of optimization, e.g.:
  - `-O3     //more aggressive than O2, trade size for speed`
  - `-Os     //optimize for size`
  - `-Ofast  //disregard standards compliance (!!)`

- Exhaustive list of gcc optimization-related flags:
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Example: Matrix Multiplication

Here's a standard matrix multiply, a triply-nested for loop:

```
void mmm(double a[][DIM], double b[][DIM], double c[][DIM], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
}
```

```
./mult          // -O0 (no optimization)
matrix multiply  25^2: cycles    0.44M
matrix multiply  50^2: cycles    3.13M
matrix multiply 100^2: cycles   24.80M
```

```
./mult_opt      // -O2 (with optimization)
matrix multiply  25^2: cycles    0.11M (opt)
matrix multiply  50^2: cycles    0.47M (opt)
matrix multiply 100^2: cycles    3.67M (opt)
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- Psychic Powers

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- ~~Psychic Powers~~

(kidding.)

Constant Folding

Constant Folding pre-calculates constants at compile-time where possible.

```
int seconds = 60 * 60 * 24 * n_days;
```

What is the consequence of this for you as a programmer? What should you do differently or the same knowing that compilers can do this for you?

Constant Folding

```
int CF(int param) {
    char arr[0x55];

    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a*param + (a + 0x15/c + strlen("hello")*b - 0x37)/4;
}
```

Constant Folding: Before (-O0)

```
0000000000400726 <CF>:
  400726:       55                      push   %rbp
  400727:       53                      push   %rbx
  400728:       48 83 ec 08             sub    $0x8,%rsp
  40072c:       89 fd                   mov    %edi,%ebp
  40072e:       f2 0f 10 05 ba 05 00    movsd  0x5ba(%rip),%xmm0
  400735:       00
  400736:       e8 c5 fe ff ff          callq  400600 <sqrt@plt>
  40073b:       f2 0f 2c c8             cvttsd2si %xmm0,%ecx
  40073f:       69 ed 07 01 00 00       imul   $0x107,%ebp,%ebp
  400745:       b8 15 00 00 00          mov    $0x15,%eax
  40074a:       99                      cltd
  40074b:       f7 f9                   idiv   %ecx
  40074d:       8d 98 07 01 00 00       lea    0x107(%rax),%ebx
  400753:       bf e4 0c 40 00          mov    $0x400ce4,%edi
  400758:       e8 73 fe ff ff          callq  4005d0 <strlen@plt>
  40075d:       48 69 c0 53 57 00 00    imul   $0x5753,%rax,%rax
  400764:       48 63 db                movslq %ebx,%rbx
  400767:       48 8d 44 18 c9          lea    -0x37(%rax,%rbx,1),%rax
  40076c:       48 c1 e8 02             shr    $0x2,%rax
  400770:       01 e8                   add    %ebp,%eax
  400772:       48 83 c4 08             add    $0x8,%rsp
  400776:       5b                      pop    %rbx
  400777:       5d                      pop    %rbp
  400778:       c3                      retq
```

Constant Folding: After (-O2)

```
0000000000400800 <CF>:
  400800:       69 c7 07 01 00 00       imul   $0x107,%edi,%eax
  400806:       05 61 6d 00 00          add    $0x6d61,%eax
  40080b:       c3                      retq
```

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

Strength Reduction

Strength reduction changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

Assignment 7: Optimization

- Explore various optimizations you can make to your code to reduce instruction count.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using callgrind
    - Optimize using –O2
    - And more…

Plan For Today

- **Recap**: Heap Allocators – Bump and Implicit
- Method 3: Explicit Free List Allocator
- Coalescing
- **Break:** Announcements
- In-Place Realloc
- Optimization

Next time: CS107 recap and parting words