# CS107 Lecture 15
## Managing the Heap

reading:

*B&O 9.9, 9.11*

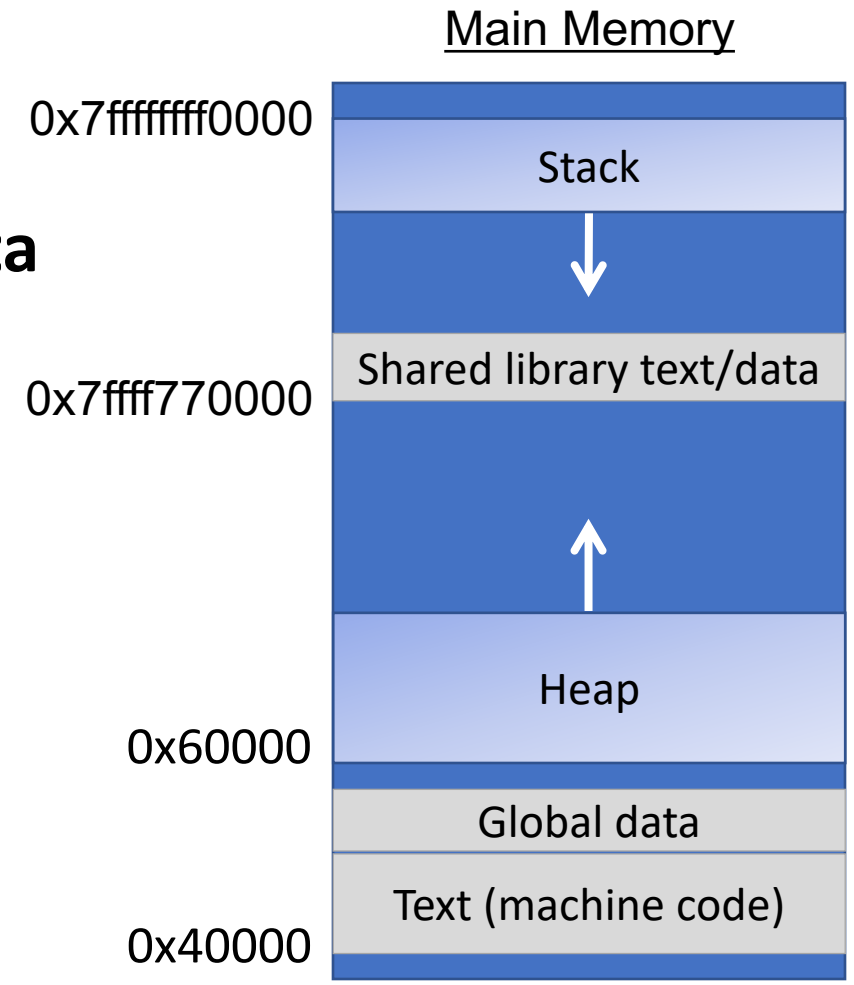
Slides by Jerry Cain and Lisa Yan, who leveraged prior work by Nick Troccoli, Julie Zelenski, Marty Stepp, Cynthia Lee, Chris Gregg, and others.

# CS107 Topic 7: How do the core `malloc/realloc/free` memory-allocation operations work?

# Running a program

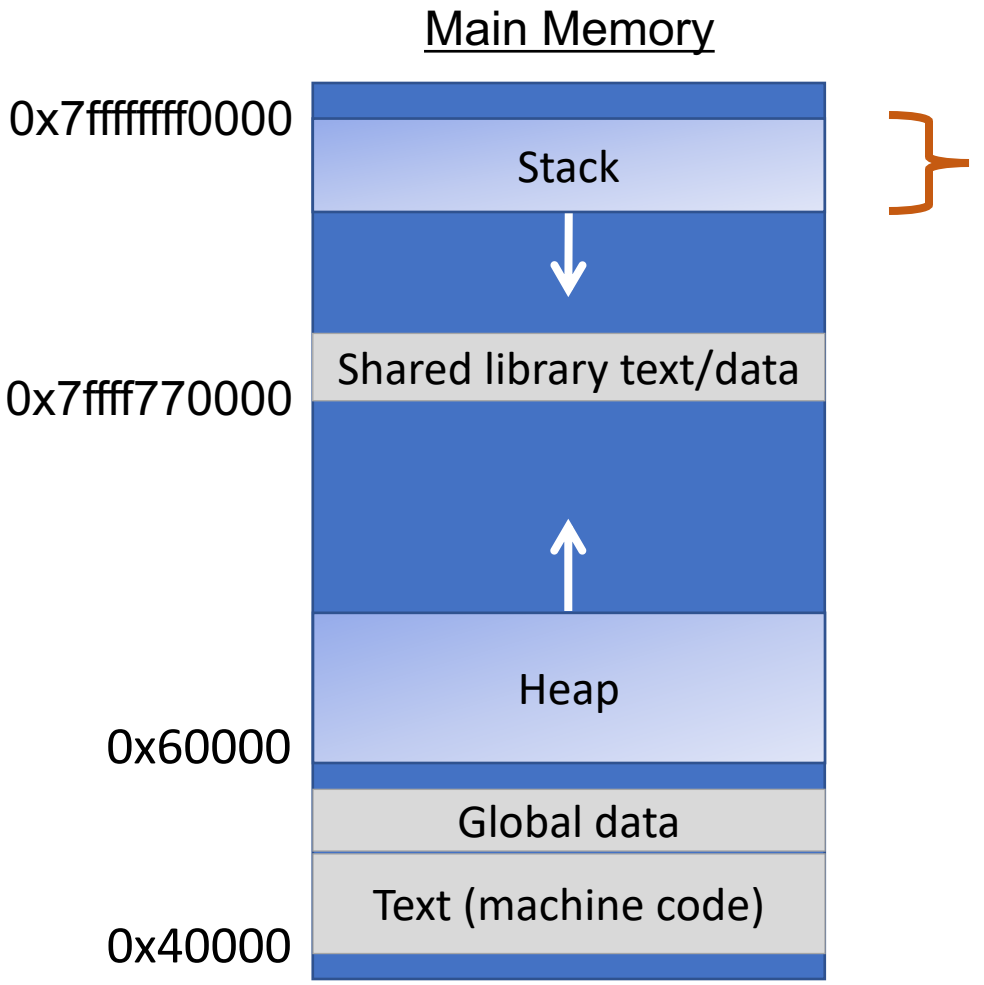
- **Creates new process**
- **Sets up address space/segments**
- **Read executable file, load instructions, global data**
  Mapped from file into gray segments
- **Libraries loaded on demand**

- **Set up stack**
  Reserve stack segment, init %rsp, call main
- **malloc written in C, will init self on use**
  Asks OS for large memory region,
  parcels out to service requests



# The Stack



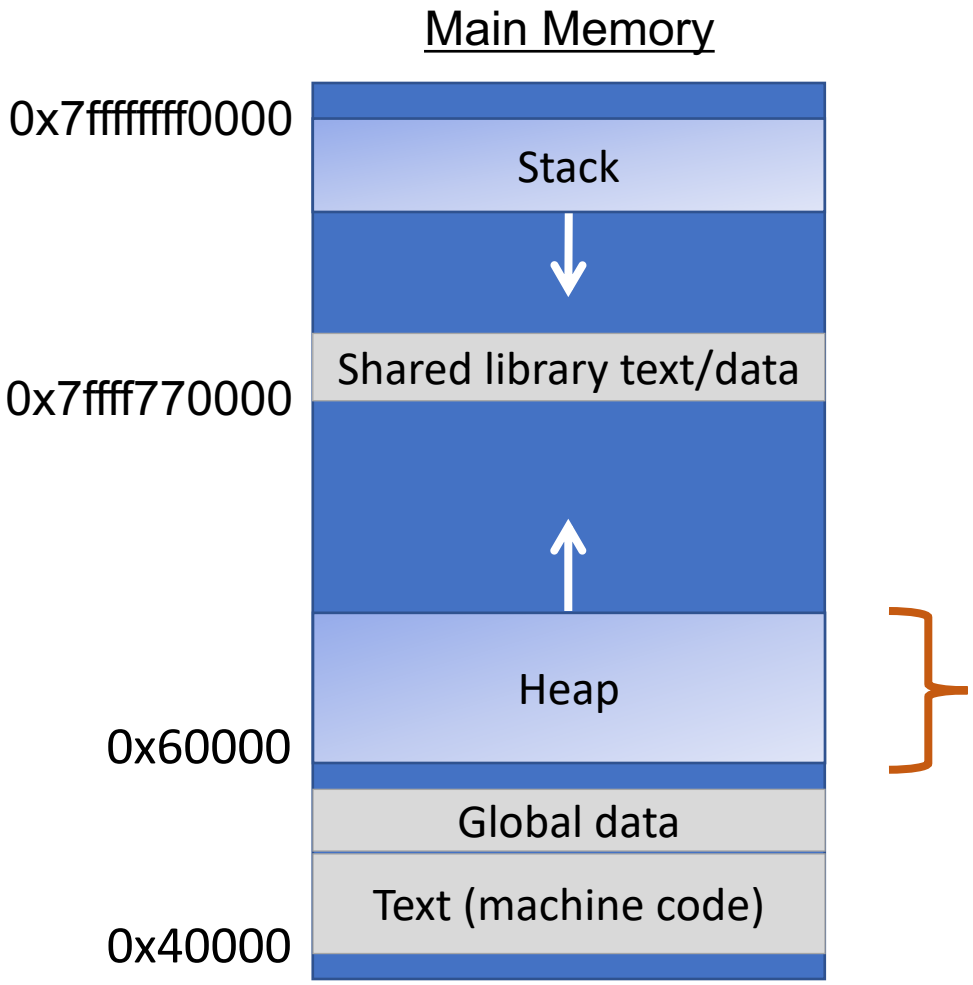



**Stack memory "goes away"** after function call ends.

**Automatically managed** at compile-time by gcc

Last lecture:
Stack management == moving %rsp around (pushq, popq, mov)

# Today: The Heap




**Heap memory persists** until caller indicates it no longer needs it.

**Managed** by C standard library functions (malloc, realloc, free)

This lecture:
How does heap management work?

# Plan For Today

- Recap: the heap
- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator
- **Break:** Announcements
- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator

# Your role so far: Client

```
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.

# Your role now: Heap Hotel Concierge


(aka **Heap Allocator**)

http://screencrave.com/wp-content/uploads/2014/03/the-grand-budapest-hotel-anderson-image-2.jpg

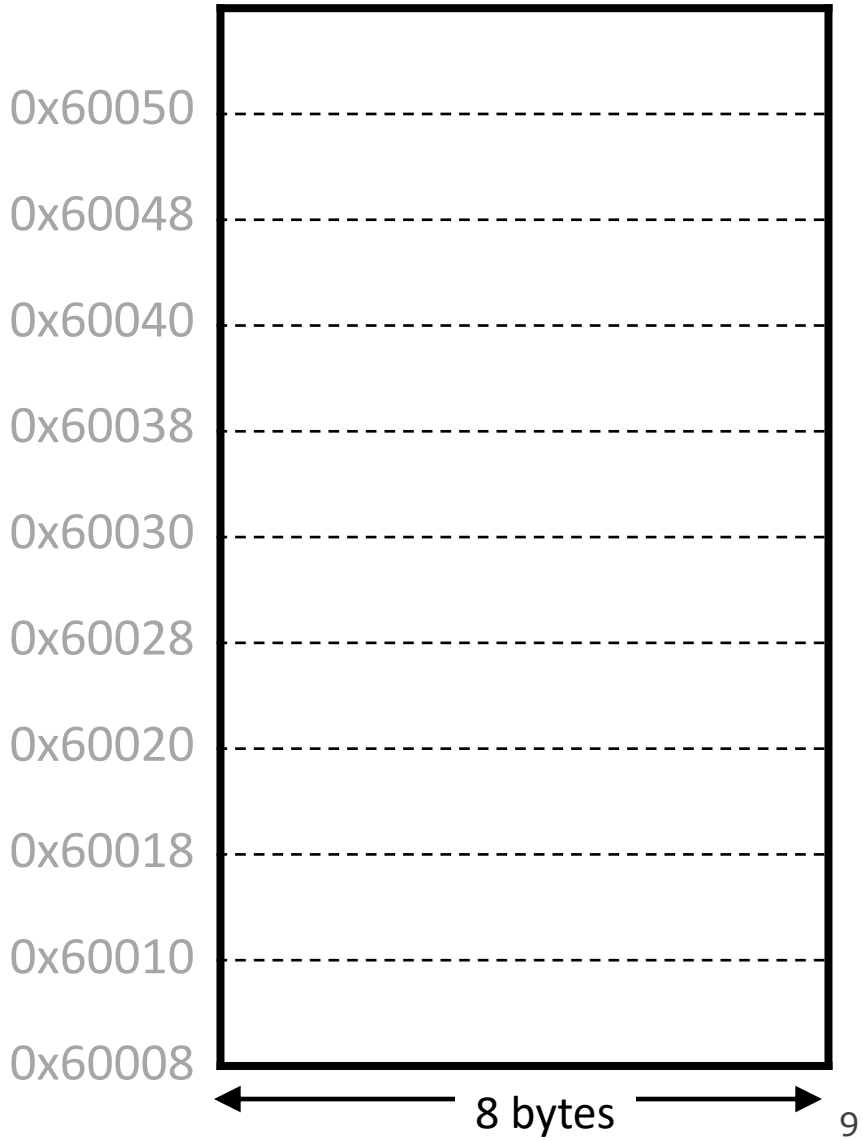
# What is a heap allocator?

A **heap allocator** is a set of functions that fulfills requests for heap memory.

- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

**Allocator metadata**

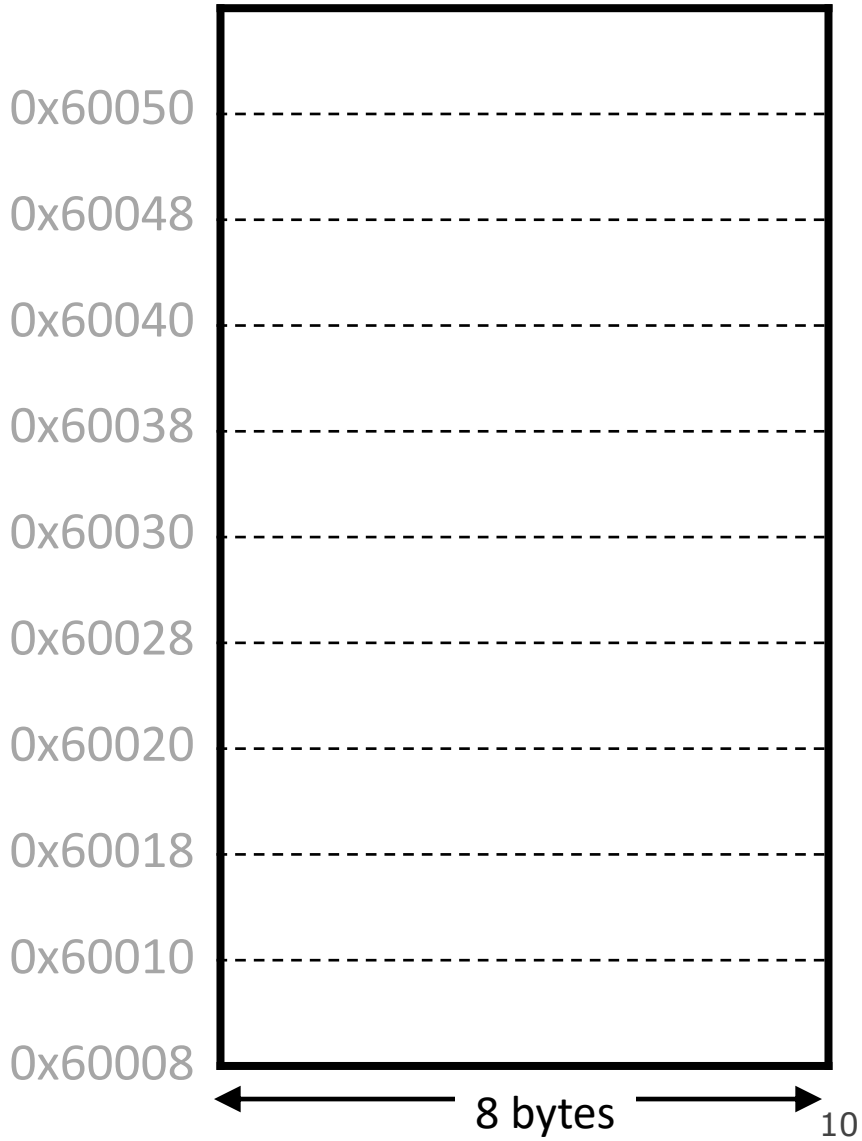
| | |
|---|---|
| `heap_start` | `0x60008` |
| `heap_size` | `80` |



# Brainstorming exercise

```
1 void *a, *b, *c;
2 a = malloc(8);
3 b = malloc(4);
4 a = realloc(a, 40);
5 free(b);
6 c = malloc(24);
```

1. What might heap memory look like after these requests?
2. What additional info might the allocator need to track while servicing requests? Should we store this in global data or in the heap itself?
3. What can an allocator ***not*** do?



# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: **Implicit Free List**, Explicit Free List
- Policy decisions/performance
- Designing your own heap allocator

# Heap Allocator Functions

```
void *malloc(size_t size);

void free(void *ptr);

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

# Heap Allocator Requirements

A heap allocator must…

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

**Cannot reorder or accumulate** allocation requests.
**Cannot** assume all allocations will have a matching free request.

**Cannot** allocate already-allocated memory or modify allocated blocks. (more later)
**Can** manipulate and modify free memory.

**Alignment requirement**: for GNU malloc (libc malloc) on Linux

# Heap Allocator Performance Goals

## 1. Utilization

(not too many holes in space)

## 2. Throughput

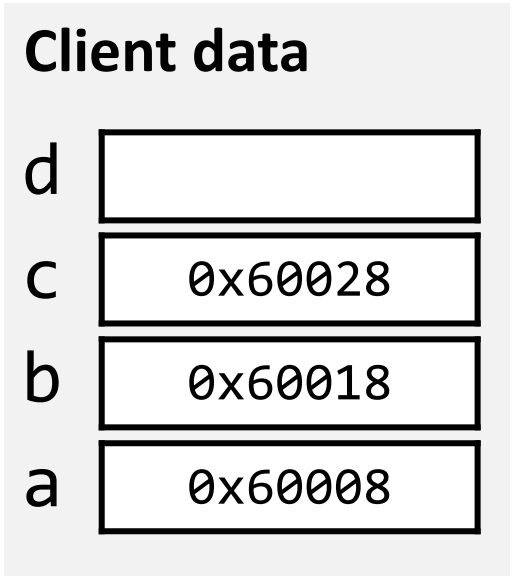
(speed of handling requests)

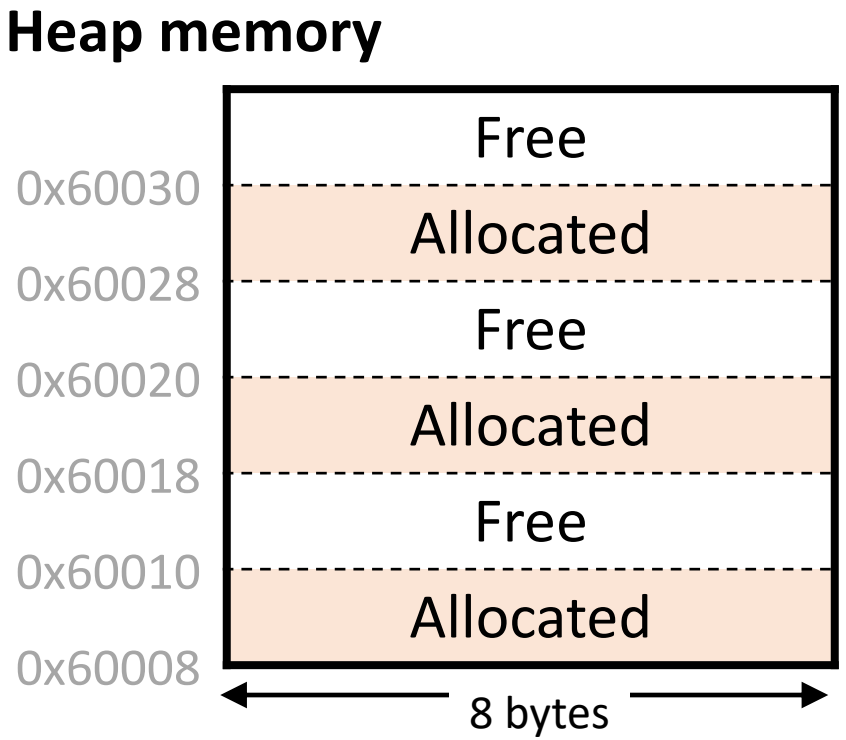
Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

# Utilization

The primary cause of poor utilization is **fragmentation**. **Fragmentation** occurs when otherwise unused memory is not available to satisfy allocation requests:

```
void *d = malloc(24); // NULL (out of heap space)
```

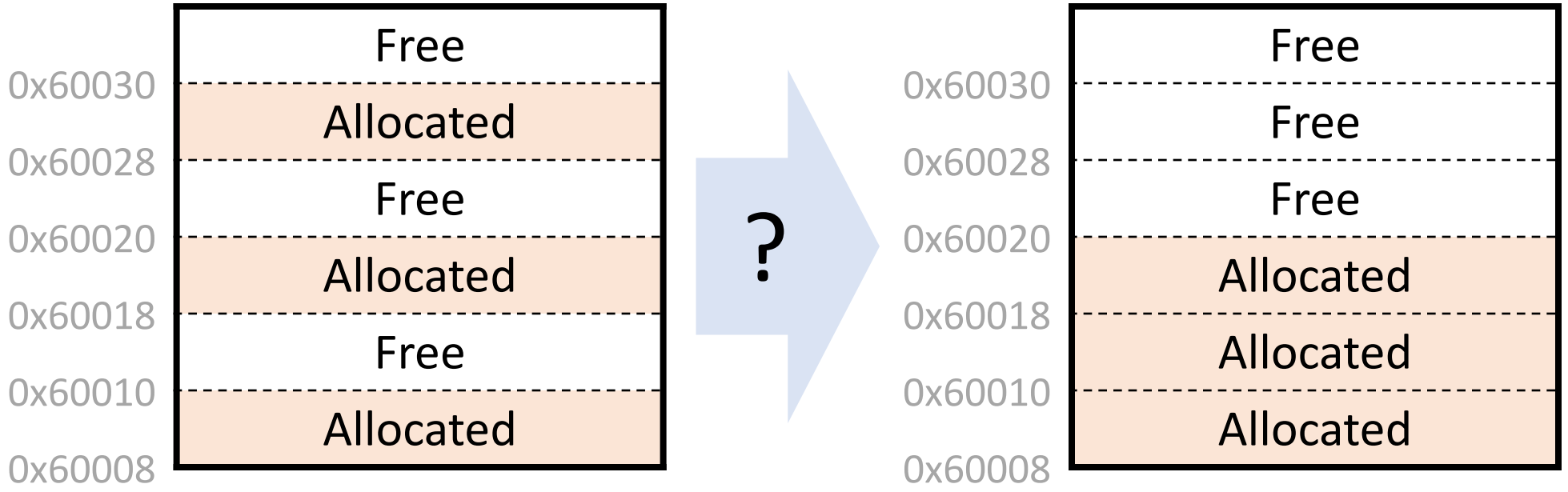


# Utilization: Defragmenting

Could the heap manager pause and do a "**defrag**" operation at this point?

```
void *d = malloc(24);
```




**Client data**

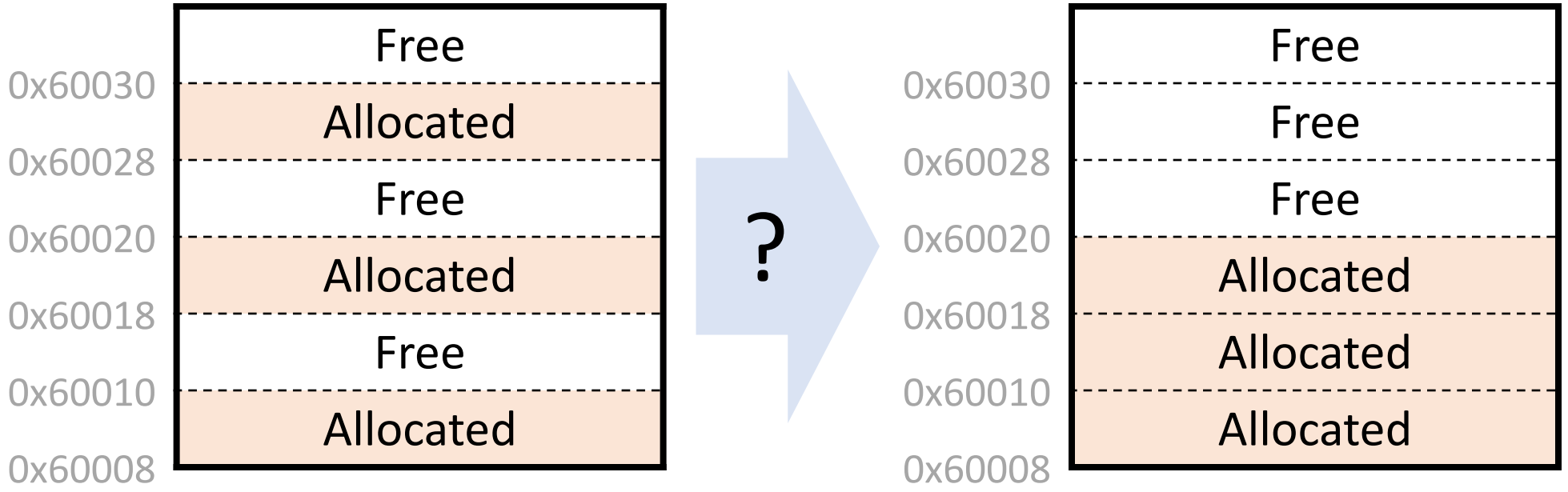
| | |
|---|---|
| d | |
| c | 0x60028 |
| b | 0x60018 |
| a | 0x60008 |

A. YES, great idea!
B. YES, it can be done, but not a good idea for some reason (e.g., not efficient use of time)
C. NO, it can't be done!

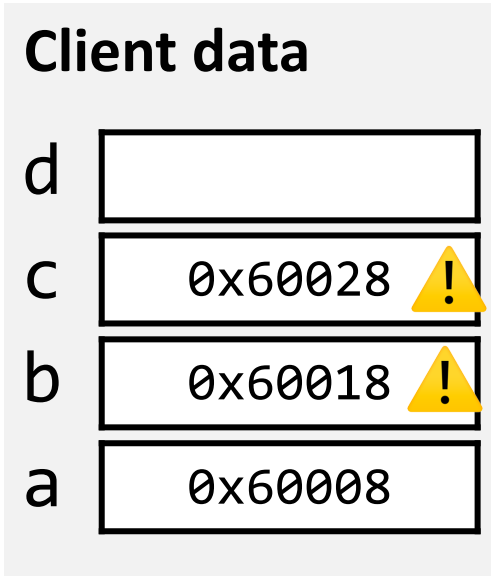
# Utilization: Defragmenting

Could the heap manager pause and do a "**defrag**" operation at this point?

```
void *d = malloc(24);
```




**Client data**




A. YES, great idea!

B. YES, it can be done, but not a good idea for some reason (e.g., not efficient use of time)

C. NO, it can't be done!

Client still has pointers to allocated memory!

# Heap Allocator Performance Goals

## 1. Utilization

Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

## 2. Throughput

Maximize **throughput**, or the number of requests completed per unit time.

i.e., minimizing the average time to satisfy a request.

These two goals are often in conflict with each other ☺

# Other Goals

1. Utilization 2. Throughput

Other desirable goals:

**Locality** (“similar” blocks allocated close in space)

**Robust** (handle client errors)

**Ease of implementation/maintenance**

# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: **Implicit Free List**, Explicit Free List
- Policy decisions/performance
- Designing your own heap allocator

# Bump Allocator Performance

## 1. Utilization

**Never** reuses memory

## 2. Throughput

**Ultra fast**, short routines

# Bump Allocator

```
1 void *a, *b, *c, *d;
2 a = malloc(8);
3 b = malloc(4);
4 c = malloc(64);
5 free(b);
6 c = malloc(8);
```

**Client data**

d

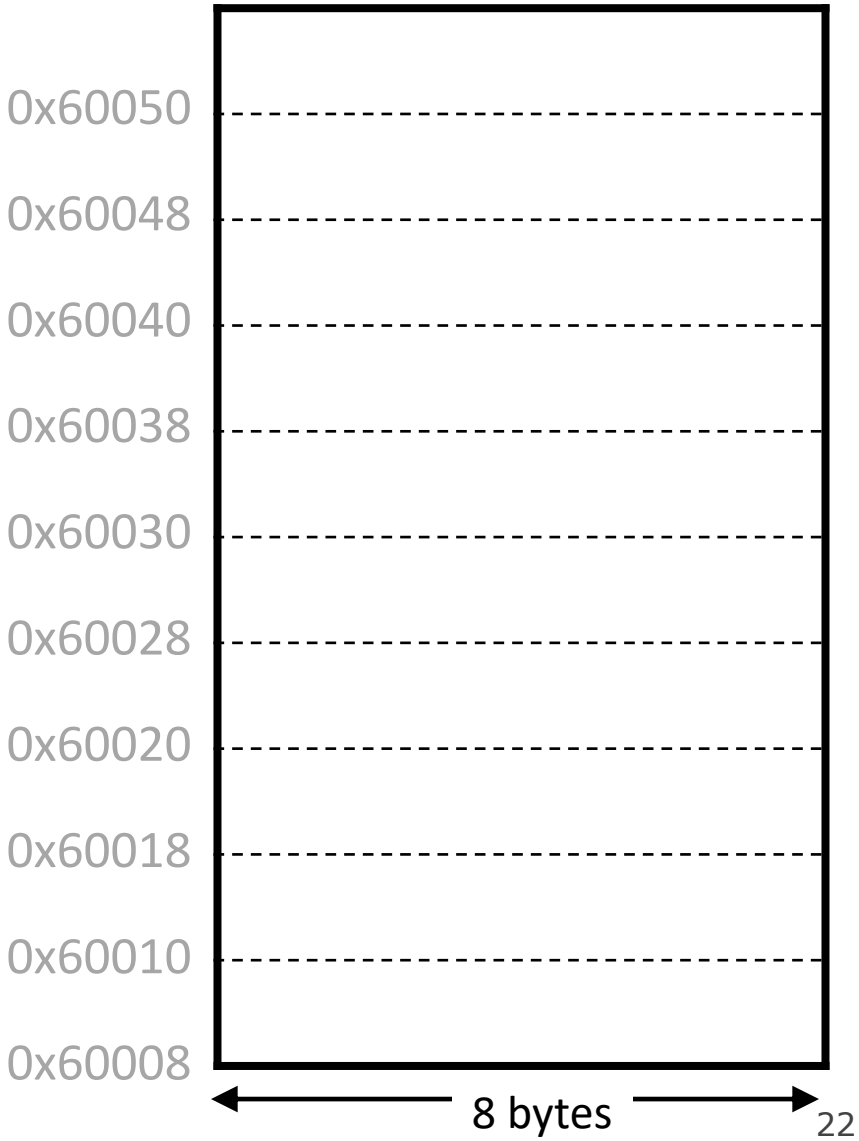
c

b

a

**Bump allocator metadata**

| | |
|---|---|
| heap_start | 0x60008 |
| bytes_used | 0 |
| heap_size | 80 |



# Bump Allocator

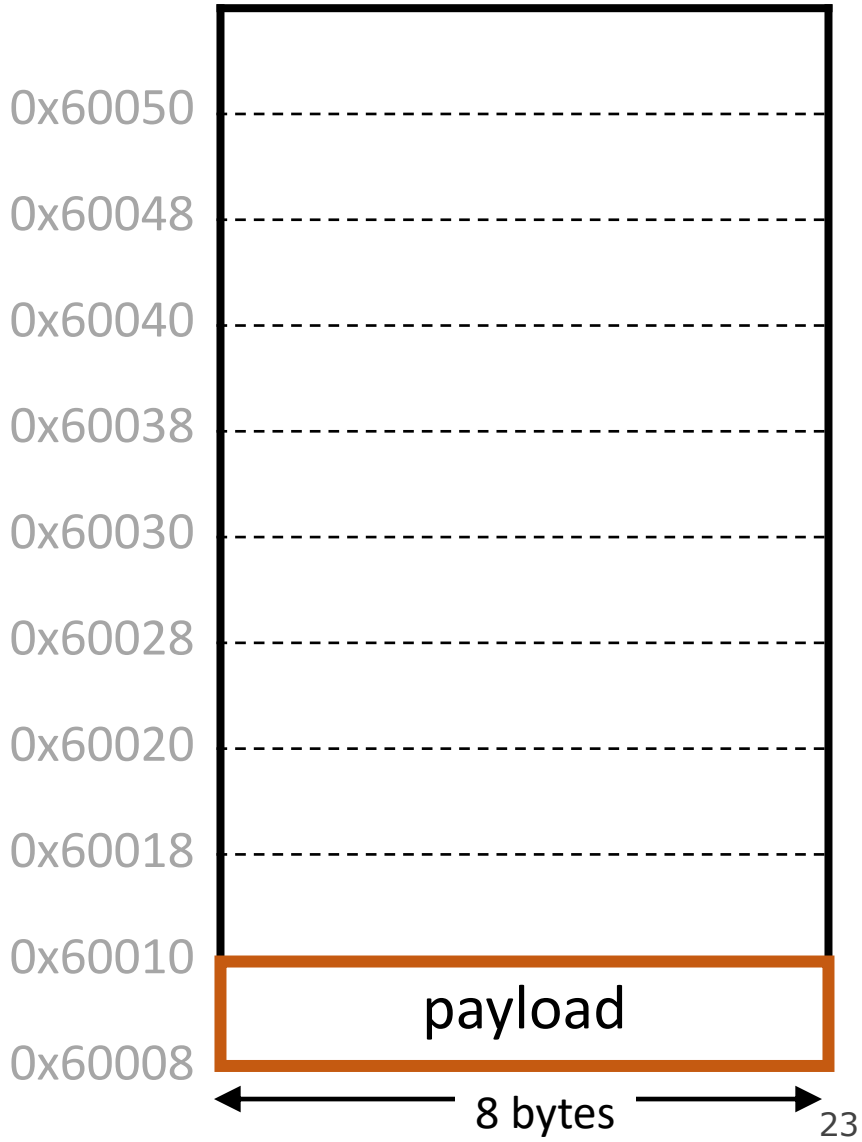
```
1 void *a, *b, *c, *d;
2 a = malloc(8);
3 b = malloc(4);
4 c = malloc(64);
5 free(b);
6 c = malloc(8);
```

**Client data**

| | |
|---|---|
| d | |
| c | |
| b | |
| a | 0x60008 |

**Bump allocator metadata**

| | |
|---|---|
| heap_start | 0x60008 |
| bytes_used | 8 |
| heap_size | 80 |



# Bump Allocator

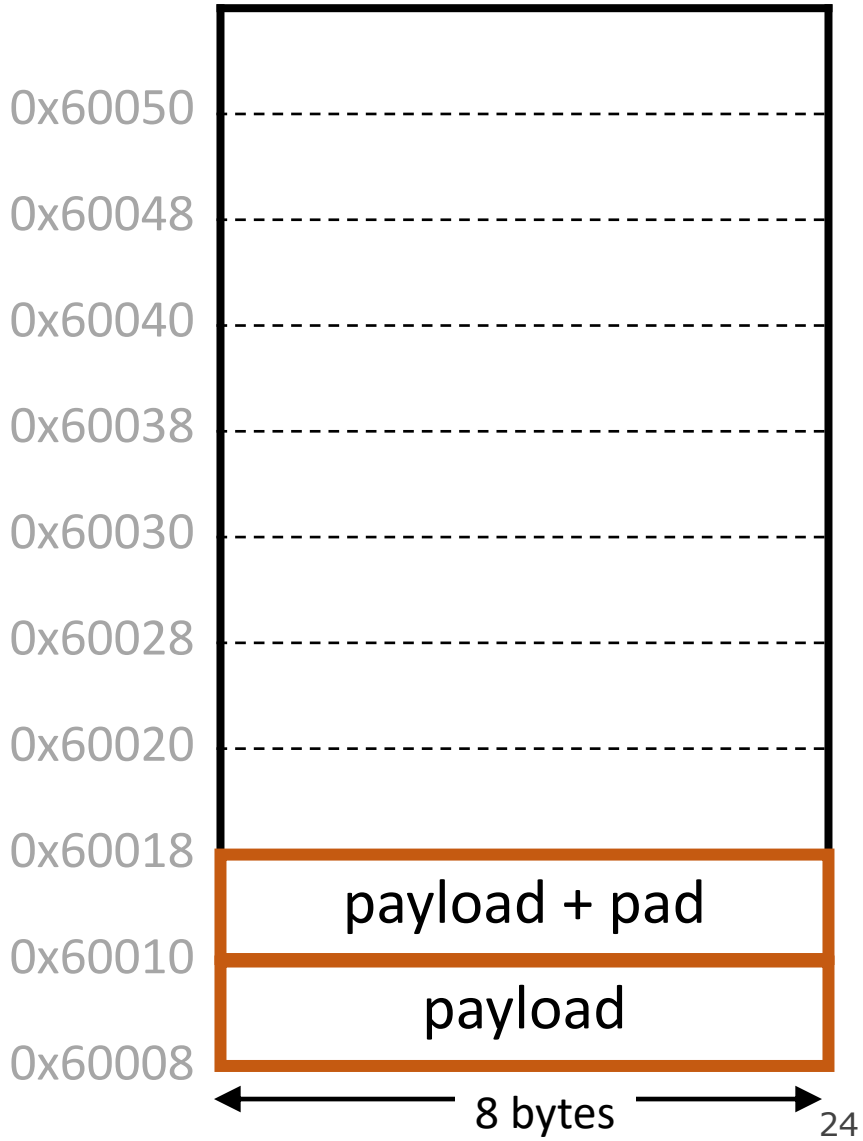
```
1 void *a, *b, *c, *d;
2 a = malloc(8);
3 b = malloc(4);
4 c = malloc(64);
5 free(b);
6 c = malloc(8);
```

**Client data**

| | |
|---|---|
| d | |
| c | |
| b | 0x60010 |
| a | 0x60008 |

**Bump allocator metadata**

| | |
|---|---|
| heap_start | 0x60008 |
| bytes_used | 16 |
| heap_size | 80 |



# Bump Allocator

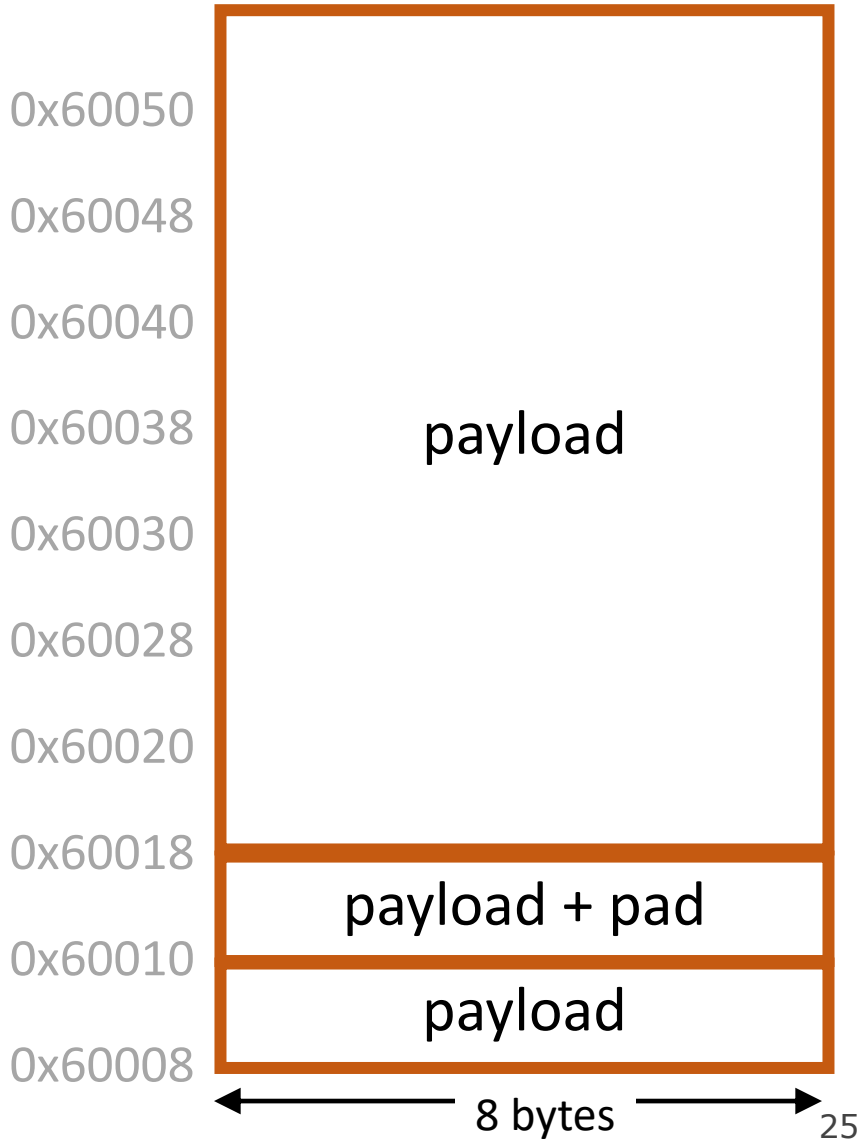
```
1 void *a, *b, *c, *d;
2 a = malloc(8);
3 b = malloc(4);
4 c = malloc(64);
5 free(b);
6 c = malloc(8);
```

**Client data**

| | |
|---|---|
| d | |
| c | 0x60018 |
| b | 0x60010 |
| a | 0x60008 |

**Bump allocator metadata**

| | |
|---|---|
| heap_start | 0x60008 |
| bytes_used | 80 |
| heap_size | 80 |



# Bump Allocator

```
1 void *a, *b, *c, *d;
2 a = malloc(8);
3 b = malloc(4);
4 c = malloc(64);
5 free(b);
6 c = malloc(8);
```

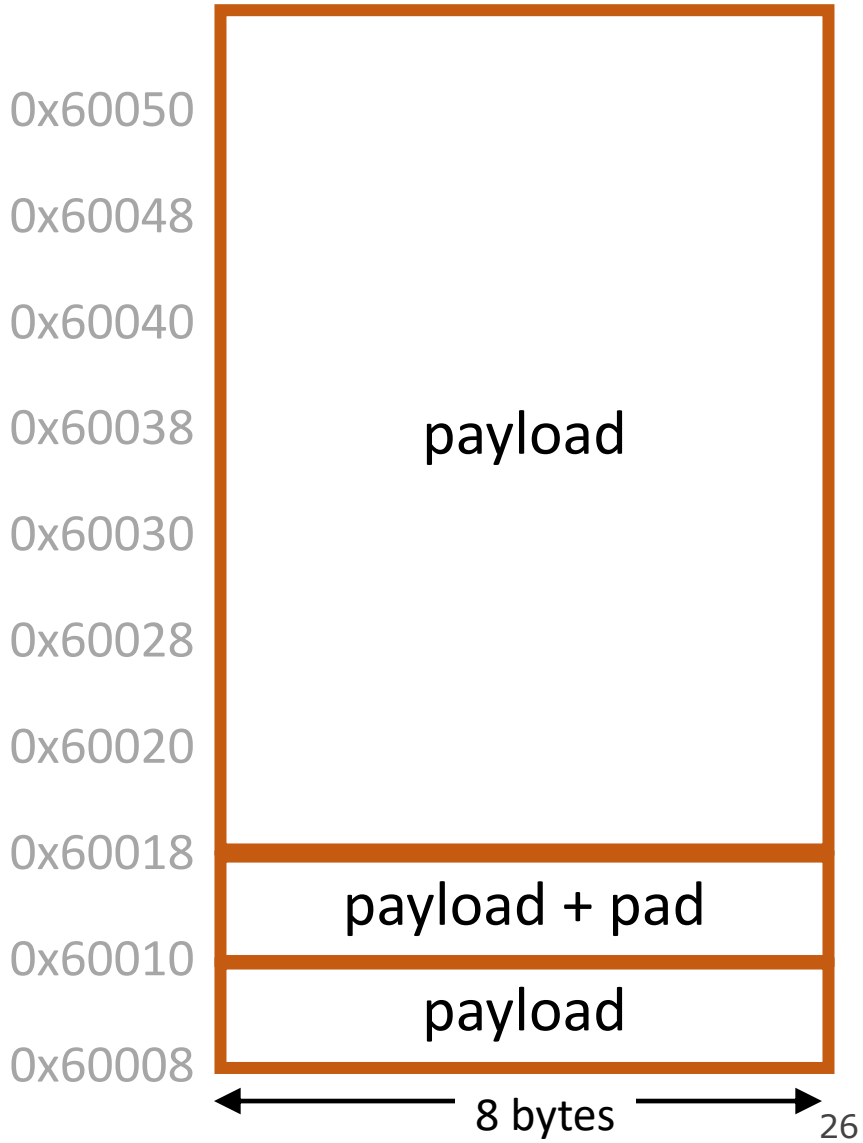
(no-op)

**Client data**

| | |
|---|---|
| d | |
| c | 0x60018 |
| b | 0x60010 |
| a | 0x60008 |

**Bump allocator metadata**

| | |
|---|---|
| heap_start | 0x60008 |
| bytes_used | 80 |
| heap_size | 80 |



# Bump Allocator

```
1 void *a, *b, *c, *d;
2 a = malloc(8);
3 b = malloc(4);
4 c = malloc(64);
5 free(b);
6 c = malloc(8);
```

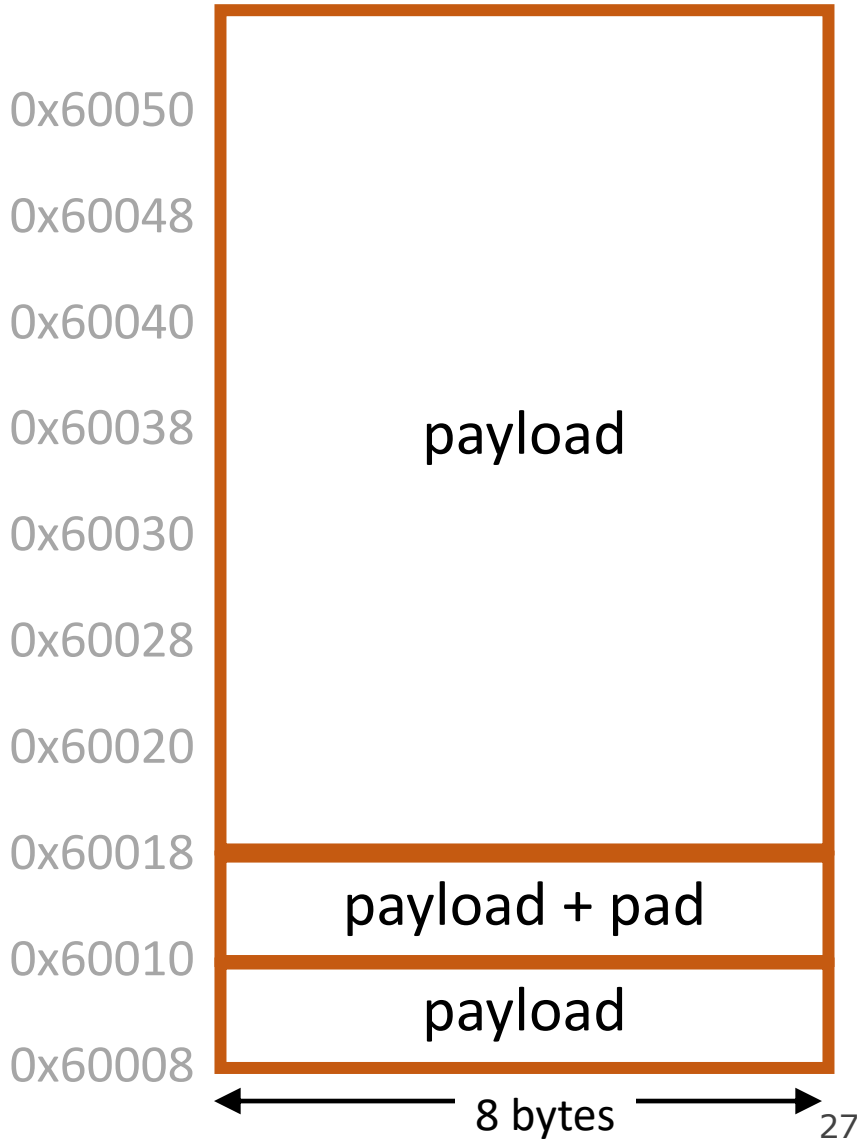
returns NULL
(heap out of space)

**Client data**

| | |
|---|---|
| d | NULL |
| c | 0x60018 |
| b | 0x60010 |
| a | 0x60018 |

**Bump allocator metadata**

| | |
|---|---|
| heap_start | 0x60008 |
| bytes_used | 80 |
| heap_size | 80 |



# Bump Allocator

- A **bump allocator** is a heap allocator design that simply:
  - Allocates the next available memory address upon an allocate request.
  - 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.

Next up: Strike a better balance
of utilization/throughput performance.

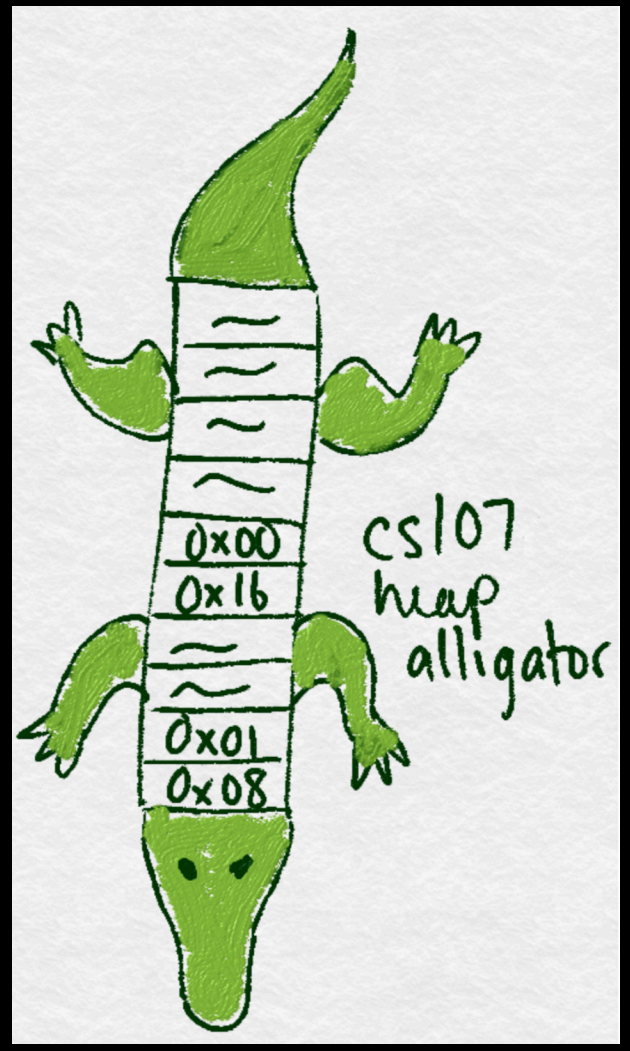
# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: **Implicit Free List**, Explicit Free List
- Policy decisions/performance
- Designing your own heap allocator

# Announcements

- We are working to process any outstanding regrade requests.
- Reminder of Collaboration Policy for assign6 and beyond

# Joke break


0x00
0x16
0x01
0x08
cs107
heap
alligator


SHOOTING STARS
SHOOT
SHOOT
SHOOT
SHOOT
SHOOT
SHOOT
SHOOT
SHOOT
SHOOT
@NATHANWPYLE

# Working with multiple files in assign7

`bump.c`

# Tips for assign7

**View/edit multiple buffers** (vim)

| | |
|---|---|
| `:split` | horizontal split |
| `:vsplit` | vertical split |
| Ctrl-w <arrow> | change to buffer |
| `:q` | quit current buffer |

**Search file**

| | |
|---|---|
| `/asdf` | find next match of key |
| n or N | next or previous match |

Emacs:
https://web.stanford.edu/class/cs107/resources/emacs_refcard.pdf

**More gdb tui mode**

**layout split # C code, assembly, gdb**

**layout src # C code and gdb**

focus cmd # arrows work in gdb

focus src # arrows work in C code

focus asm # arrows work in assembly

ctrl-x a # exit

Ctrl-l or refresh # refresh layout

# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: **Implicit Free List**, Explicit Free List
- Policy decisions/performance
- Designing your own heap allocator

# Managing block information

**Option 1**: Separate housekeeping

- Free/in-use information maintained in list/table
- Often not used in practice
  - Need global memory or large bookkeeping area in heap
  - Tools like Valgrind or special case allocators


Global data

**Option 2**: Block header

- Block information stored in heap memory that precedes payload
- Most common approach in current use


Heap memory

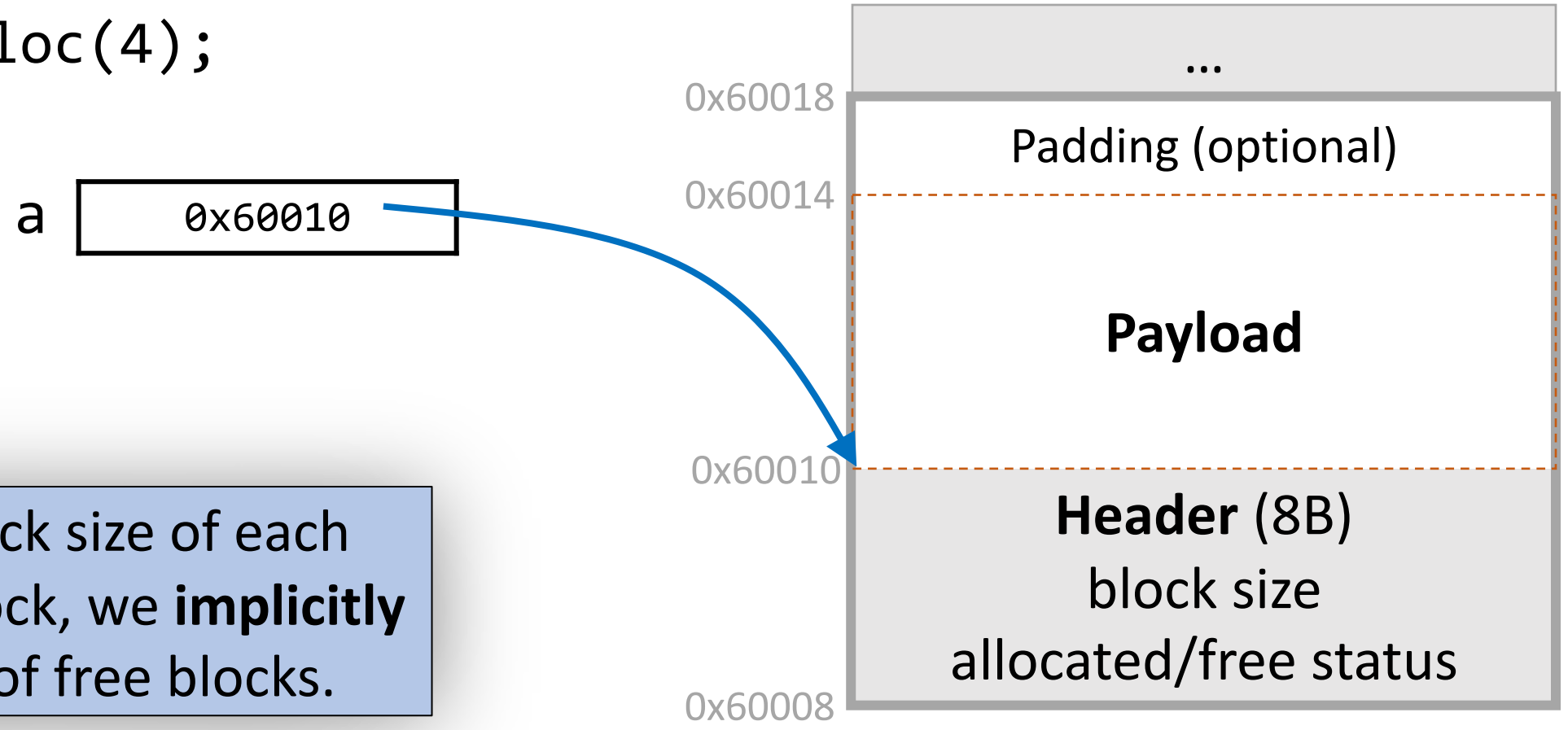
In your assignment: Use block headers with 500B of global data

# 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: Design idea

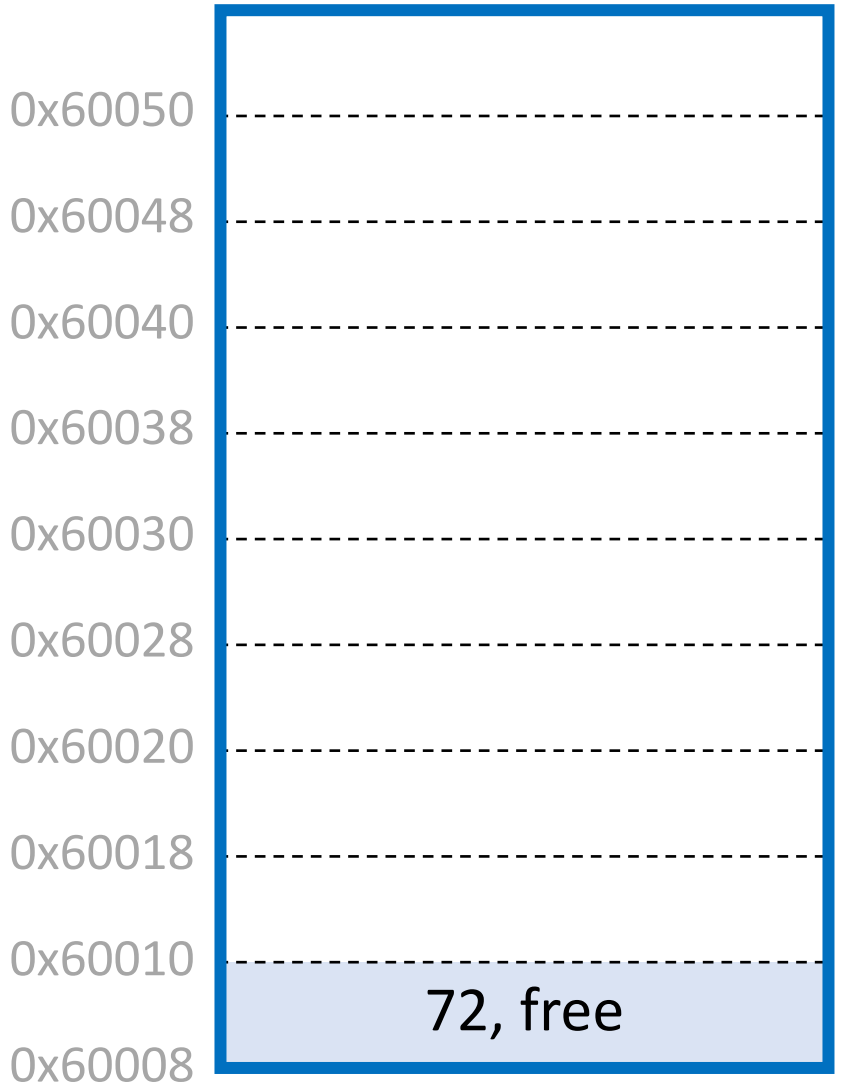
```
void *a = malloc(4);
```




By storing the block size of each allocated/free block, we **implicitly** have a linked **list** of free blocks.

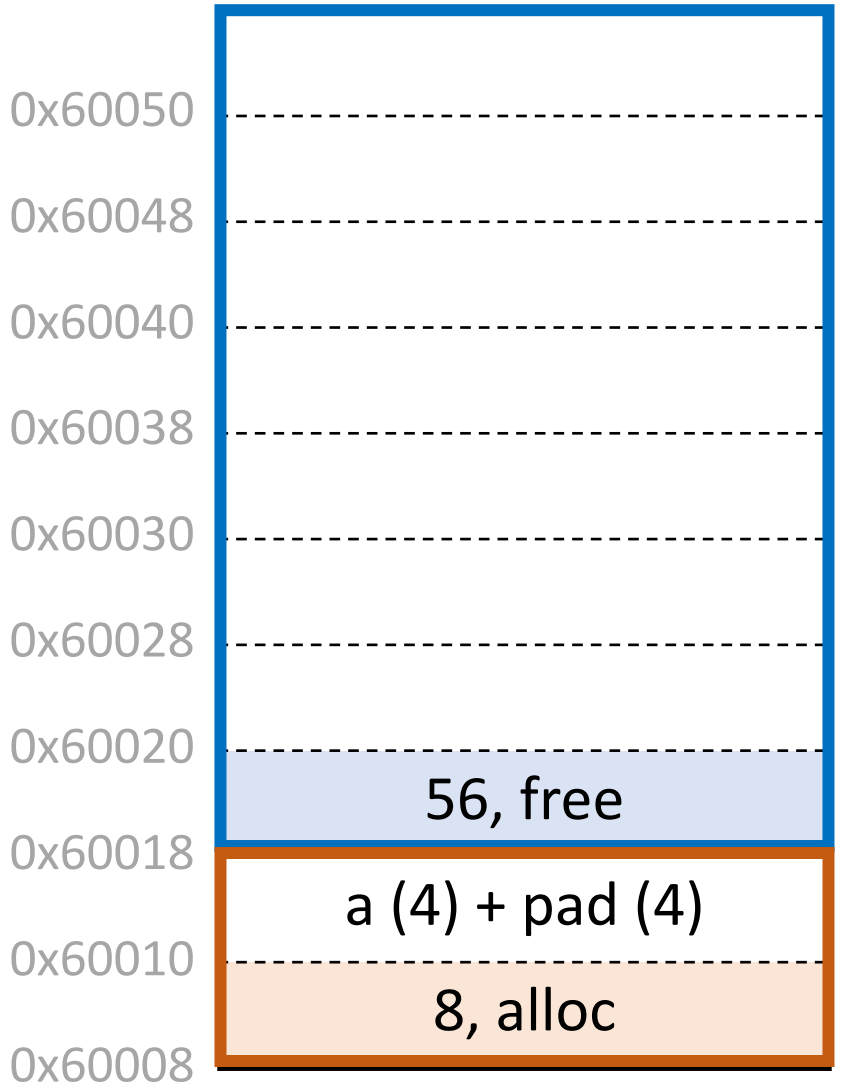
# Implicit Free List Allocator

```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```



# Implicit Free List Allocator

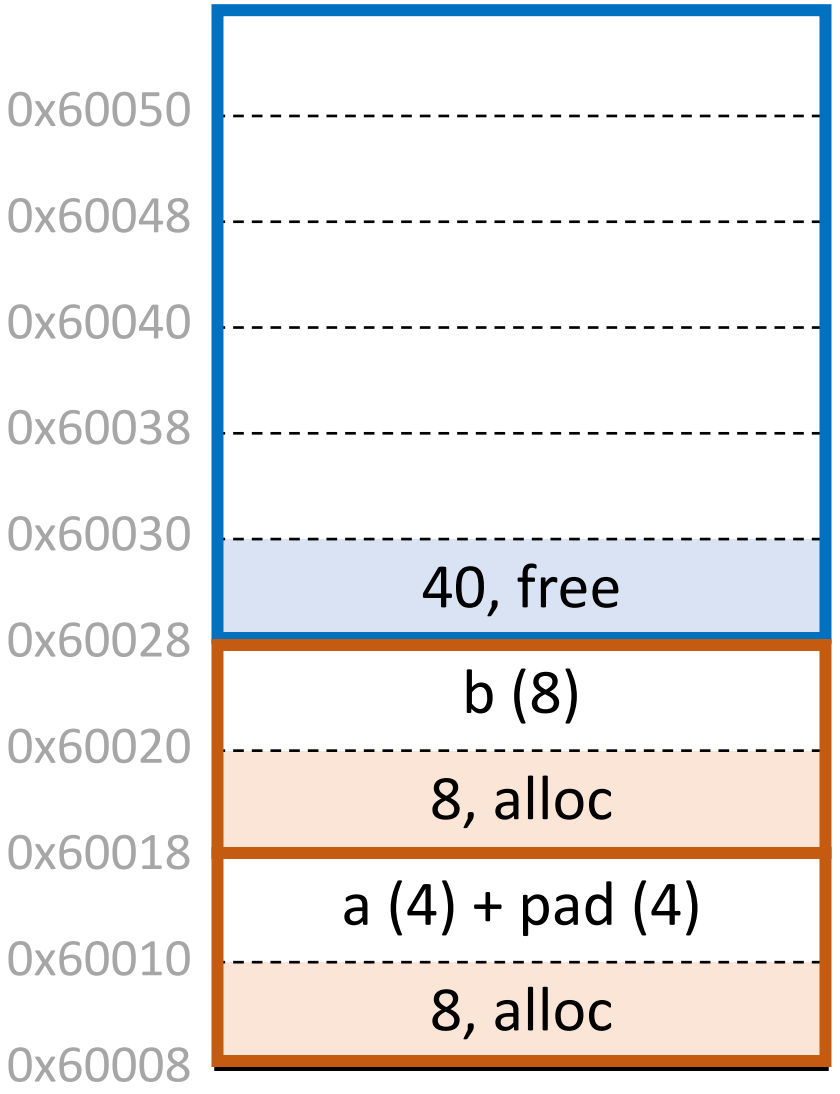
```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```




a | 0x60010

# Implicit Free List Allocator

```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```

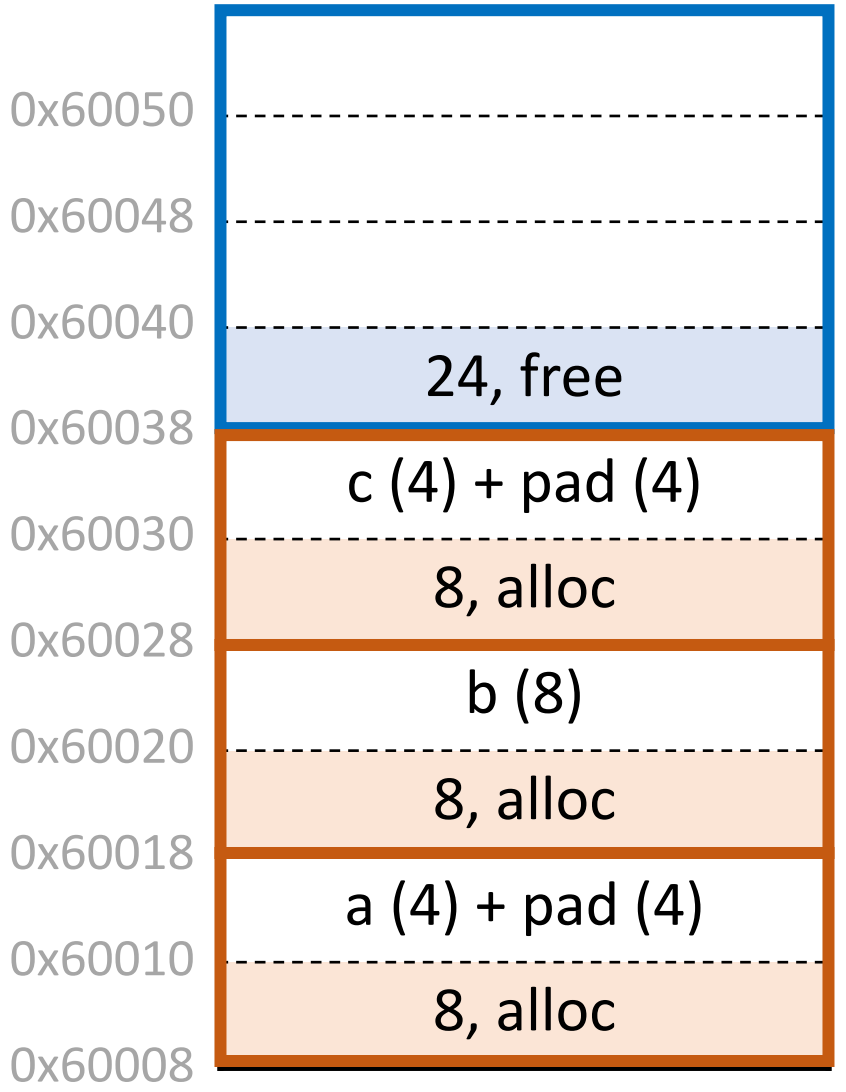



| | |
|---|---|
| b | 0x60020 |
| a | 0x60010 |

# Implicit Free List Allocator

```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```

| | |
|---|---|
| c | 0x60030 |
| b | 0x60020 |
| a | 0x60010 |



# Implicit Free List Allocator

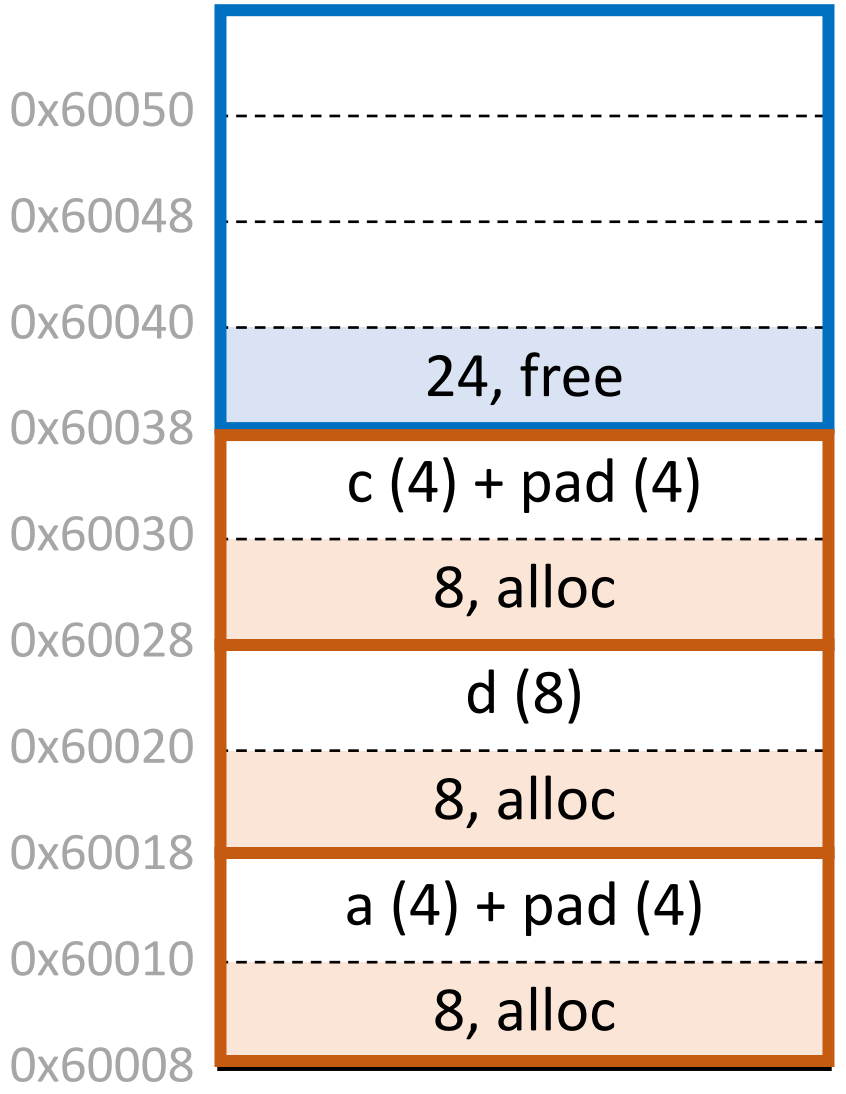
```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```

| | |
|---|---|
| c | 0x60030 |
| b | 0x60020 |
| a | 0x60010 |

# Implicit Free List Allocator

```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```

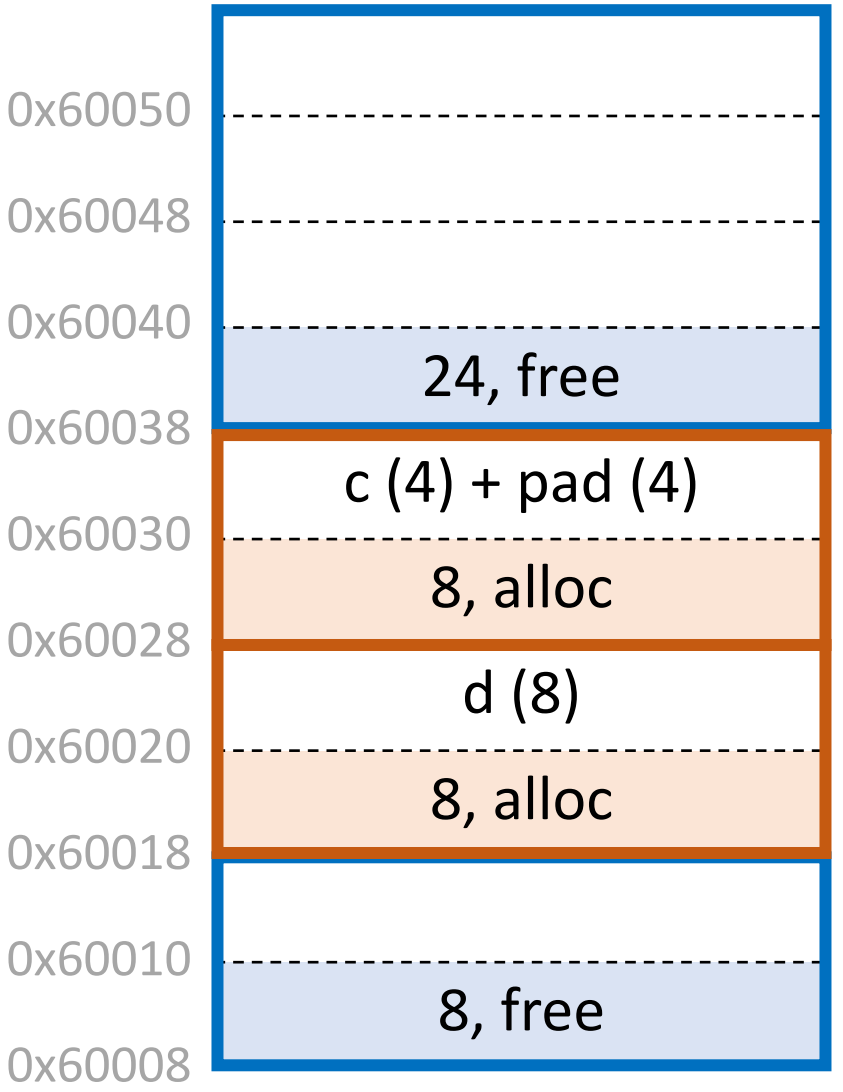
| | |
|---|---|
| d | 0x60020 |
| c | 0x60030 |
| b | 0x60020 |
| a | 0x60010 |



# Implicit Free List Allocator

```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```

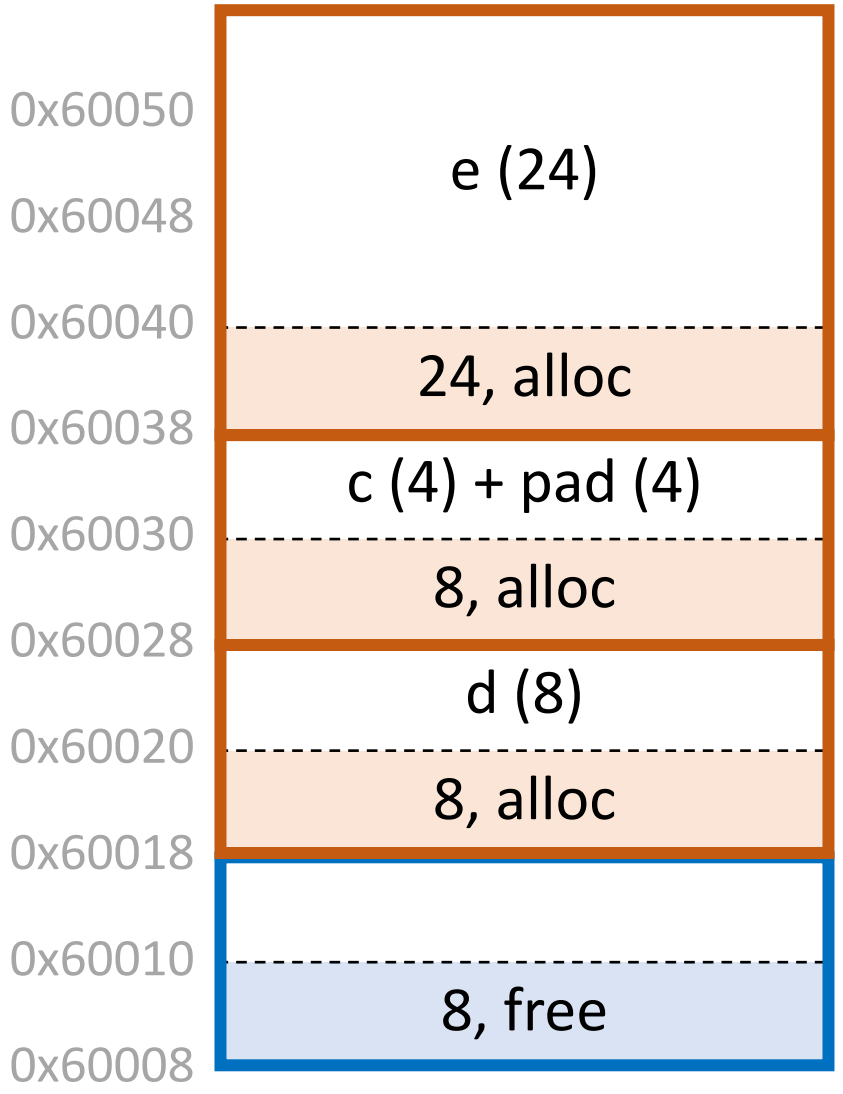
| | |
|---|---|
| d | 0x60020 |
| c | 0x60030 |
| b | 0x60020 |
| a | 0x60010 |



# Implicit Free List Allocator

```
1 void *a = malloc(4);
2 void *b = malloc(8);
3 void *c = malloc(4);
4 free(b);
5 void *d = malloc(8);
6 free(a);
7 void *e = malloc(24);
```

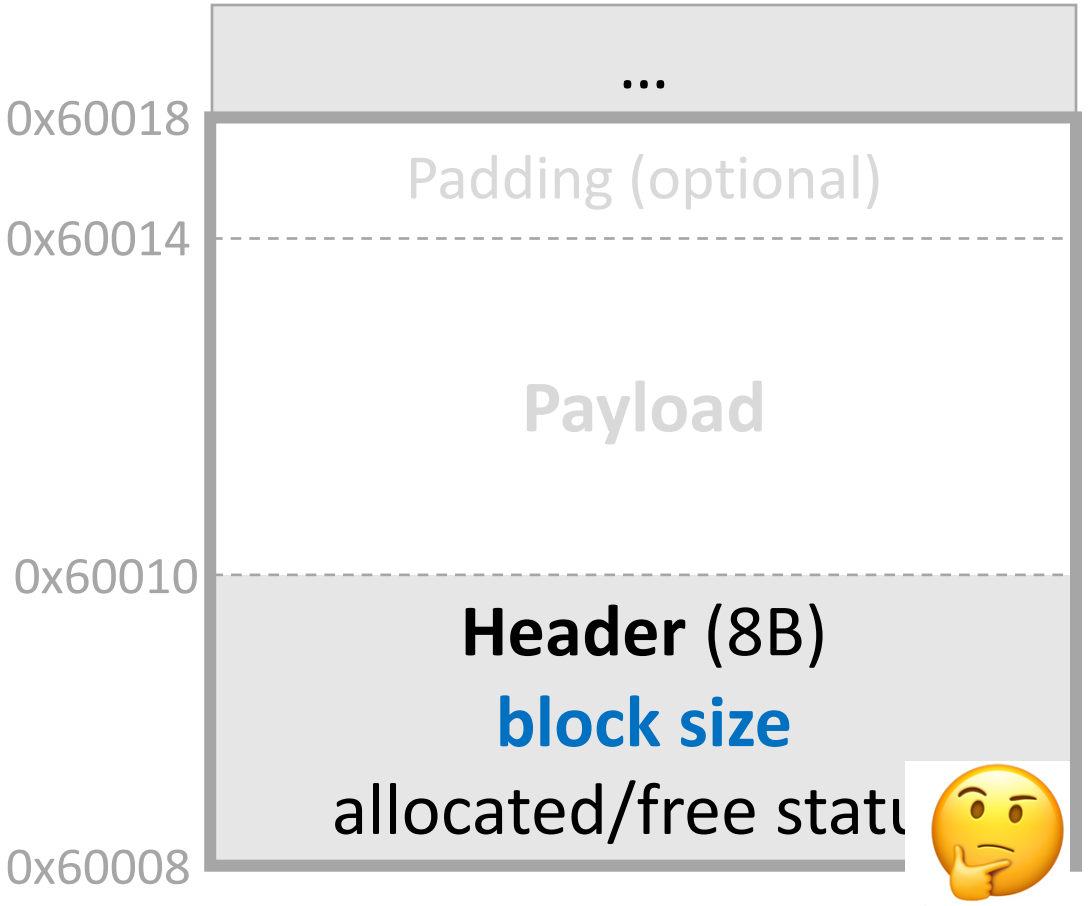
| | |
|---|---|
| e | 0x60040 |
| d | 0x60020 |
| c | 0x60030 |
| b | 0x60020 |
| a | 0x60010 |



# Implicit Free List: Header design

1. Should we store the **block size** as (A) payload size, or (B) header + payload size?

2. How can we store both a block size and a used/free status in 8B?



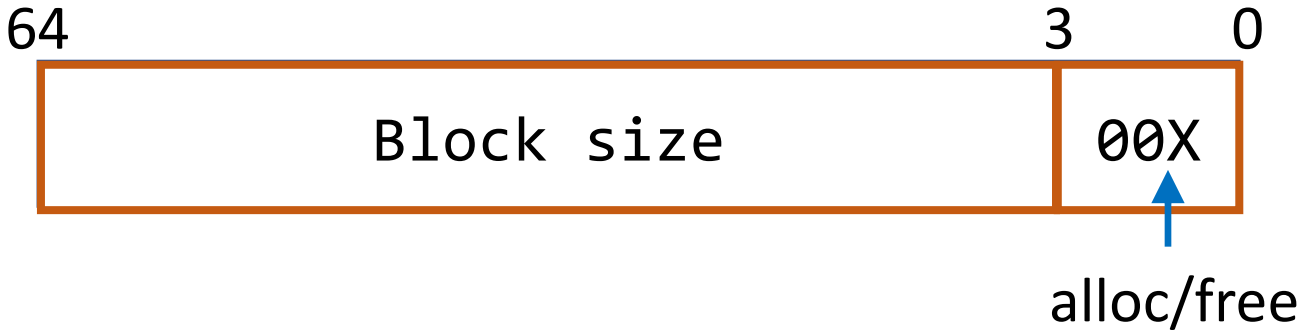
# Implicit Free List: Header design

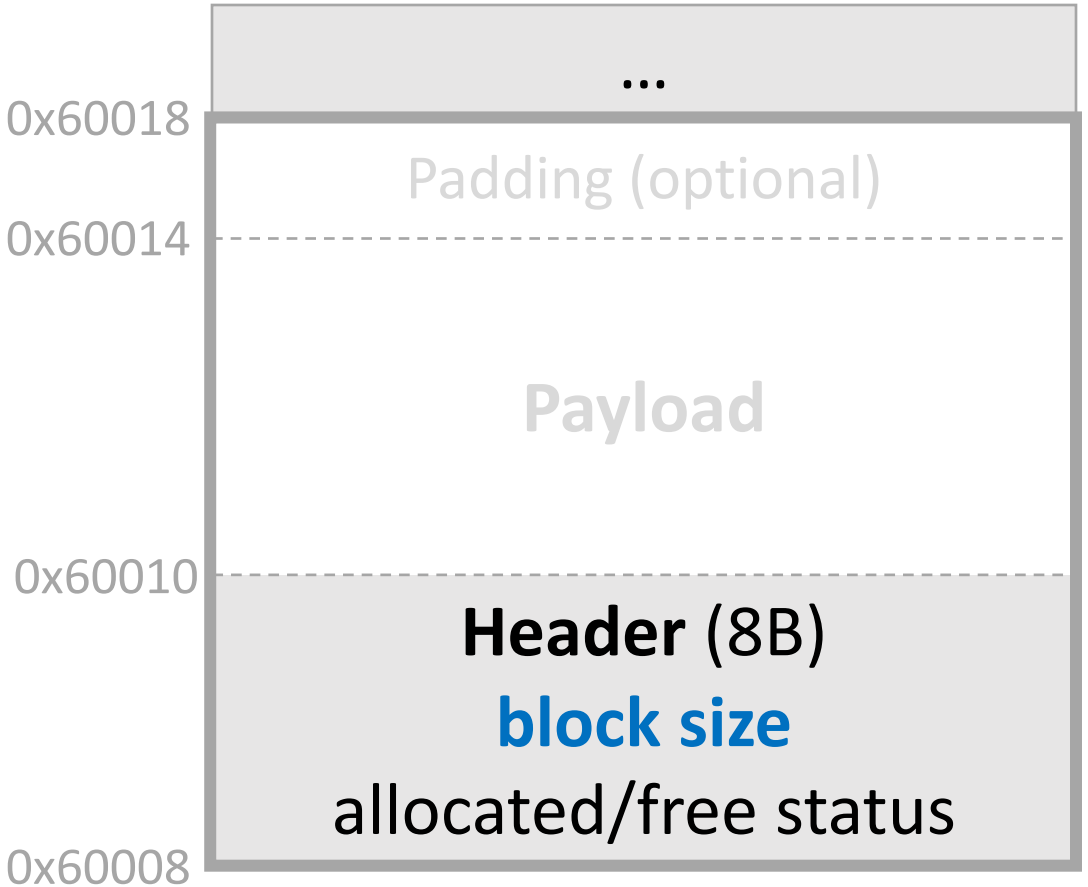
1. Should we store the **block size** as (A) payload size, or (B) header + payload size?

   Up to you! Your decision affects how you traverse the list (be careful of off-by-one)

2. How can we store both a block size and a used/free status in 8B?

   8-byte alignment → block sizes = multiples of 8!



# Designing `malloc()`

Suppose we use a **first-fit placement policy** to choose the free block:

For each allocation request:

- Start at the beginning of the heap
- Find the first available block that is free and fits the request size

1. Throughput: What is the worst-case runtime of first-fit placement?

   A. Constant
   B. Linear in number of allocated and free blocks
   C. Linear in number of free blocks
   D. Other

2. What are some alternatives to first-fit that might improve memory utilization or throughput?

# Designing `malloc()`

Suppose we use a **first-fit placement policy** to choose the free block:

For each allocation request:
- Start at the beginning of the heap
- Find the first available block that is free and fits the request size

1. Throughput: What is the worst-case runtime of first-fit placement?

   A. Constant
   B. Linear in number of allocated and free blocks
   C. Linear in number of free blocks
   D. Other

2. What are some alternatives to first-fit that might improve memory utilization or throughput?

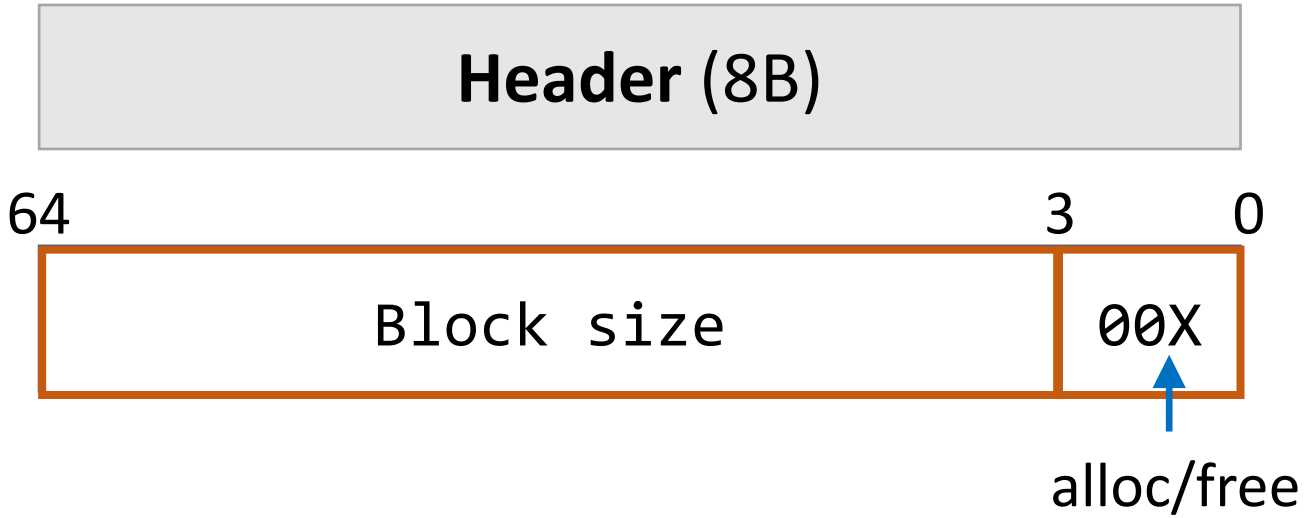
Up to you!

(we'll come back to this)

# Implicit Free List Summary

For **all blocks**,

- Have a header that stores size and status.
- Our list links *all* blocks, allocated (A) and free (F).




Keeping track of free blocks:

- **Improves memory utilization** (vs bump allocator)
- **Decreases throughput** (worst case allocation request has O(A + F) time)
- Increases design complexity ☺

Next up: Keep track of only free (F) blocks with **explicit free lists**. Worst-case O(F) time!

# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: Implicit Free List, **Explicit Free List**
- Policy decisions/performance
- Designing your own heap allocator

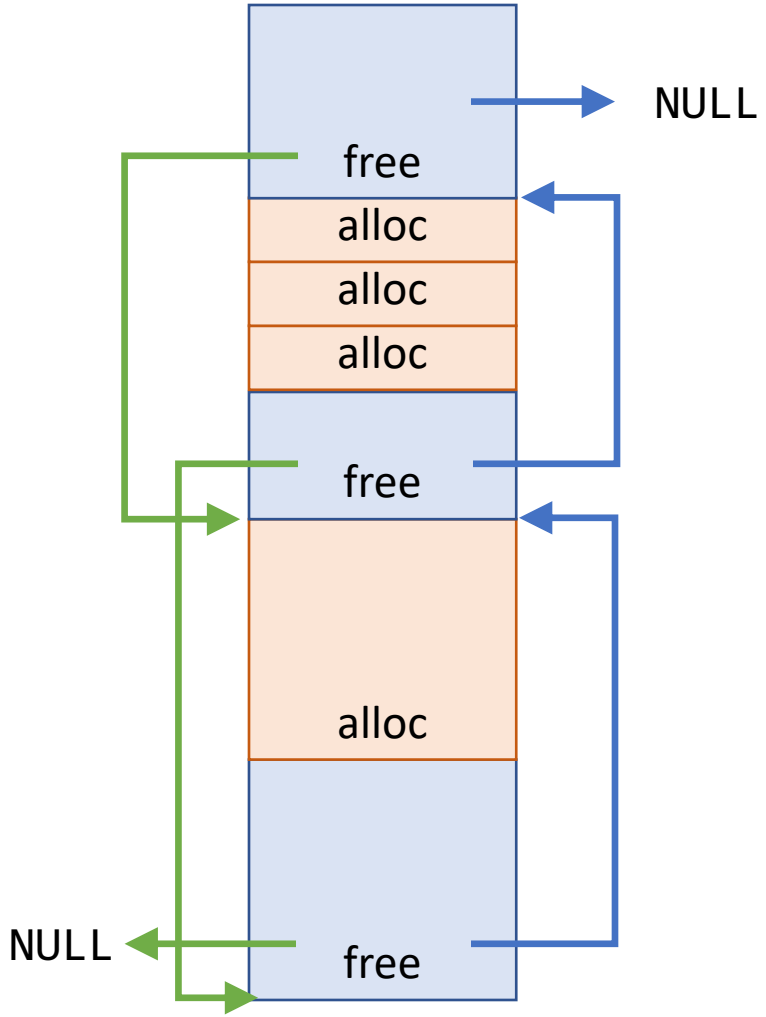
# Explicit Free List Design

Goal: A **doubly-linked list** of free blocks:

- Pointer to the previous free block
- Pointer to the next free block

Allocation run-time is now
linear in number of ***free*** blocks.

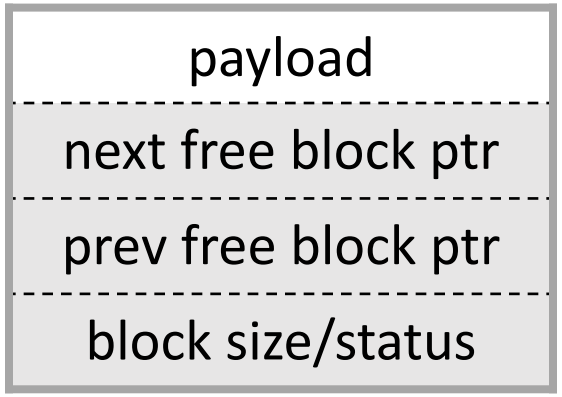
This **explicit** list of free blocks increases request throughput, with some costs (design and internal fragmentation)



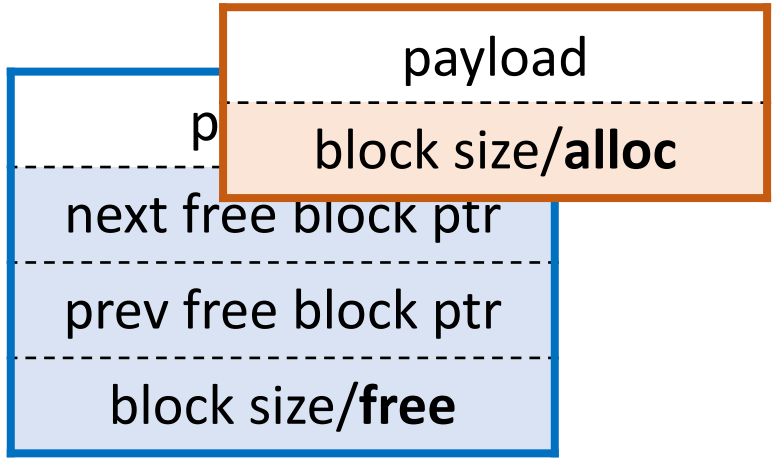
# Explicit Free List: Header Design

1. For each block, where should we include these pointers?

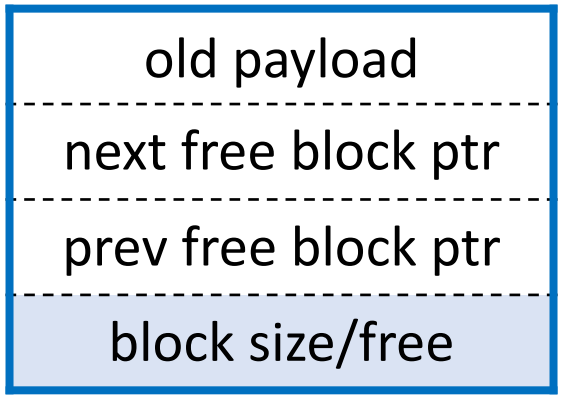
A. New header for all blocks




B. Separate headers for free/allocated blocks




C. Insert into free block payload




D. Other

2. If we choose option C, does this change our minimum payload size?

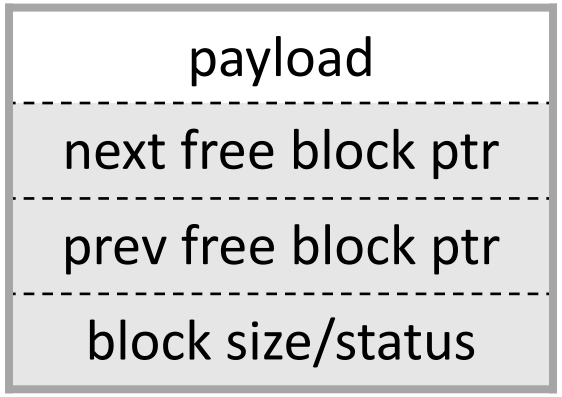
A. No
(keep 8B header + **8B** payload)

B. Yes
(change to 8B header + **16B** payload)

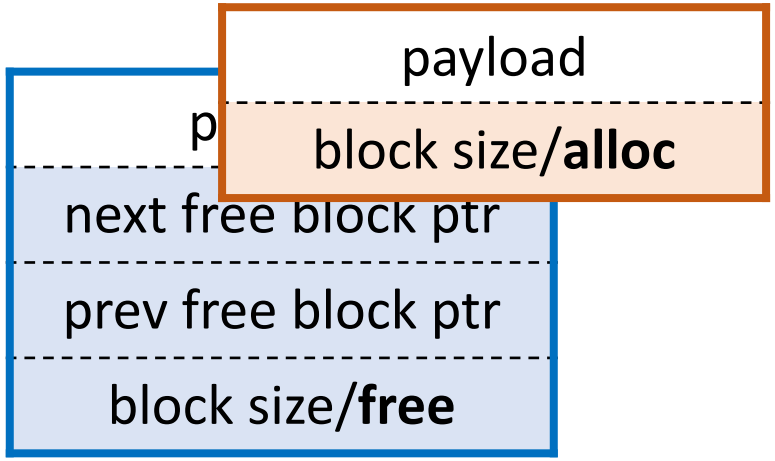
# Explicit Free List: Header Design

1. For each block, where should we include these pointers?

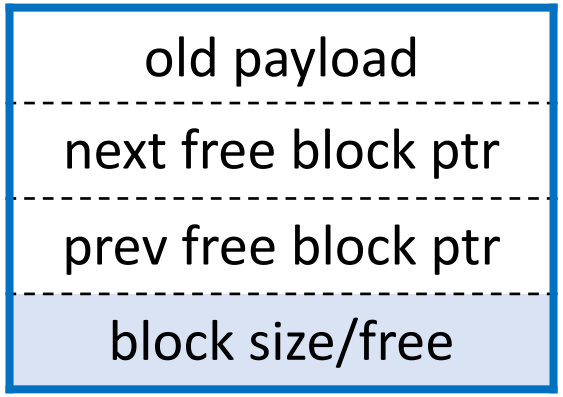
A. New header for all blocks




B. Separate headers for free/allocated blocks




C. Insert into free block payload




D. Other

2. If we choose option C, does this change our minimum payload size?

A. No
(keep 8B header + **8B** payload)

B. Yes
(change to 8B header + **16B** payload)

Up to you!

# Explicit Free List: List Design

1. `free` inserts a new block into the explicit free list. What will be the worst-case runtime of `free`?

2. How do you want to organize your explicit free list? (compare utilization/throughput)
   A. Address-order (each block's address is less than successor block's address)
   B. Last-in first-out (LIFO)/like a stack, where newly freed blocks are at the beginning of the list
   C. Other (e.g., by size, etc.)

Up to you!

# Explicit Free List: List Design

1. `free` inserts a new block into the explicit free list. What will be the worst-case runtime of `free`?

- Need a linear search through list to determine prev/next addresses
- Want constant time? Check out textbook's boundary tag/footer solution!

2. How do you want to organize your explicit free list? (compare utilization/throughput)

Up to you!

A. Address-order (each block's address is less than successor block's address)

Better memory util, Linear free

B. Last-in first-out (LIFO)/like a stack, where newly freed blocks are at the beginning of the list

Constant free (push recent block onto stack)

C. Other (e.g., by size, etc.)

(more at end of lecture)

# Implicit vs. Explicit: Summary

**Implicit Free List**

- 8B header for size + alloc/free status
- Allocation requests are worst-case linear in total number of blocks
- Implicitly address-order

**Explicit Free List**

- 8B header for size + alloc/free status
- Free block payloads store prev/next free block pointers
- Allocation requests are worst-case linear in number of free blocks
- Can choose block ordering
- Potentially more internal fragmentation (min block size larger)

In assignment 7:

- First build an implicit free list.
- Then, build your final, high-performing allocator using an explicit free list.

# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: Implicit Free List, Explicit Free List
- Policy decisions/performance
- Designing your own heap allocator

# Policy decisions

1. **Placement policy:** How to pick which free block from available options

2. **Splitting policy**: What to do with excess space when allocating

3. **Coalescing policy**: How to recycle a freed block?

Also: implementing an efficient `realloc`

Many of these policy decisions are Up to you!

Up to you!

# Placement policy

How do we choose a free block to use for an allocation request, e.g., `malloc()`?

- **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.

Easier to implement

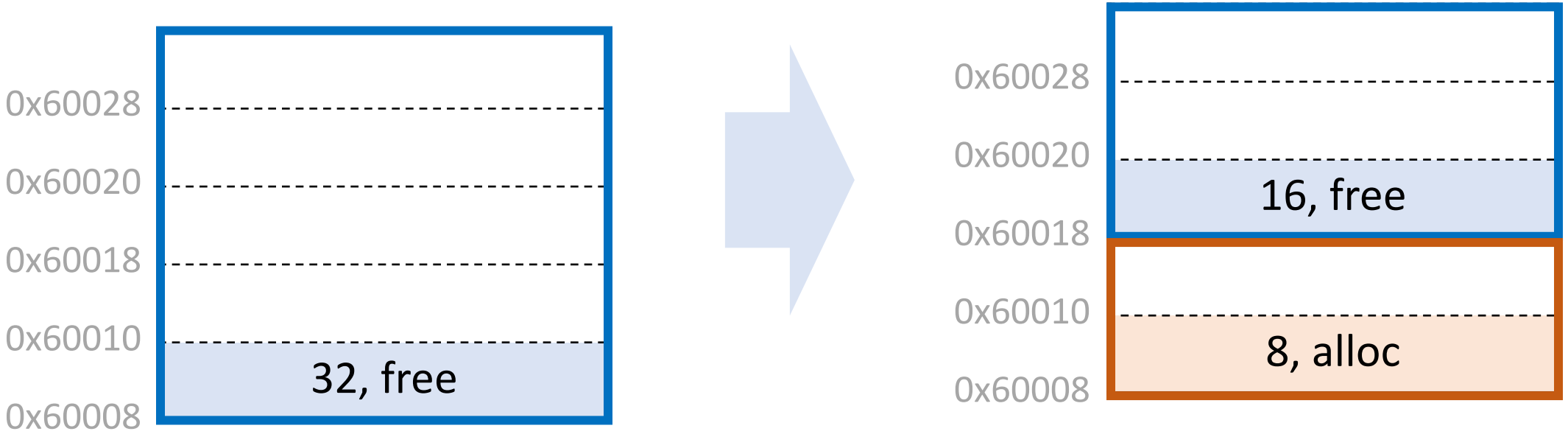
What are the pros/cons for each approach? Throughput vs. Utilization?

More on B&O p. 849

# Splitting

A reasonable allocation request splits a free block into an allocated block and a free block with remaining space:

```
void *ptr = malloc(8);
```



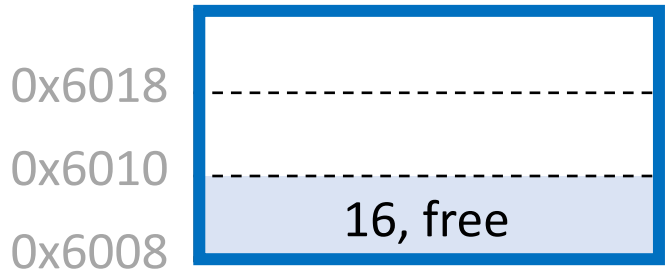

Not all cases are this friendly…

Up to you!

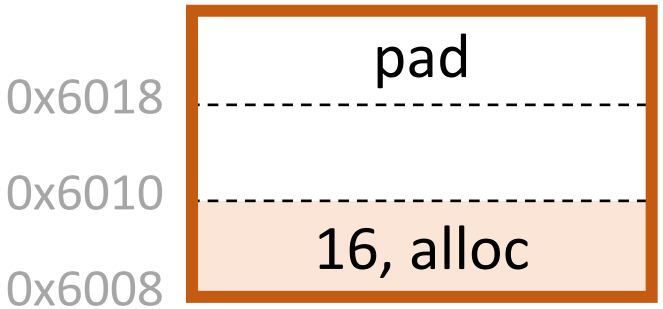
# Splitting Policy

**Problem**: Some blocks are just too small.

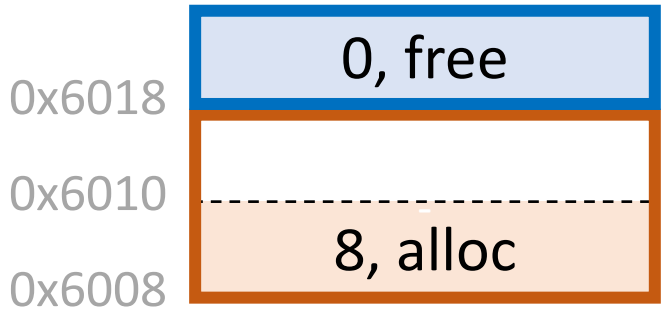
```
void *ptr = malloc(8);
```




A.




B.



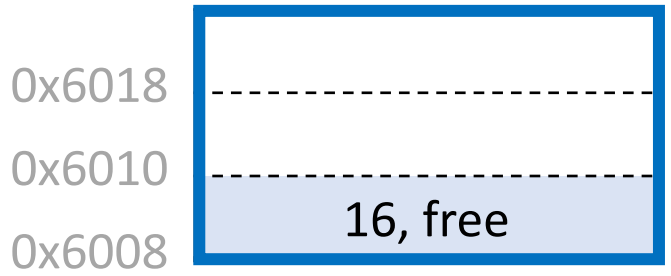

Which splitting policy would you take here? What are the tradeoffs?

Up to you!

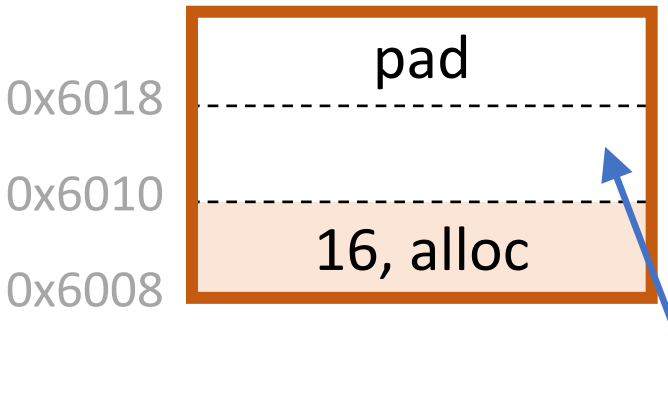
# Splitting Policy

**Problem**: Some blocks are just too small.

```
void *ptr = malloc(8);
```




A.



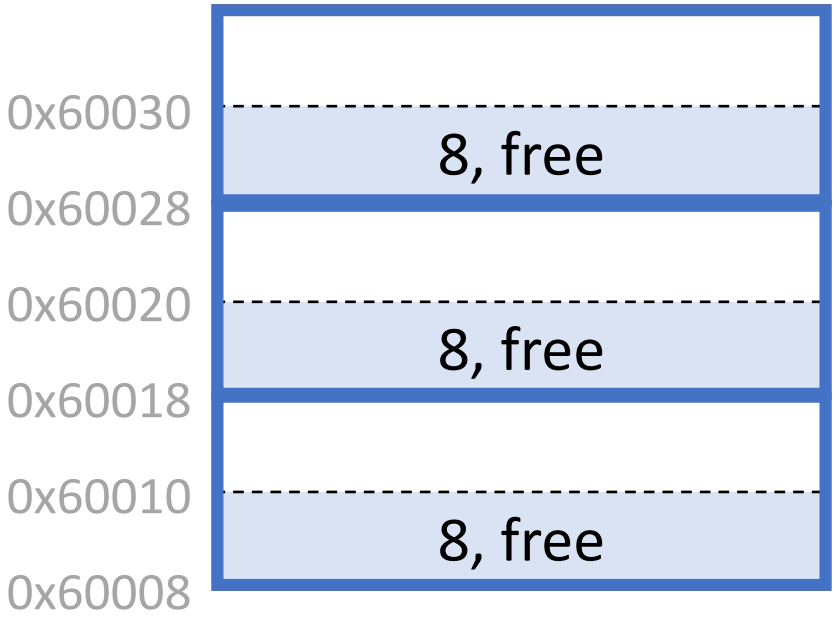

**Internal** fragmentation:
unused bytes b/c of padding

B.


**External** fragmentation:
unused free blocks

# Coalescing

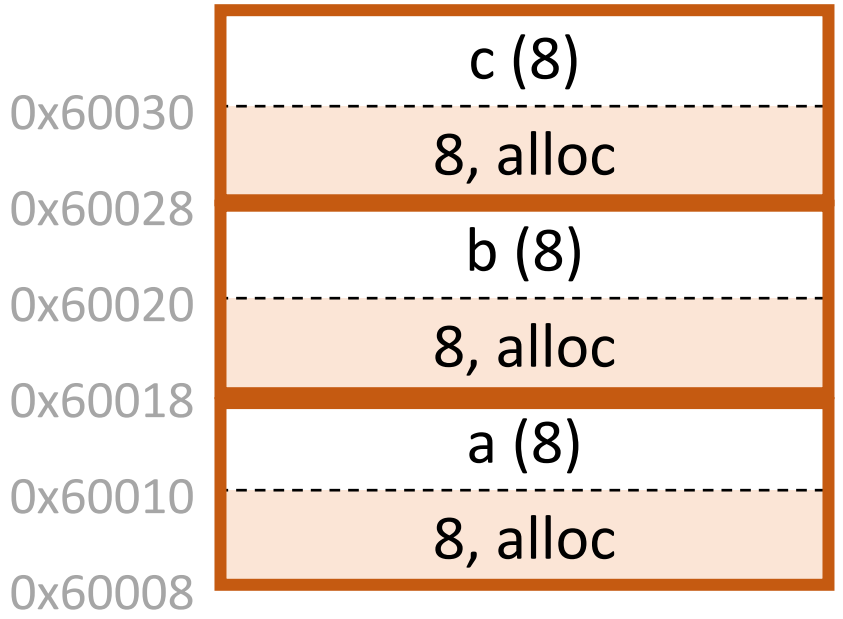
```
1 void *a = malloc(8);
2 void *b = malloc(8);
3 void *c = malloc(8);
4 free(a);
5 free(c);
6 free(b);
7 void *d = malloc(24); // NULL
```




- The last request will fail because we have enough memory space, but it is **fragmented** into free blocks sized from earlier requests!
- We need to **coalesce** (i.e., combine adjacent) free blocks to avoid this external fragmentation.

# Coalescing

```
1 void *a = malloc(8);
2 void *b = malloc(8);
3 void *c = malloc(8);
4 free(a);
5 free(c);
6 free(b);
7 void *d = malloc(24); // NULL
```




1. When can you coalesce adjacent free blocks?

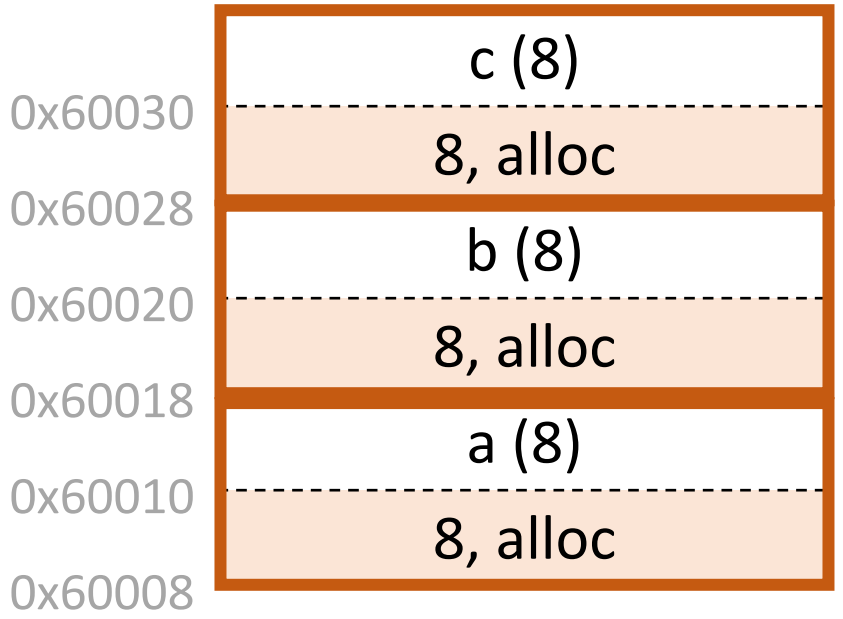
A. Line 4
C. Line 6
B. Line 5
D. Line 7

2. Line 6: When freeing block 0x60020, which blocks would you coalesce?

A. Previous block (header 0x60008)
B. Next block (header 0x60028)
C. Both A and B

# Coalescing

```
1 void *a = malloc(8);
2 void *b = malloc(8);
3 void *c = malloc(8);
4 free(a);
5 free(c);
6 free(b);
7 void *d = malloc(24); // NULL
```




1. When can you coalesce adjacent free blocks?

~~A.~~ Line 4 C. Line 6 Immediate

~~B.~~ Line 5 D. Line 7 Deferred

2. Line 6: When freeing block 0x60020, which blocks would you coalesce?

A. Previous block (header 0x60008)
B. Next block (header 0x60028)
C. Both A and B

# How to make operations fast?

`malloc` is generally about search

- More quickly identify which blocks are appropriate to consider
- The fewer blocks examined, the better. Be less picky about which block to use
- Placement policy/Splitting policy

`free` is mostly about update

- Ideal data structures modified in constant-time
- Postpone work until clearly needed (immediate vs. deferred coalesce)

`realloc`

- Move to a new block: malloc/free/copy payload
- Big win if avoid copy payload data and just resize **in-place**

# realloc

```
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.

Realloc can **grow** or **shrink** the data size.

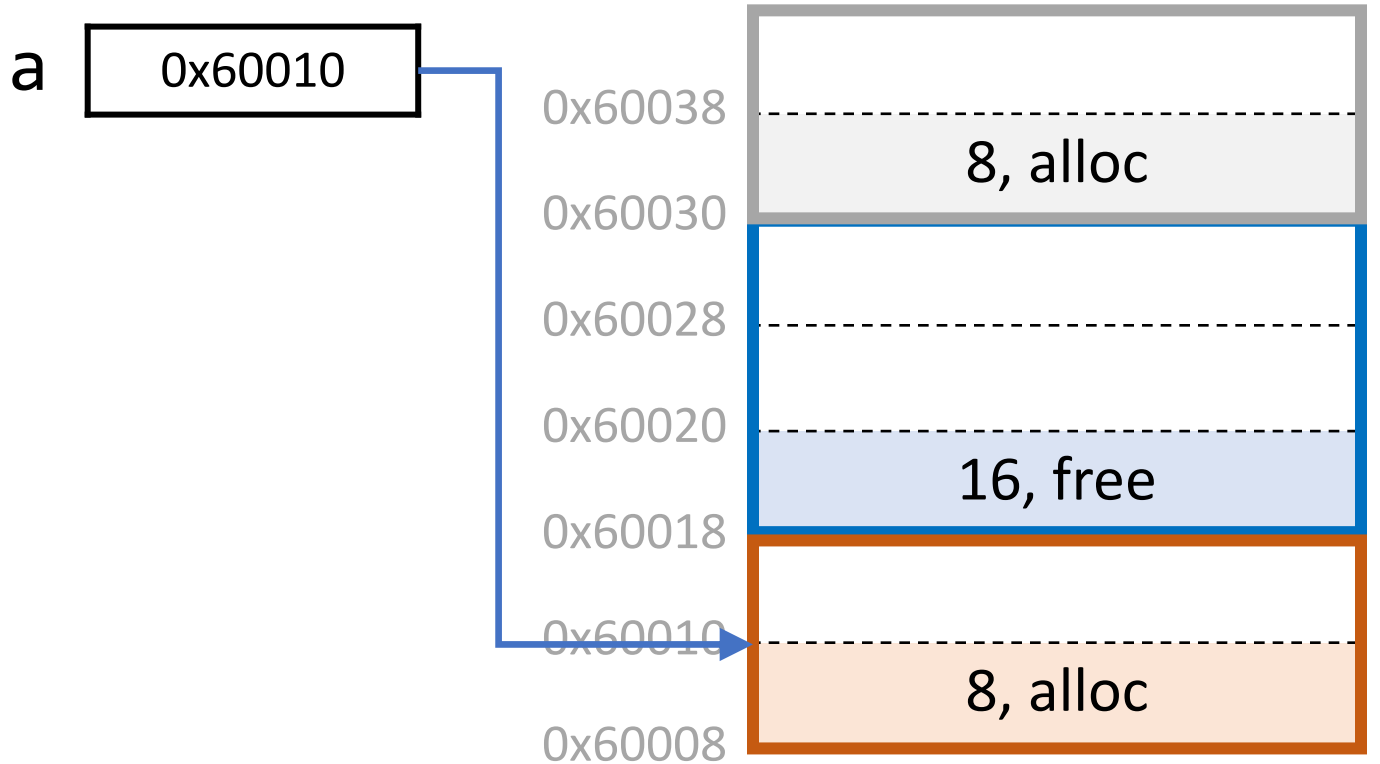
```
void *realloc(void *ptr, size_t size);
```

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.

# When can we `realloc` in-place?

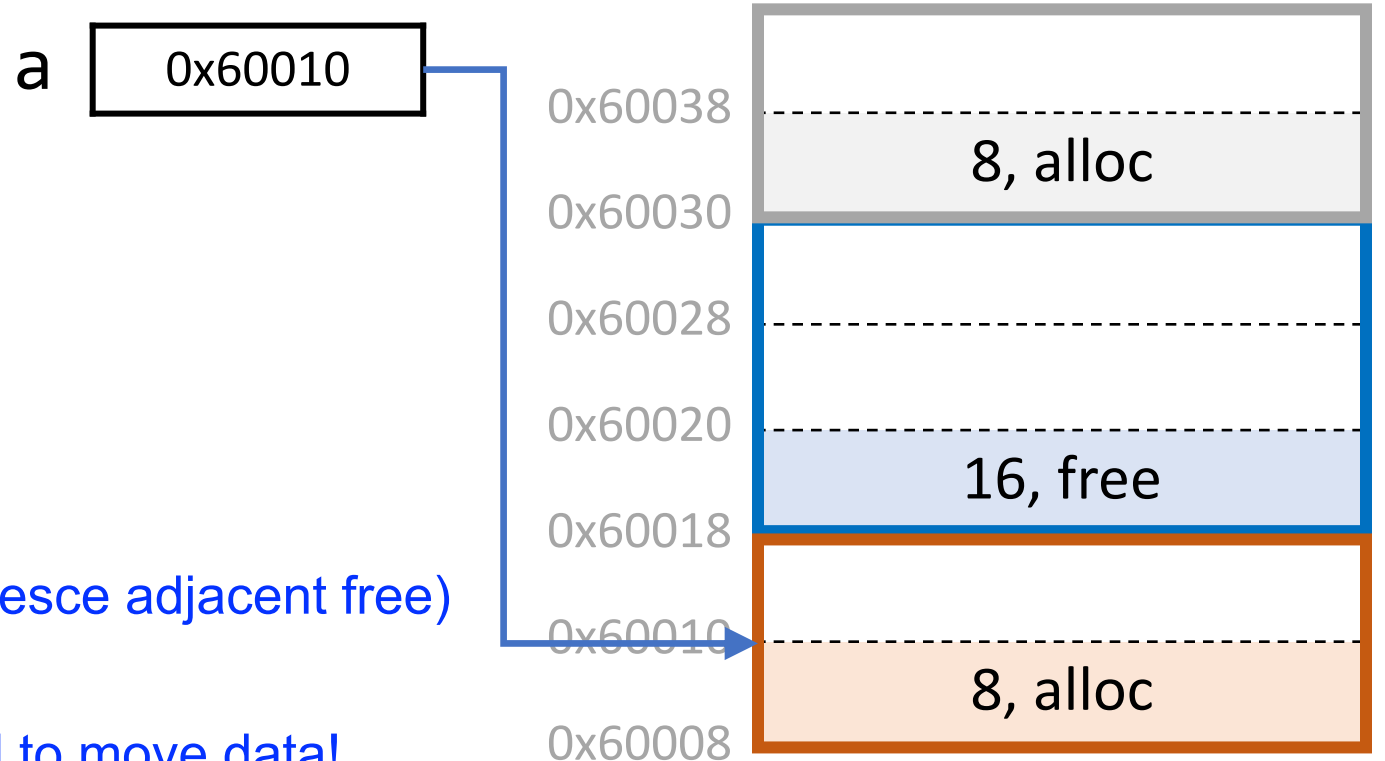
```
void *a = malloc(6);
```

1. `realloc(a, 8);`
2. `realloc(a, 2);`
3. `realloc(a, 32);`
4. `realloc(a, 64);`




# When can we `realloc` in-place?

```
void *a = malloc(6);
```



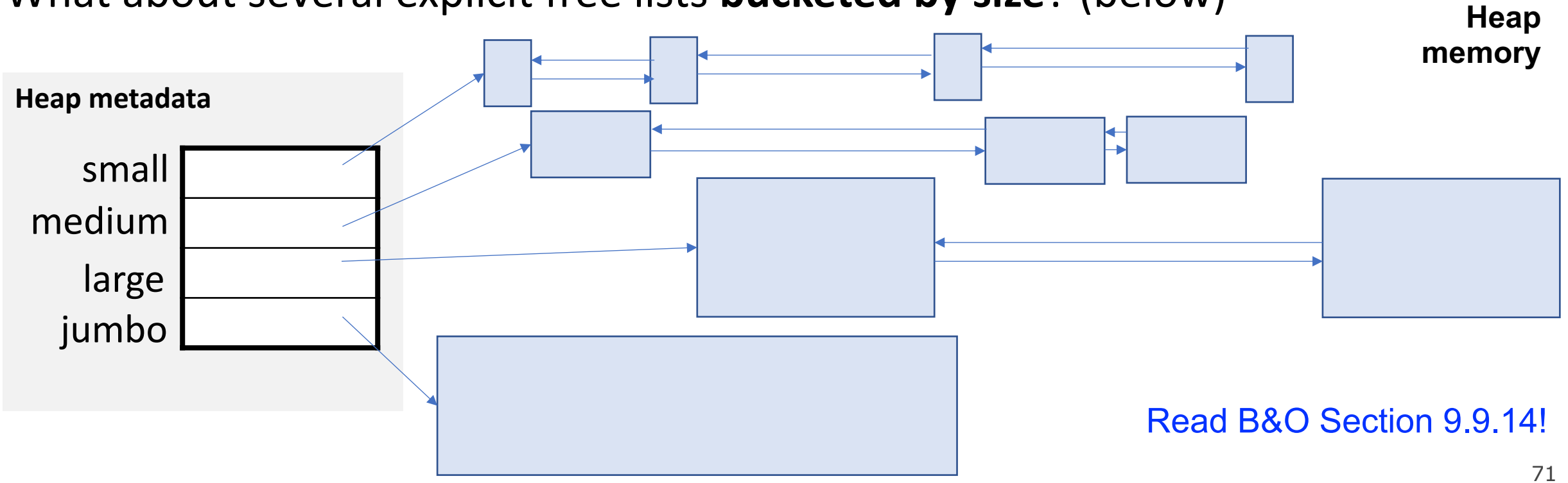

1. `realloc(a, 8);` Yes
2. `realloc(a, 2);` Yes
3. `realloc(a, 32);` Yes (coalesce adjacent free)
4. `realloc(a, 64);` No! need to move data!

Thoughts for you:

- What is necessary to allow resize in-place? Is it worth it to anticipate that?
- How prominent is realloc in the mix of operations?

# Going beyond: Explicit list w/size buckets

- Explicit lists are much faster than implicit lists.
- However, a first-fit placement policy is still linear in total # of free blocks.
- What about an explicit free list **sorted by size** (e.g., as a tree)?
- What about several explicit free lists **bucketed by size**? (below)




Read B&O Section 9.9.14!

# Plan For Today

- What is a heap allocator?
- Heap allocator requirements and goals
- Simple example: Bump Allocator
- **Break:** Announcements
- Data Structures: Implicit Free List, Explicit Free List
- Policy decisions/performance
- Designing your own heap allocator

# Tips for assign7

**Read B&O textbook.**

- Offers some starting tips for implementing your heap allocators.
- Make sure to cite any design ideas you discover.

**Honor Code/collaboration**

- All non-textbook code is off-limits.
- Please do not discuss discuss code-level specifics with others.
- Your code should be designed, written, and debugged by you independently.

**Helper hours**

- We will provide good debugging techniques and strategies!
- Come and discuss design tradeoffs!

# Assignment 7: Implicit Allocator

- **Must have** headers that track block information (size, status in-use or free). You must use an 8 byte header size, storing the status using the free bits.
  (note: this is larger than the 4 byte headers specified in the book)
- **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 free list (i.e. traverses block-by-block). Placement policy up to you.

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

# Assignment 7: Explicit Allocator

- **Must have** headers that track block information like in implicit (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 (*immediate coalescing*)
- **Must** do in-place realloc whenever possible.