# Dynamic Memory Management

# Goals of this Lecture

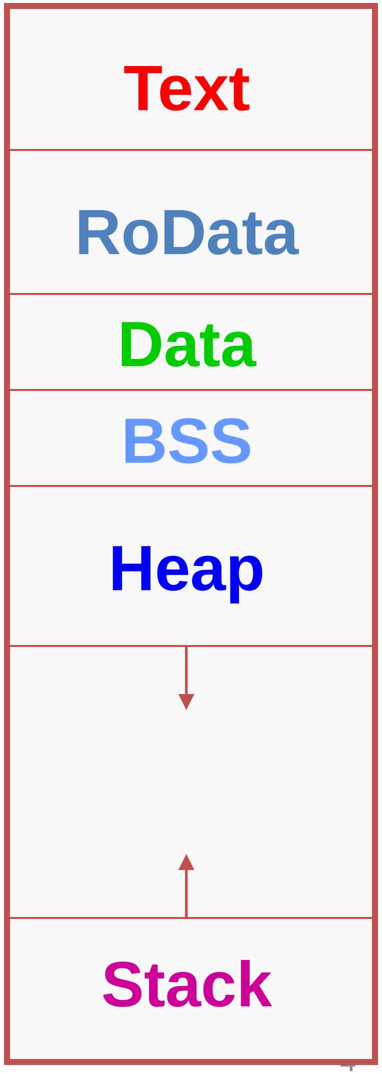
- Help you learn about:
  - Dynamic memory management techniques
    - Garbage collection by the run-time system (Java)
    - Manual deallocation by the programmer (C, C++)
  - Design decisions for the "K&R" heap manager implementation
    - Circular linked-list of free blocks with a "first fit" allocation
    - Coalescing of adjacent blocks to create larger blocks

# Part 1:
# What do `malloc()` and `free()` do?

# Memory Layout: Heap

```c
char* string = "hello";
int iSize;

char* f()
{
    char* p;
    scanf("%d", &iSize);
    p = malloc(iSize);
    return p;
}
```




Needed when required memory size is not known before the program runs

# Allocating & Deallocating Memory

- Dynamically *allocating* memory
  - Programmer explicitly requests space in memory
  - Space is allocated dynamically on the heap
  - E.g., using "malloc" in C, and "new" in Java/C++
- Dynamically *deallocating* memory
  - Must reclaim or recycle memory that is never used again
  - To avoid (eventually) running out of memory
- "Garbage"
  - Allocated block in heap that will not be accessed again
  - Can be reclaimed for later use by the program

# Option #1: Garbage Collection

- *Run-time system* does garbage collection (Java)
  - Automatically determines objects that can't be accessed
  - And then reclaims the resources used by these objects

```
Object x = new Foo();
Object y = new Bar();
x = new Quux();

if (x.check_something()) {
  x.do_something(y);
}
System.exit(0);
```

Object **Foo()** is never used again!

# Challenges of Garbage Collection

- Detecting the garbage is not always easy
  - "if (complex_function(y))  x = Quux();"
  - Run-time system cannot collect *all* of the garbage
- Detecting the garbage introduces overhead
  - Keeping track of references to objects (e.g., counter)
  - Scanning through accessible objects to identify garbage
  - Sometimes walking through a large amount of memory
- Cleaning the garbage leads to bursty delays
  - E.g., periodic scans of the objects to hunt for garbage
  - Leading to unpredictable "freeze" of the running program
  - Very problematic for real-time applications
  - … though good run-time systems avoid long freezes

# Option #2: Manual Deallocation

- *Programmer* deallocates the memory (*C* and *C++*)
  - Manually determines which objects can't be accessed
  - And then explicitly returns the resources to the heap
  - E.g., using "free" in *C* or "delete" in *C++*
- Advantages
  - Lower overhead
  - No unexpected "pauses"
  - More efficient use of memory
- Disadvantages
  - More complex for the programmer
  - Subtle memory-related bugs
  - Security vulnerabilities in the (buggy) code

# Manual Deallocation Can Lead to Bugs

- Dangling pointers
  - Programmer frees a region of memory
  - … but still has a pointer to it
  - Dereferencing pointer reads or writes *nonsense values*

```
int main(void) {
     char *p;
     p = malloc(10);
     …
     free(p);
     …
     putchar(*p);
}
```

May print nonsense character.

# Manual Deallocation Can Lead to Bugs

- Memory leak
  - Programmer neglects to free unused region of memory
  - So, the space can never be allocated again
  - Eventually may consume all of the available memory

```
void f(void) {
    char *s;
    s = malloc(50);
    return;
}

int main(void) {
    while (1) f();
    return 0;
}
```

Eventually, malloc() returns NULL

# Manual Deallocation Can Lead to Bugs

- Double free
  - Programmer mistakenly frees a region more than once
  - Leading to corruption of the heap data structure
  - … or premature destruction of a *different* object

```
int main(void) {
    char *p, *q;
    p = malloc(10);
    …
    free(p);
    q = malloc(10);
    free(p);
    …
}
```

Might free the space allocated to **q**!

# `malloc()` and `free()` Challenges

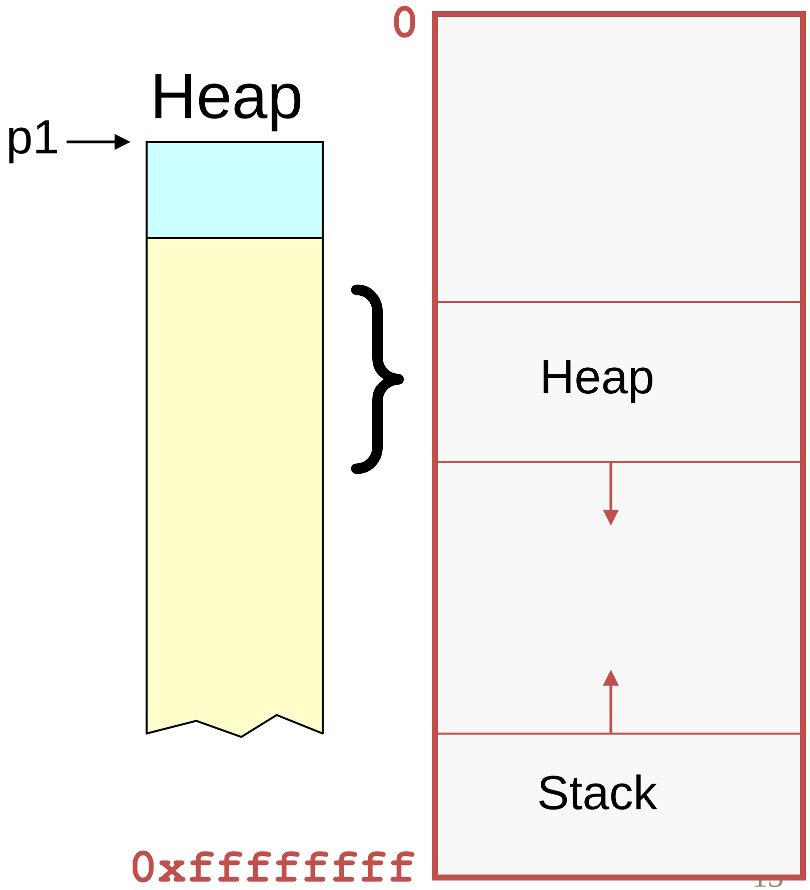
- **`malloc()`** may ask for arbitrary number of bytes
- Memory may be allocated & freed in different order
- Cannot reorder requests to improve performance

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

# Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

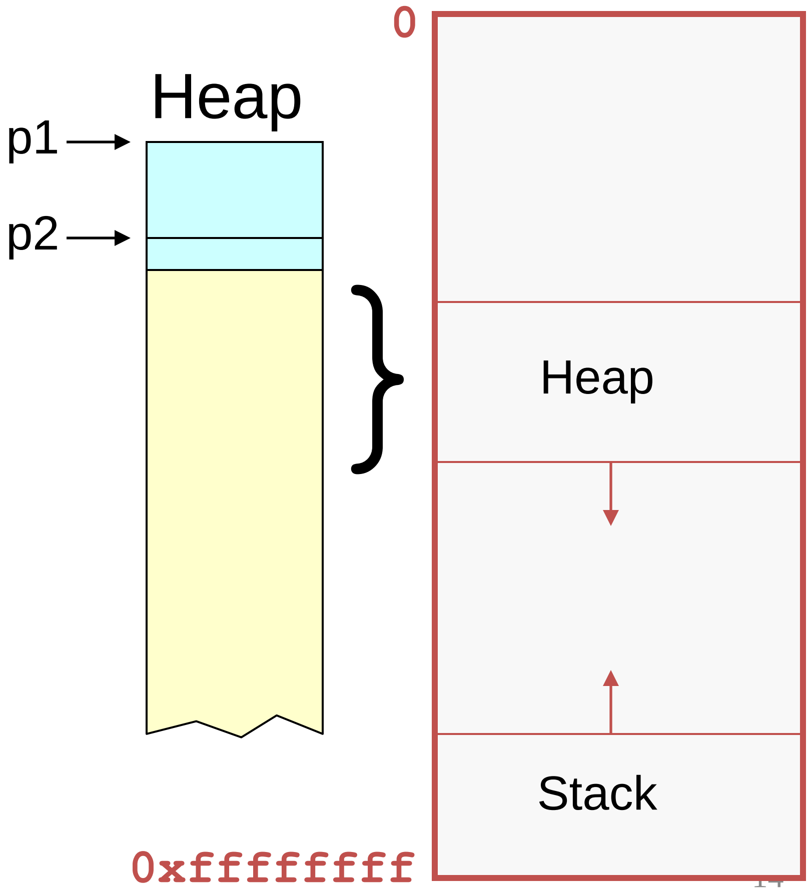
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

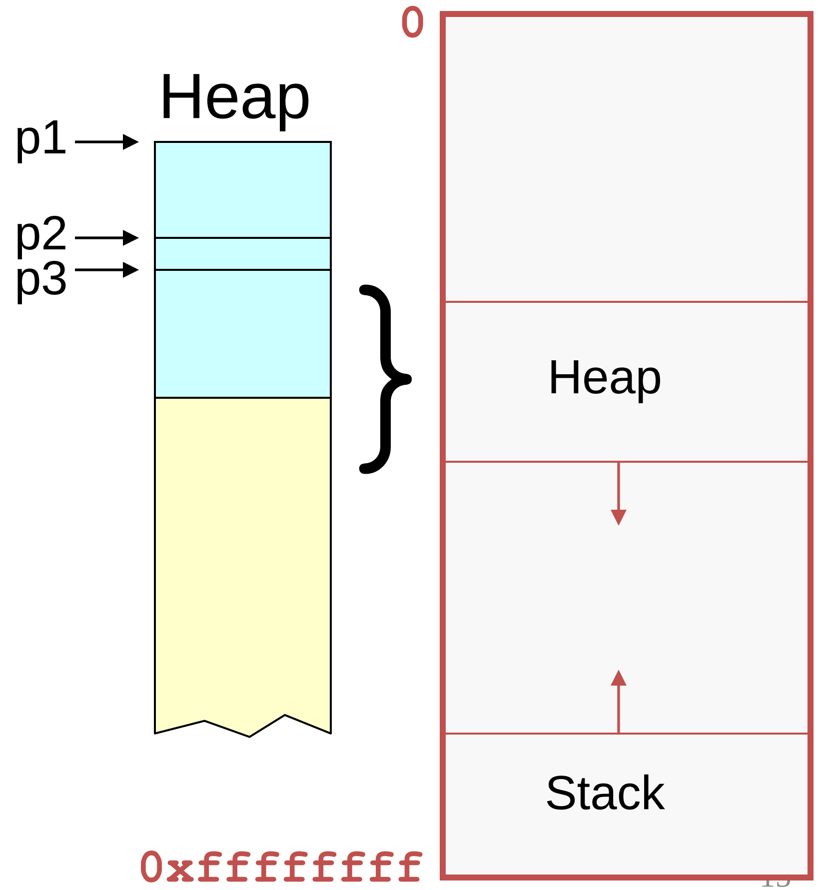
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

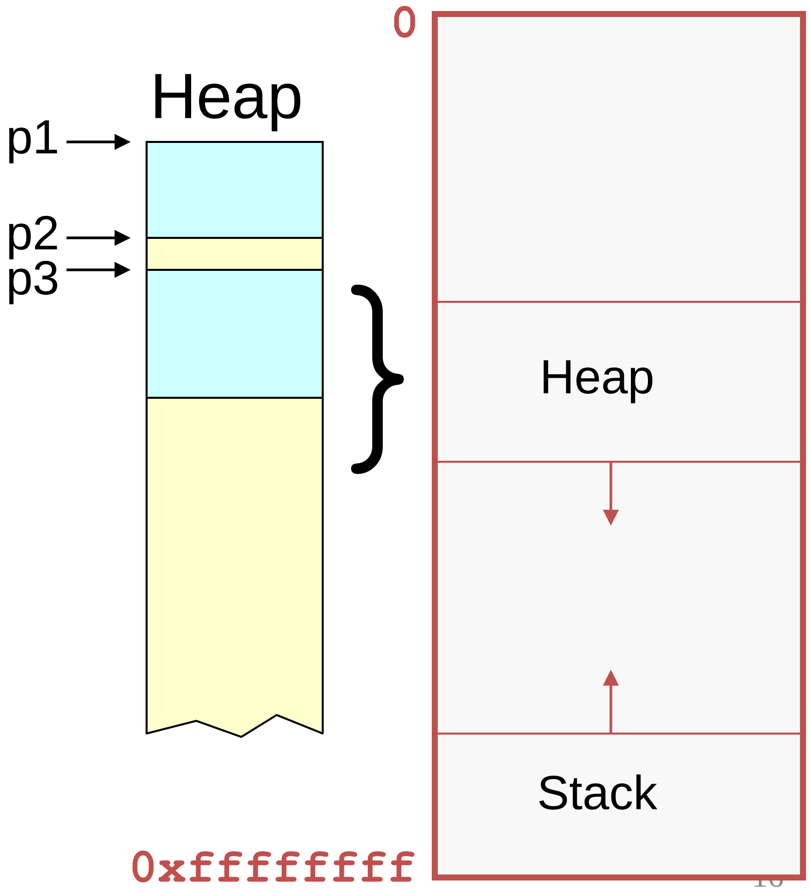
```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

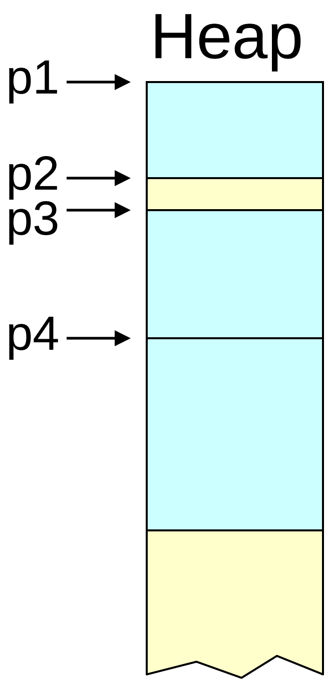
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

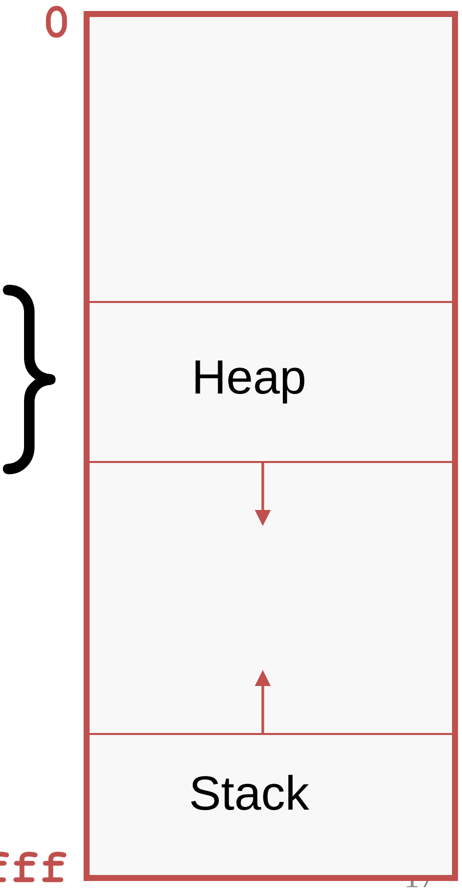


# Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

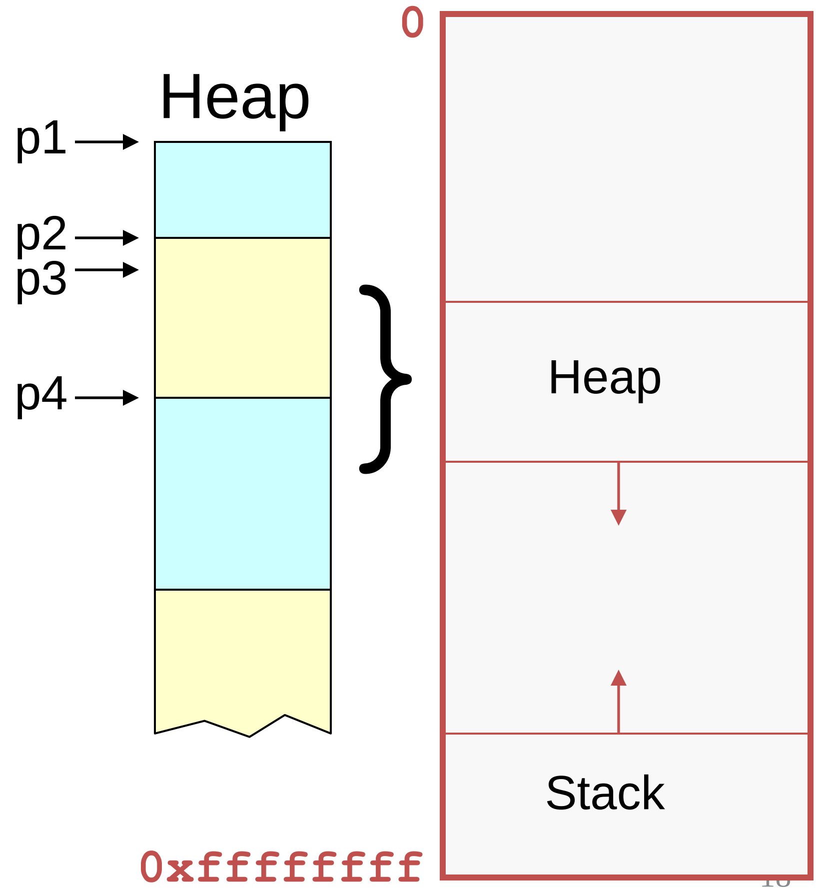


# Heap: Dynamic Memory

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

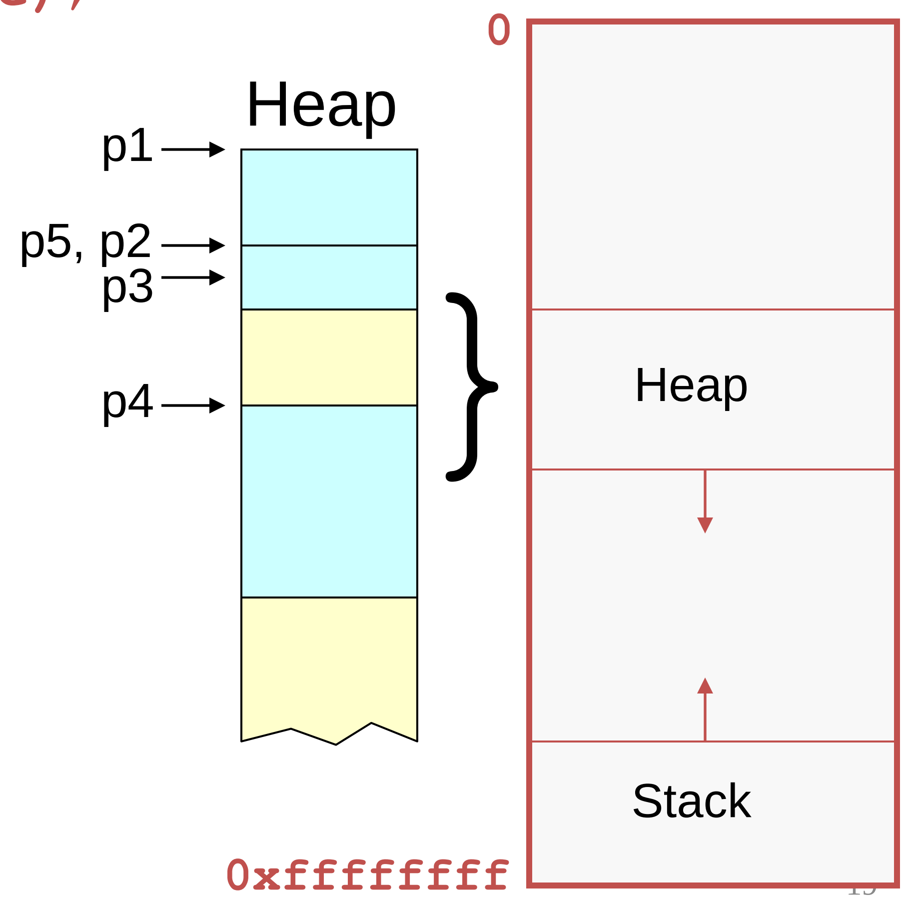
```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

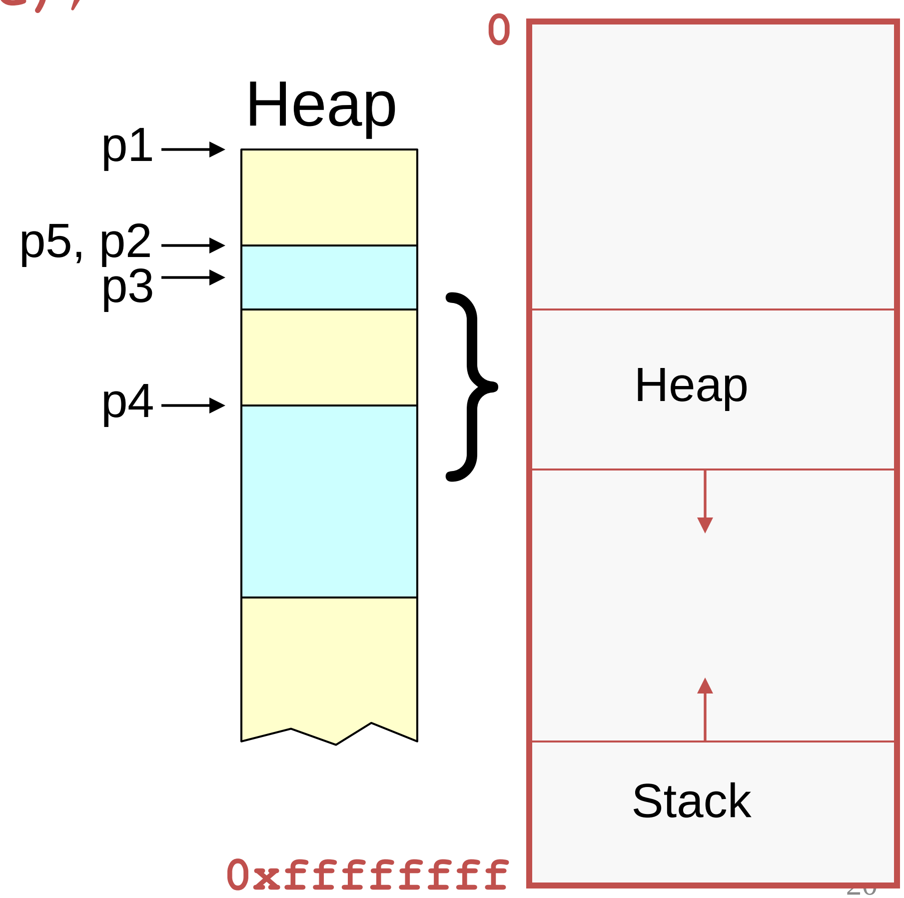
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

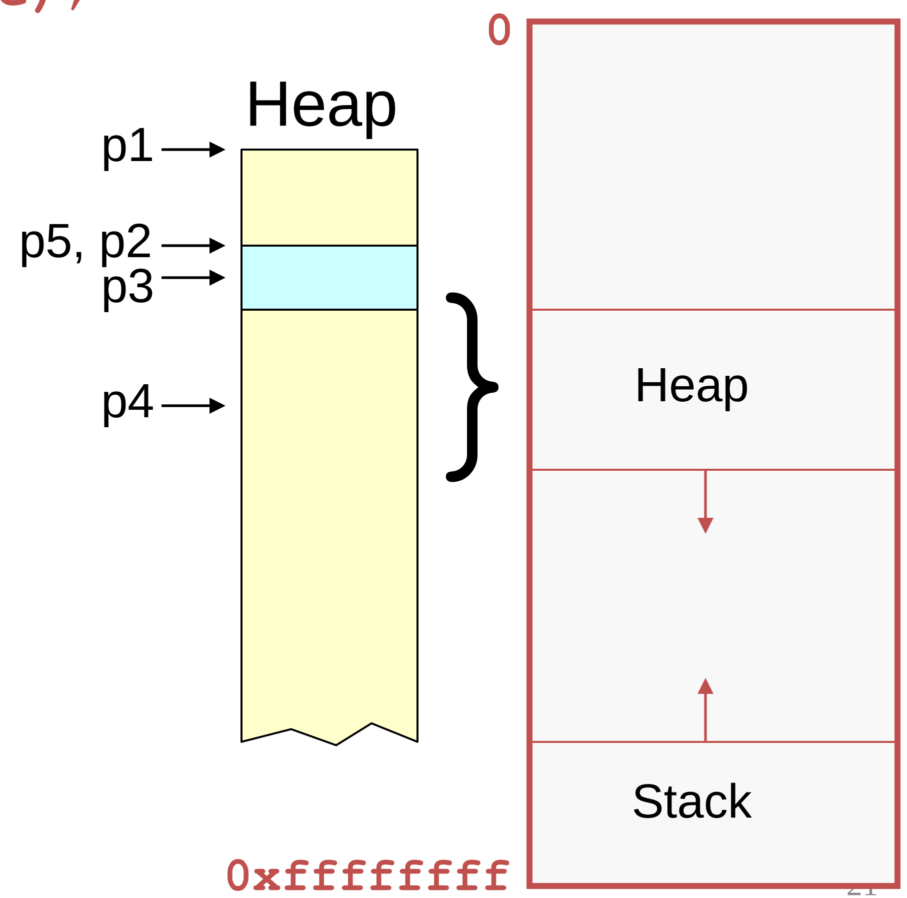
```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

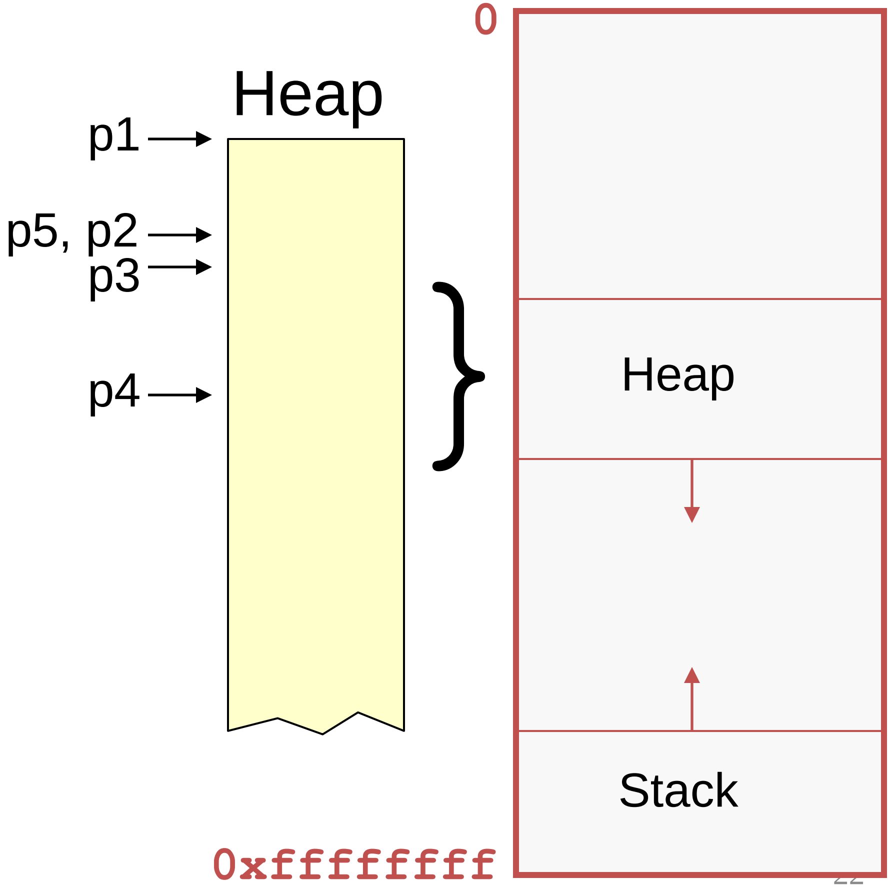
```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



# Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

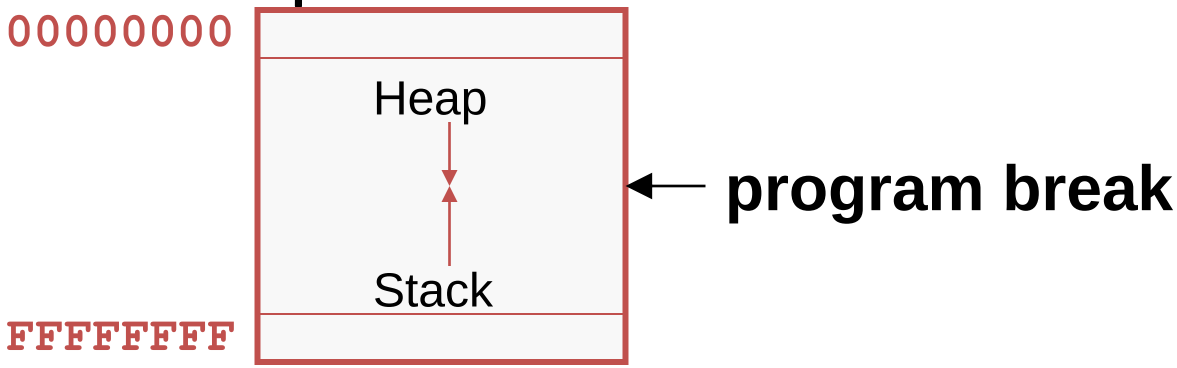


# Part 2:

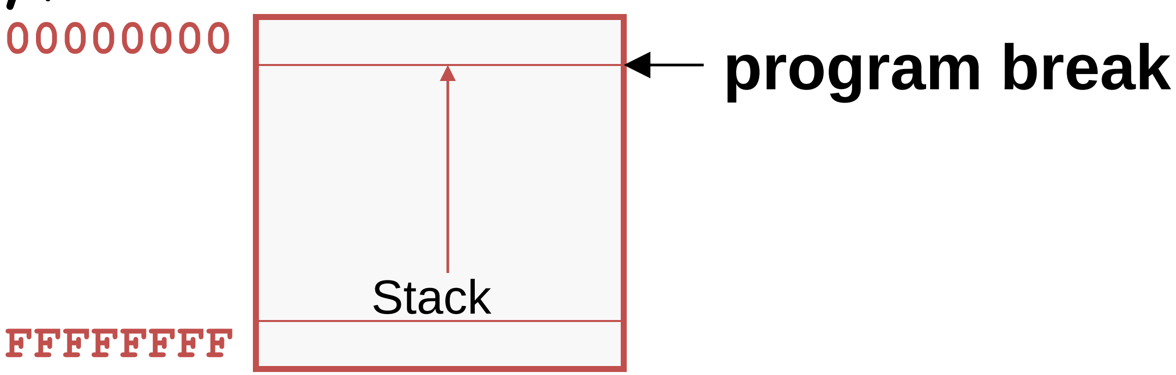
How do `malloc()` and `free()` work?

# The Program Break

The **program break** marks the boundary between heap and stack




Initially, stack has maximum size



# Acquiring Heap Memory

Q: How does `malloc()` acquire heap memory?

*A*: Moves the program break downward via `sbrk()` or `brk()` system call

**`void *sbrk(intptr_t increment);`**

- Increment the program break by the specified amount. Calling the function with an increment of 0 returns the current location of the program break. Return the ptr to the previous program break if successful and -1 otherwise.

**`int brk(void *newBreak);`**

- Move the program break to the specified address. Return 0 if successful and -1 otherwise.

# Using Heap Memory

Q: Having acquired heap memory, how do **`malloc()`** and **`free()`** manipulate it?

A: Topic of much research; an introduction…

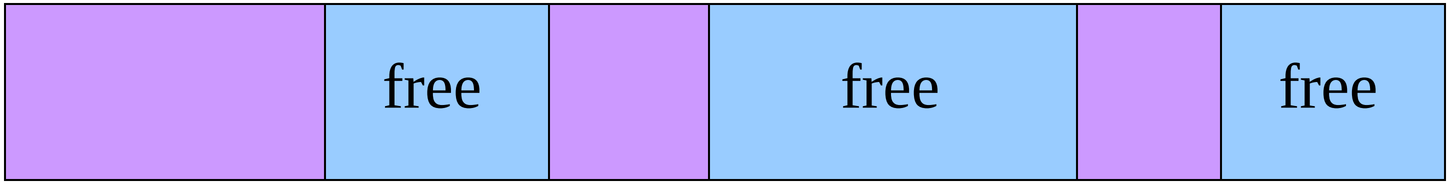
# Goals for `malloc()` and `free()`

- Maximizing throughput
  - Maximize number of requests completed per unit time
  - Need both **`malloc()`** and **`free()`** to be fast
- Maximizing memory utilization
  - Minimize the amount of wasted memory
  - Need to minimize size of data structures

- Strawman #1: **`free()`** does nothing
  - Good throughput, but poor memory utilization
- Strawman #2: **`malloc()`** finds the "best fit"
  - Good memory utilization, but poor throughput

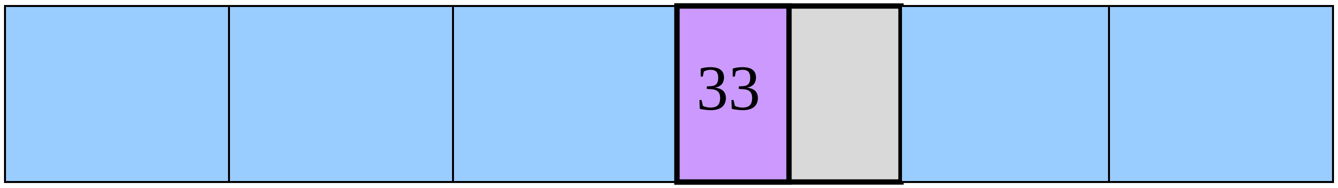
# Keeping Track of Free Blocks

- Maintain a list of free blocks of memory
  - Allocate memory from one of the blocks in the free list
  - Deallocate memory by returning the block to the free list
  - When necessary, call **`sbrk()`** to ask OS for additional memory, and create a new large block
- Design questions
  - How to keep track of the free blocks in memory?
  - How to choose an appropriate free block to allocate?
  - What to do with the left-over space in a free block?
  - What to do with a block that has just been freed?

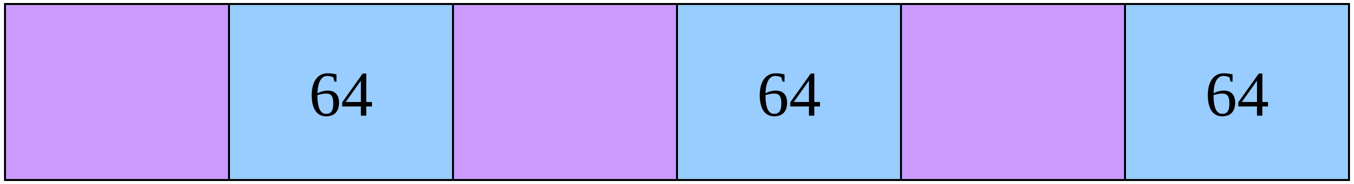


# Need to Minimize Fragmentation

- Internal fragmentation
  - Allocated block is larger than `malloc()` requested
  - E.g., `malloc()` imposes a minimum size (e.g., 64 bytes)




- External fragmentation
  - Enough free memory exists, but no block is big enough
  - E.g., `malloc()` asks for 128 contiguous bytes

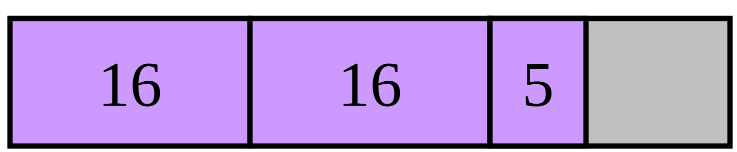


# Simple "K&R-Like" Approach

- Memory allocated in multiples of a base size
  - E.g., 16 bytes, 32 bytes, 48 bytes, …
- Linked list of free blocks
  - **`malloc()`** and **`free()`** walk through the list to allocate and deallocate
- **`malloc()`** allocates the *first* big-enough block
  - To avoid sequencing further through the list
- **`malloc()`** *splits* the free block
  - To allocate what is needed, and leave the rest available
- Linked list is *circular*
  - To be able to continue where you left off
- Linked list in the order the blocks appear in memory
  - To be able to "coalesce" neighboring free blocks

# Allocate Memory in Multiples of Base Size

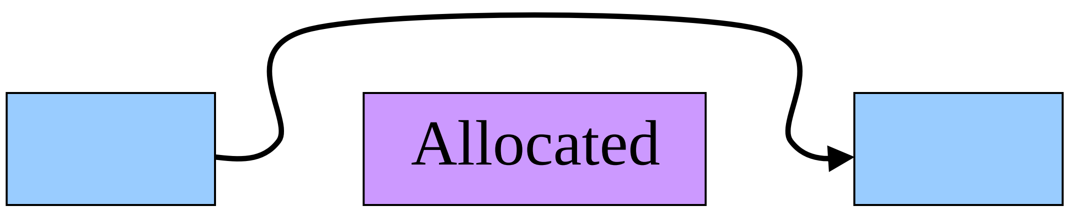
- Allocate memory in multiples of a base size
  - Avoid maintaining very tiny free blocks
  - Align memory on size of largest data type (e.g., double)
- Requested size is "rounded up"
  - Allocation in units of base_size
  - Round: **`(nbytes+base_size-1)/base_size`**
- Example:
  - Suppose nbytes is 37
  - And base_size is 16 bytes
  - Then (37 + 16 - 1)/16 is 52/16 which rounds down to 3



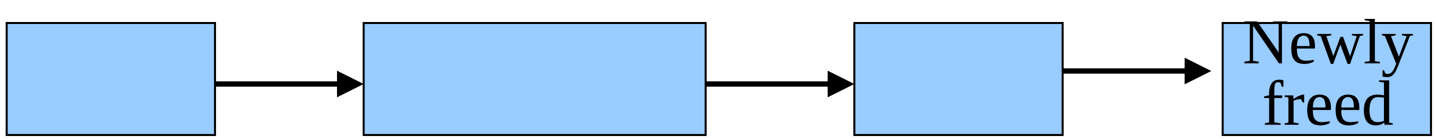
# Linked List of Free Blocks

- Linked list of free blocks

- **`malloc()`** allocates a big-enough block

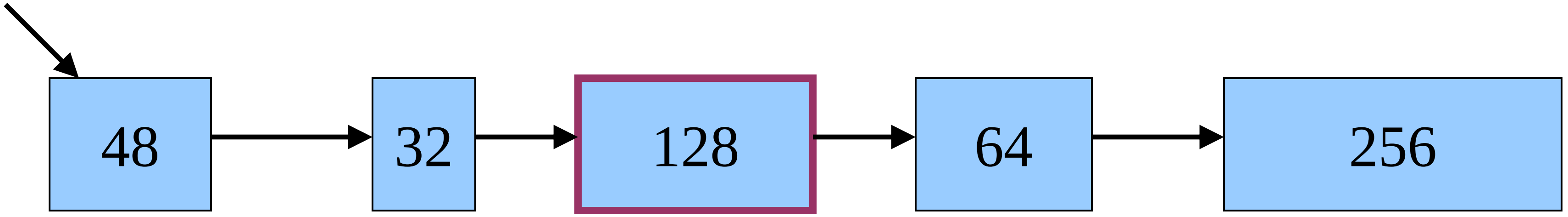



- **`free()`** adds newly-freed block to the list



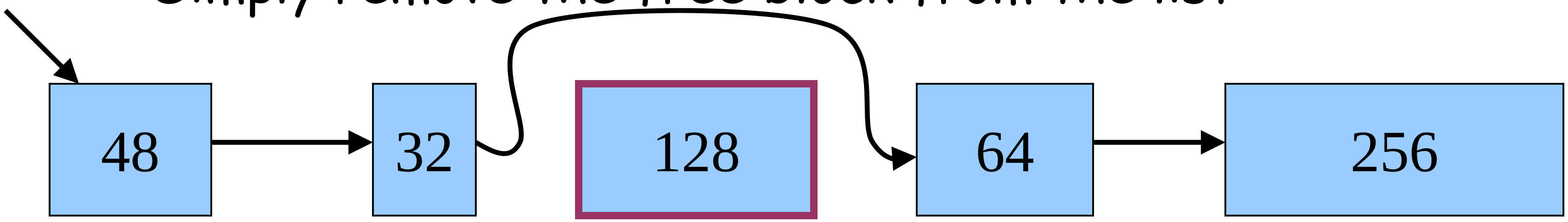
# "First-Fit" Allocation

- Handling a request for memory (e.g., **`malloc()`**)
  - Find a free block that satisfies the request
  - Must have a "size" that is big enough, or bigger
- Simplest approach: first fit
  - Sequence through the linked list
  - Stop upon encountering a "big enough" free block
- Example: request for 64 bytes
  - First-fit algorithm stops at the 128-byte block

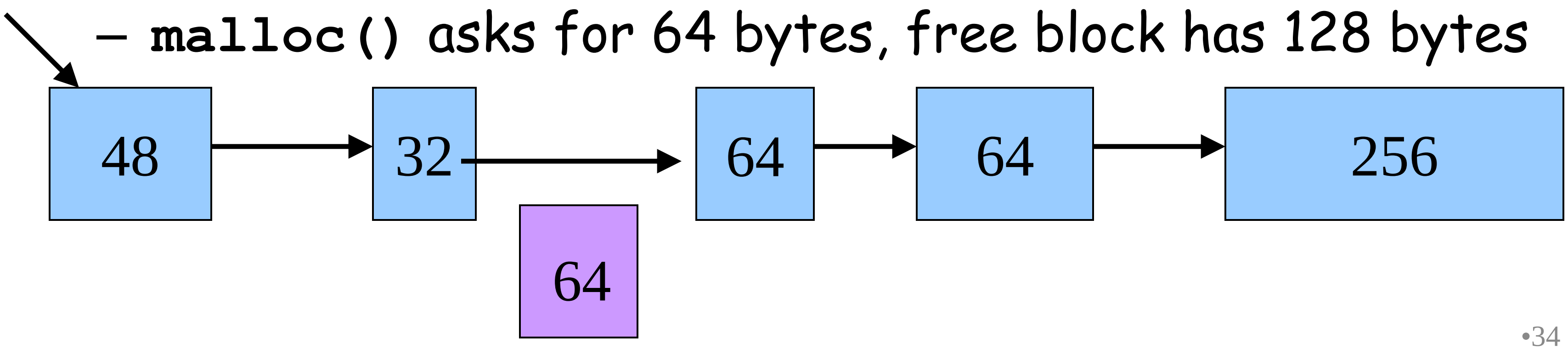


# Splitting an Oversized Free Block

- Simple case: perfect fit
  - `malloc()` asks for 128 bytes, free block has 128 bytes
  - Simply remove the free block from the list

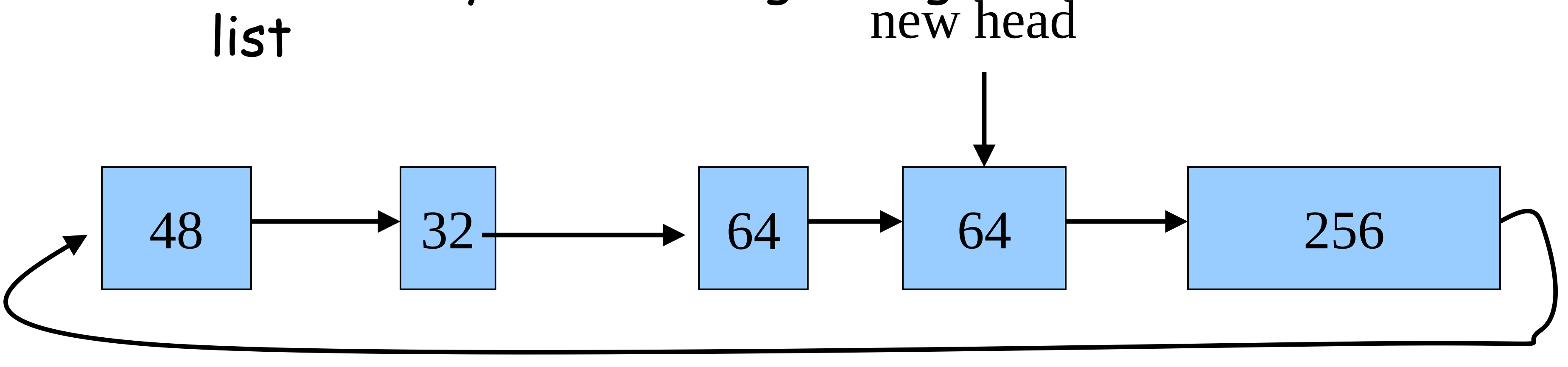



- Complex case: splitting the block
  - `malloc()` asks for 64 bytes, free block has 128 bytes



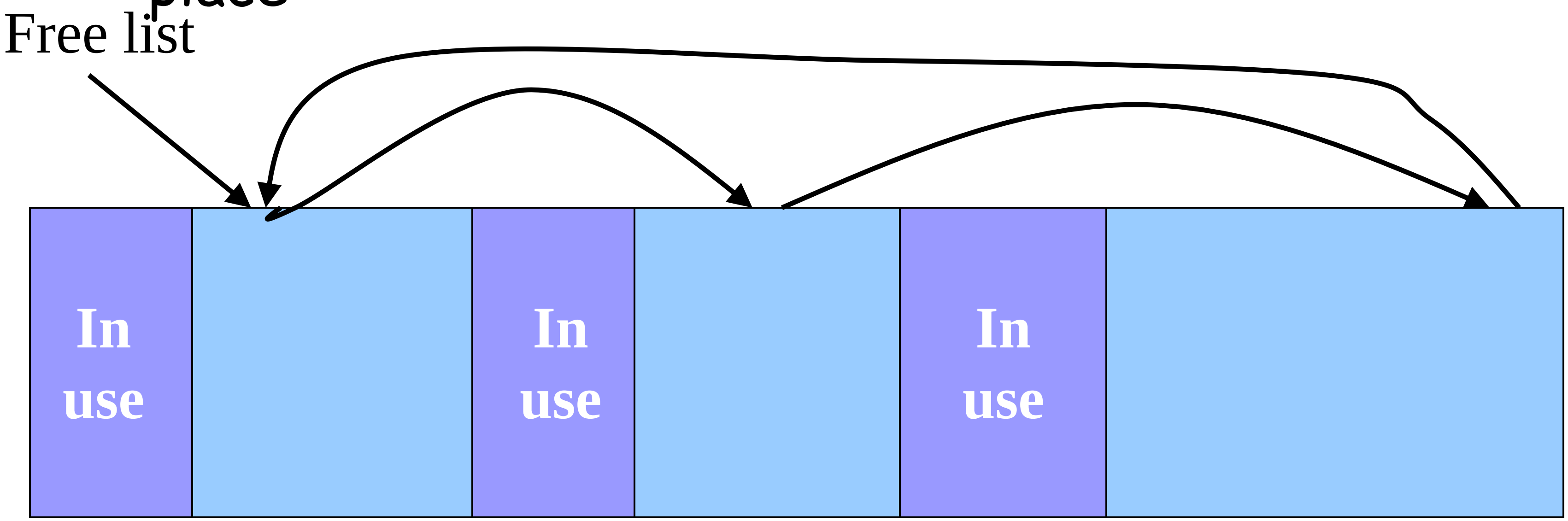
# Circular Linked List of Free Blocks

- Advantages of making free list a circular list
  - Any element in the list can be the beginning
  - Don't have to handle the "end" of the list as special
- Performance optimization
  - Make the head be where last block was found
  - More likely to find "big enough" blocks later in the list



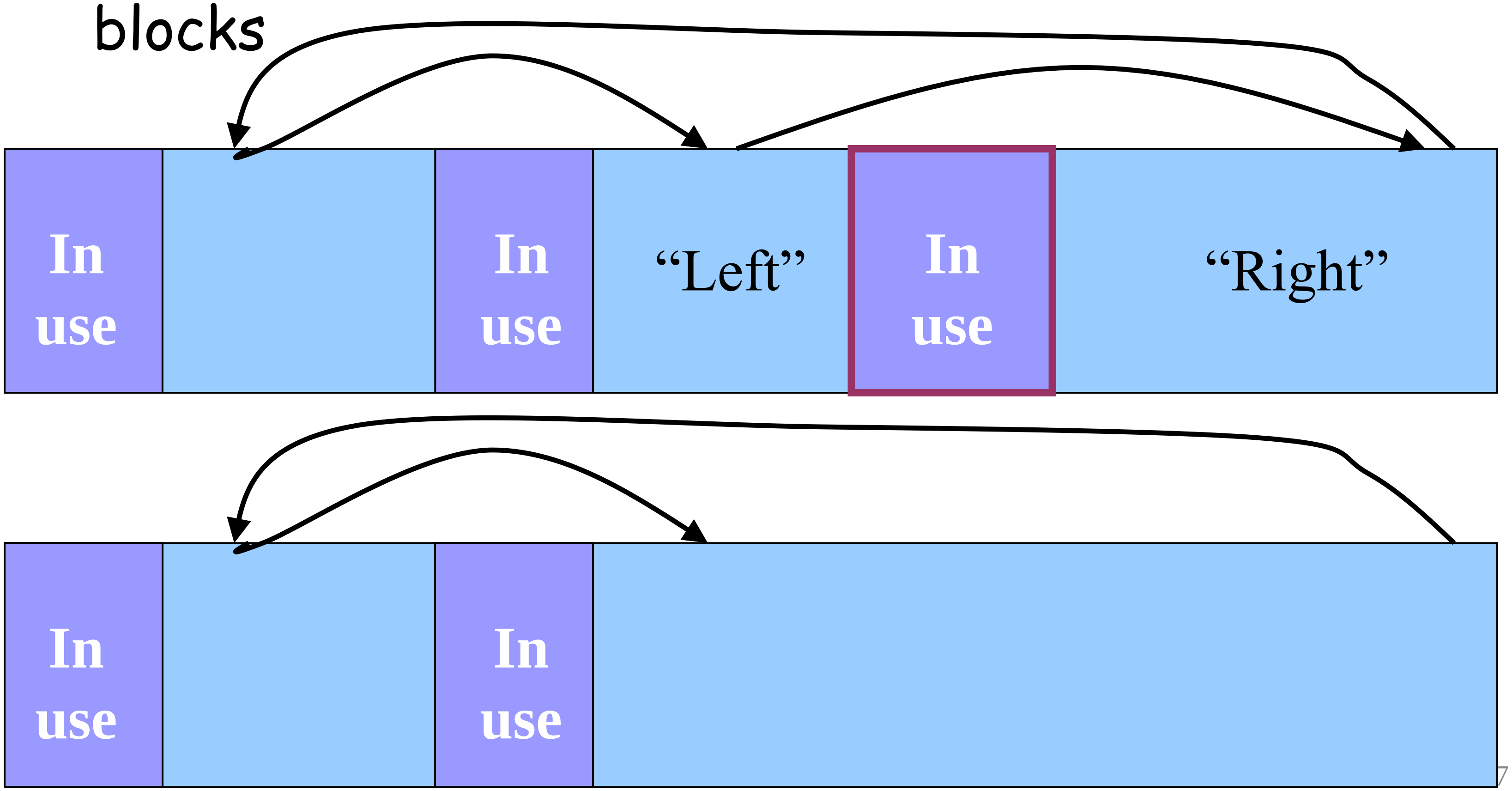
# Maintaining Free Blocks in Order

- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks
- Though, makes calls to free() more expensive
  - Need to insert the newly-freed block in the right place



# Coalescing Adjacent Free Blocks

- When inserting a block in the free list
  - "Look left" and "look right" for neighboring free blocks



# Conclusion

- Elegant simplicity of K&R **`malloc()`** and **`free()`**
  - Simple header with pointer and size in each free block
  - Simple circular linked list of free blocks
  - Relatively small amount of code (~25 lines each)
- Limitations of K&R functions in terms of efficiency
  - **`malloc()`** requires scanning the free list
    - To find the first free block that is big enough
  - **`free()`** requires scanning the free list
    - To find the location to insert the to-be-freed block