CS 24
Introduction to Computer Systems

8: Memory Management

Questions?

Objective

- Dynamic Memory Management
  - C-style (malloc/free)
  - Automatic

About Memory

- Statically-allocated globals
- Stack-allocated local variables
- Dynamically-allocated

```
int *resultarr =
    (int *) malloc(5*sizeof(int));
```

About Memory

- Statically-allocated globals

```
    .data
x:  .long 3257  # int x = 3257;
a:  .space 36   # int a[9];
```

- Stack-allocated local variables

- Dynamically-allocated

```
int *resultarr =
    (int *) malloc(5*sizeof(int));
```

About Memory

- Statically-allocated globals

```
    .data
x:  .long 3257 # int x = 3257;
a:  .space 36  # int a[9];
```

- Stack-allocated local variables

```
push %ebp
mov %esp,%ebp
sub $4,%esp
movl $7381,-4(%ebp)
```

- Dynamically-allocated

```
int *resultarr =
    (int *) malloc(5*sizeof(int));
```

About Memory

- Statically-allocated globals

```
    .data
x:  .long 3257  # int x = 3257;
a:  .space 36   # int a[9];
```

- Stack-allocated local variables

```
push %ebp
mov %esp,%ebp
sub $4,%esp
movl $7381,-4(%ebp)
```

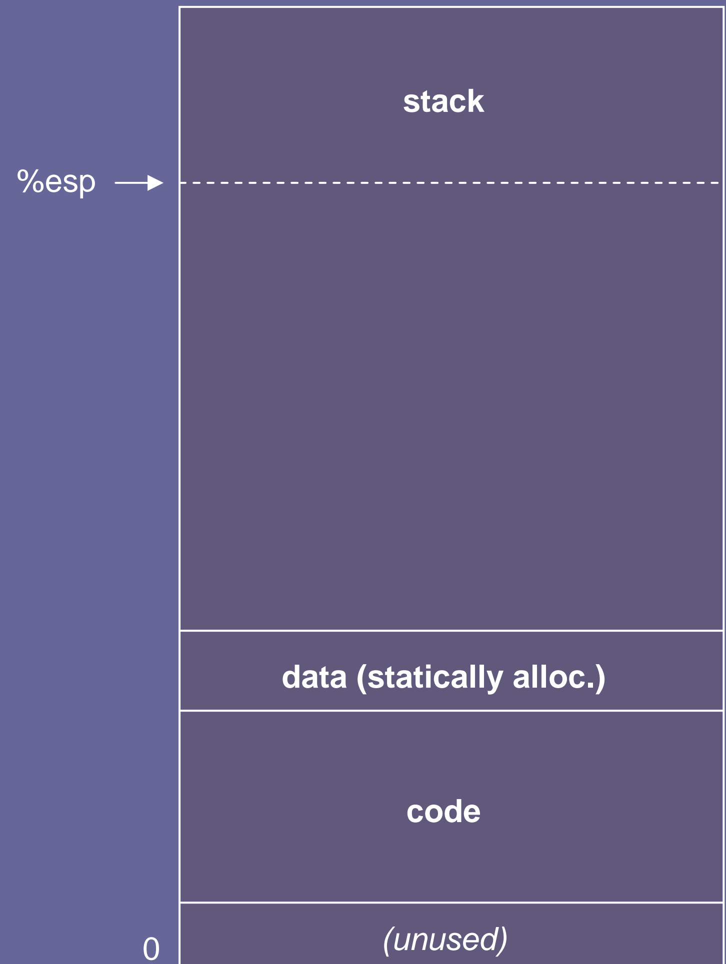
- Dynamically-allocated

```
int *resultarr =
    (int *) malloc(5*sizeof(int));
```

???

All of Memory (not to scale)

- For now, ignore operating system.
- Code, static data fill fixed amount of memory.
- Stack grows into empty space from upper limit.



Dynamic Allocation

- Most languages have support for dynamic memory allocation.
  - malloc; new; cons.

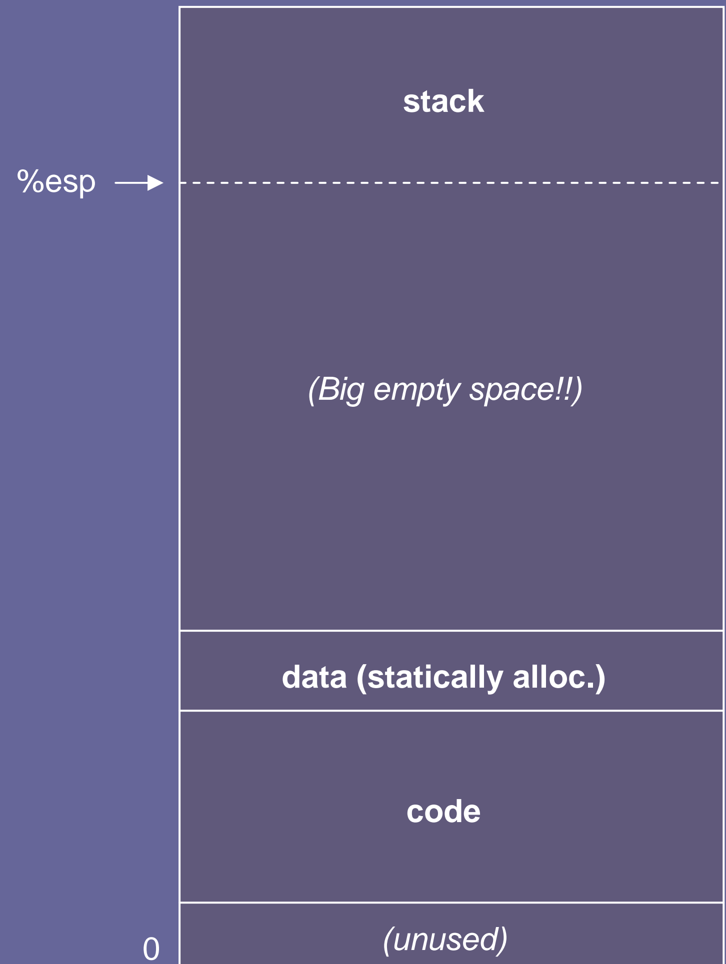
- Unbounded size.
  - Unbounded # of “objects”.
  - Size of each object not statically known.

    ```
    int[] a = new int[Integer.parseInt(System.in.readLine())];  // Java
    ```
- Arbitrary lifetime.
  - Can create new “object” at any time.
  - Not tied to particular procedure activation.
    - May outlive creating procedure.
    - Can’t live in stack frame.

- Need a new mechanism to:
  - Keep track of available/unavailable space.
  - Determine where to put each new “object”.

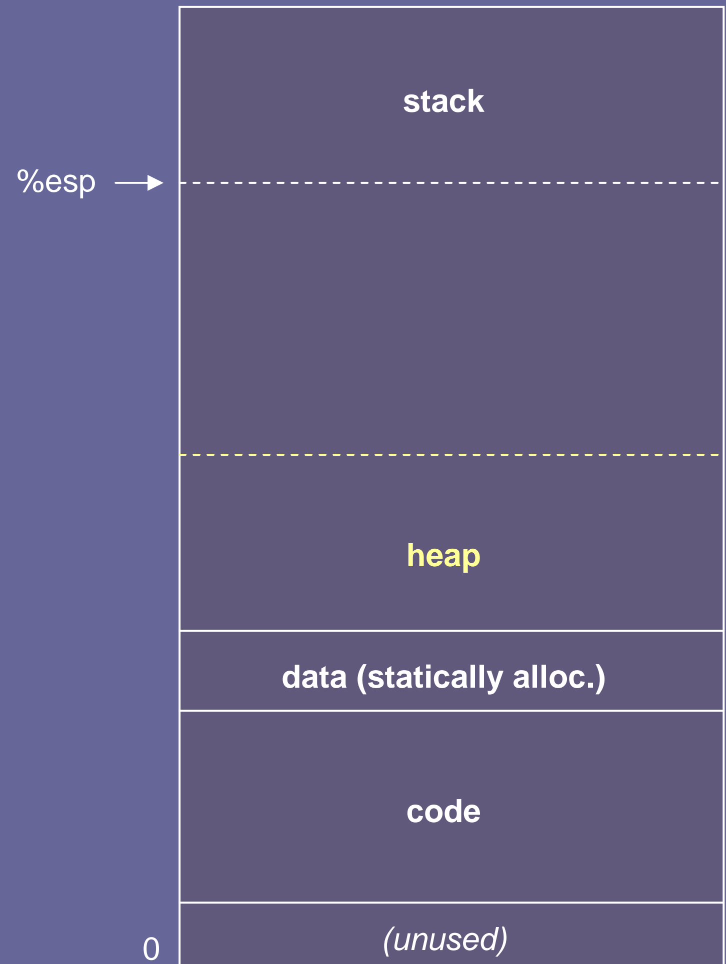
All of Memory (not to scale)

- Empty space in middle is bigger than it looks.
- Plenty of room for more data.



All of Memory (not to scale)

- Empty space in middle is bigger than it looks.
- Plenty of room for more data.
- Dynamically-allocated area grows upward.
- Called the heap.
  - Sort of like the stack, but less orderly.
- If stack & heap collide, you're out of memory.



Managing the Heap

- Allocating heap "objects" would be easy…

```
        .data
allocptr: .space 4
        .text
        …
        mov allocptr,%eax
        add $OBJECTSIZE,allocptr
        …
```

- …if they had to last forever.
  - Need to find out about no-longer-needed objects.
  - Re-use that space rather than continuing to increase allocptr.
  - Otherwise, long-running program would run out of memory even if it only needed a small amount *at a time*.

- In fact, managing a heap requires nontrivial algorithms.

C Memory Management

Two functions you've already seen:

- void *malloc(unsigned int size);
  - Request a block of size bytes, aligned so anything can go there.
  - Used like this:

```
/* For single object: */
sometype *w = (sometype *) malloc(sizeof(sometype));
/* For array: */
sometype *w = (sometype *) malloc(arraylen*sizeof(sometype));
```

- void free(void *p);
  - Notify heap manager that block pointed to by p is no longer needed.

Note: (void *) means pointer to don't-know-what.

Memory Allocation in Assembly

- malloc and free are part of the standard C library.
- Library itself is written (mostly) in C.
- By default, gcc links library object code into every program it generates.
- So malloc/free (and lots more) can be called from our assembly programs.
  - Since our "driver" programs are in C, we might as well use the library.

- Call malloc just like any other function:

```
pushl $8
call malloc
add $4,%esp
# %eax is a pointer to an 8-byte block.
```

Implementing malloc and free

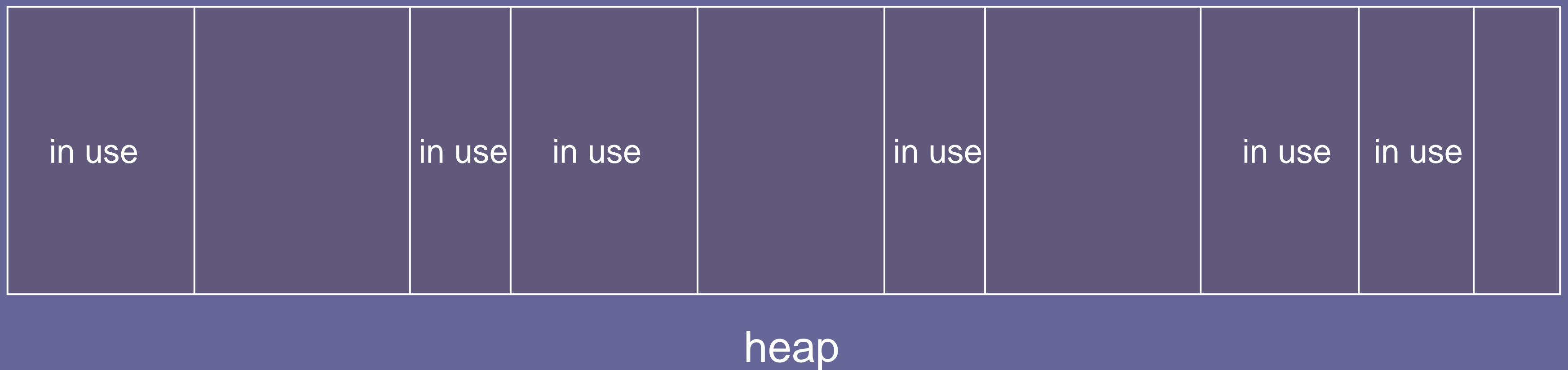
- Basic idea: maintain a “free list”.
  - Blocks that were allocated, then freed; now available to be used again.
- To malloc:
  - If free list contains some block of sufficient size, remove it from free list and return it.
  - If can’t find a good block in free list, either:
    - Expand heap further into unused region of memory (may need OS’s permission), or
    - Give up (return NULL).
- To free:
  - Place block in free list.

The Free List

- Q: Where to store list of free blocks?

A: Use the free blocks! (No one else is!)

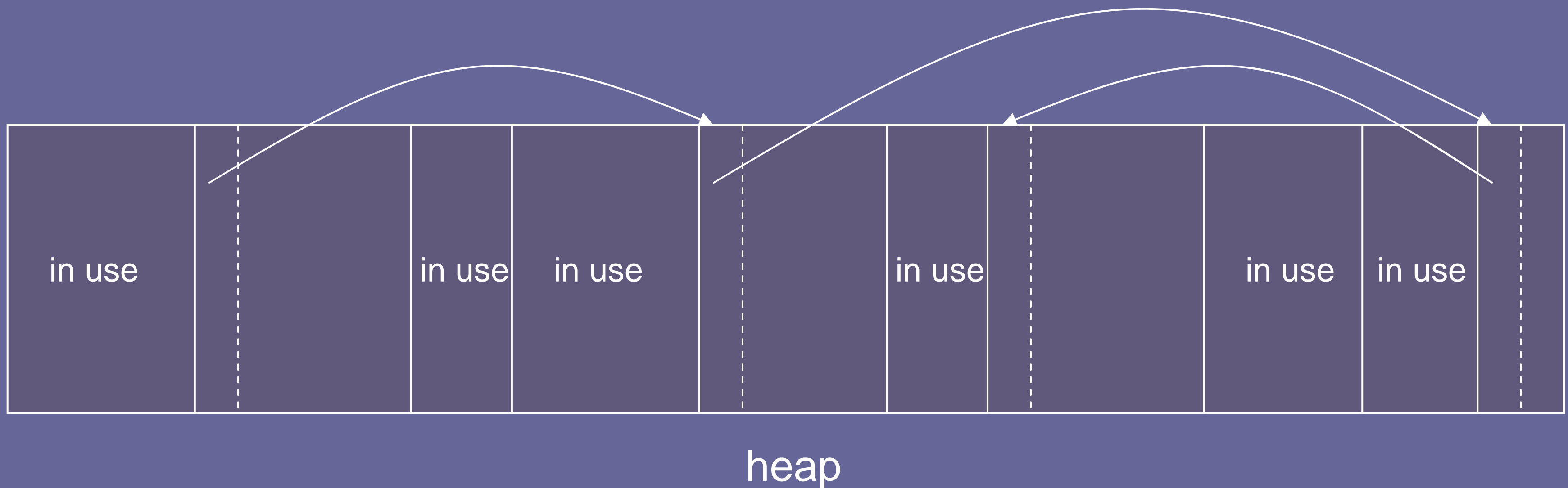
Each free block has header w/size, pointer to next one



The Free List

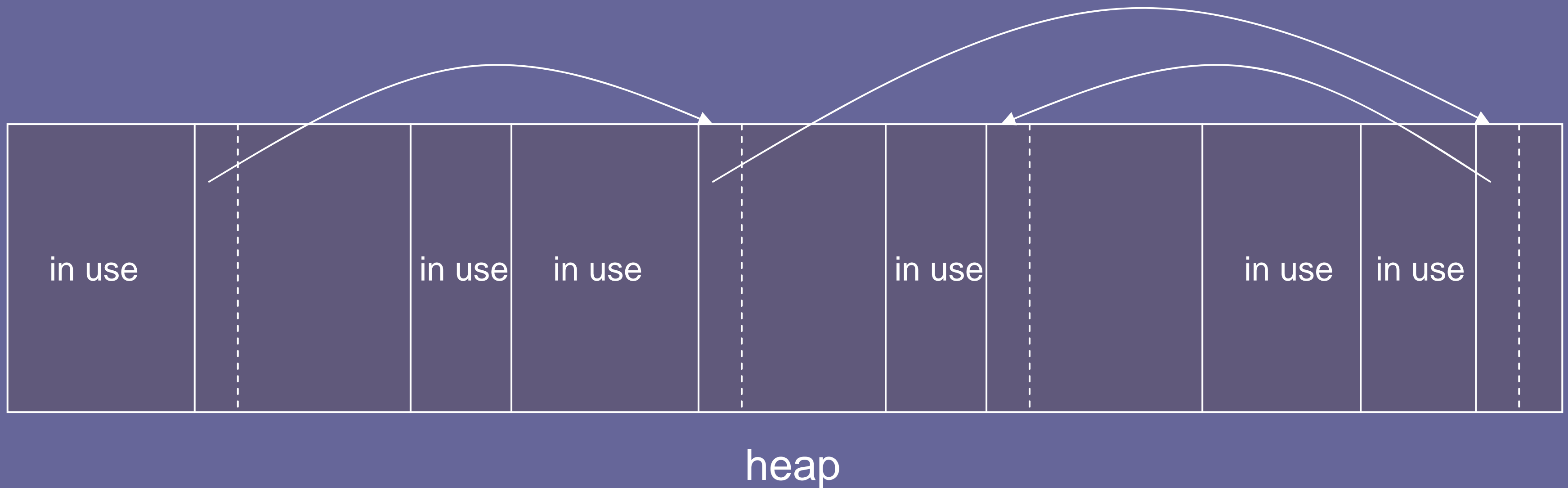
- Q: Where to store list of free blocks?

  A: Use the free blocks! (No one else is!)
  Each free block has header w/size, pointer to next one.



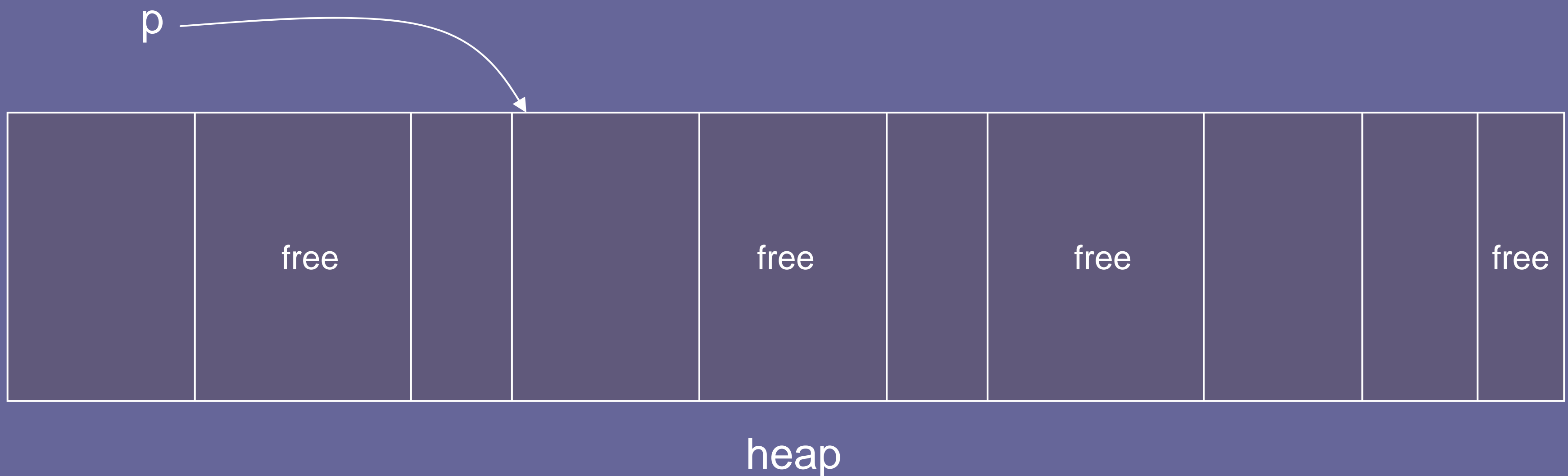
The Free List

```
struct memblock {
    unsigned size;
    struct memblock *next;
    unsigned char[] data;          /* Variable size. */
}
```



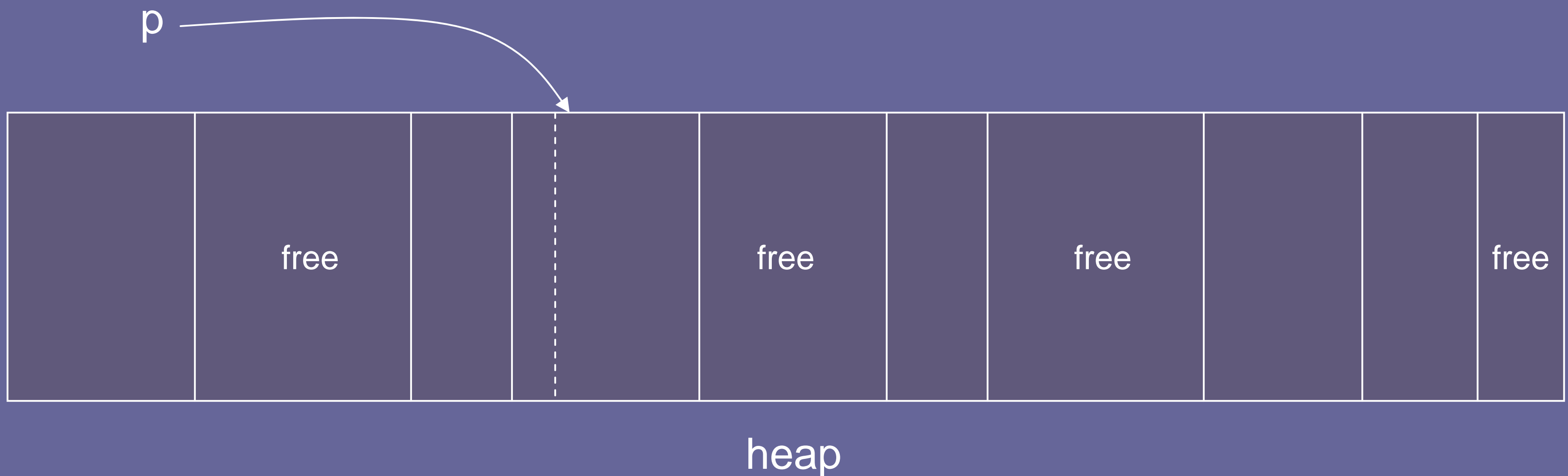
Next Problem

- free(p) must insert p into free list.
- Need to know size of block p points to.



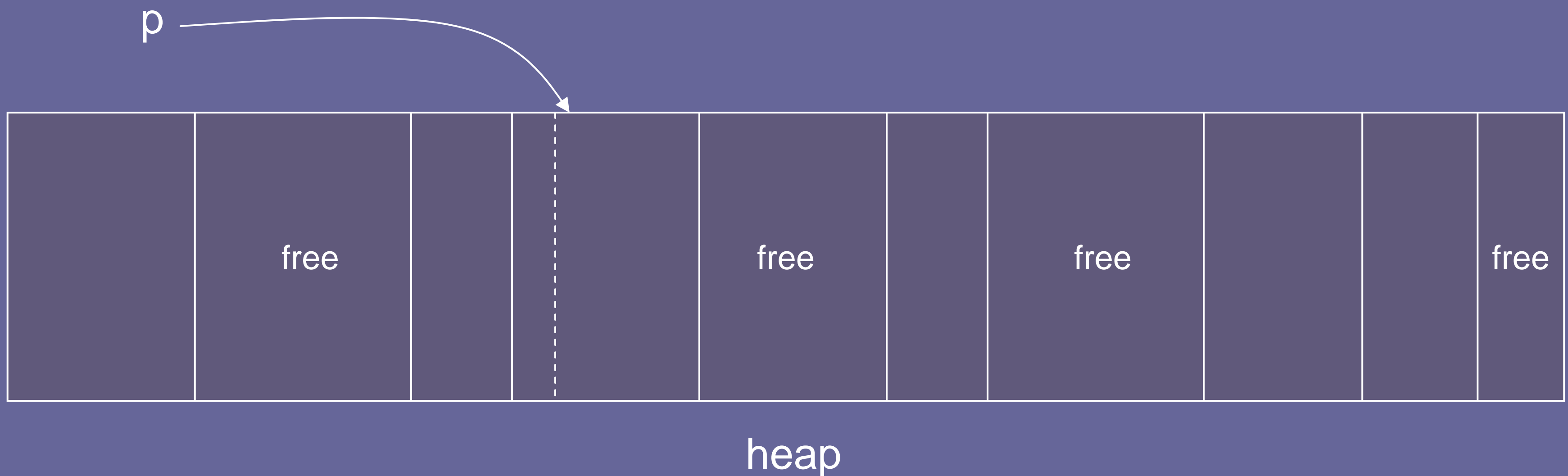
Next Problem

- Allocated blocks have headers too.
- malloc returns pointer to location *after* header.



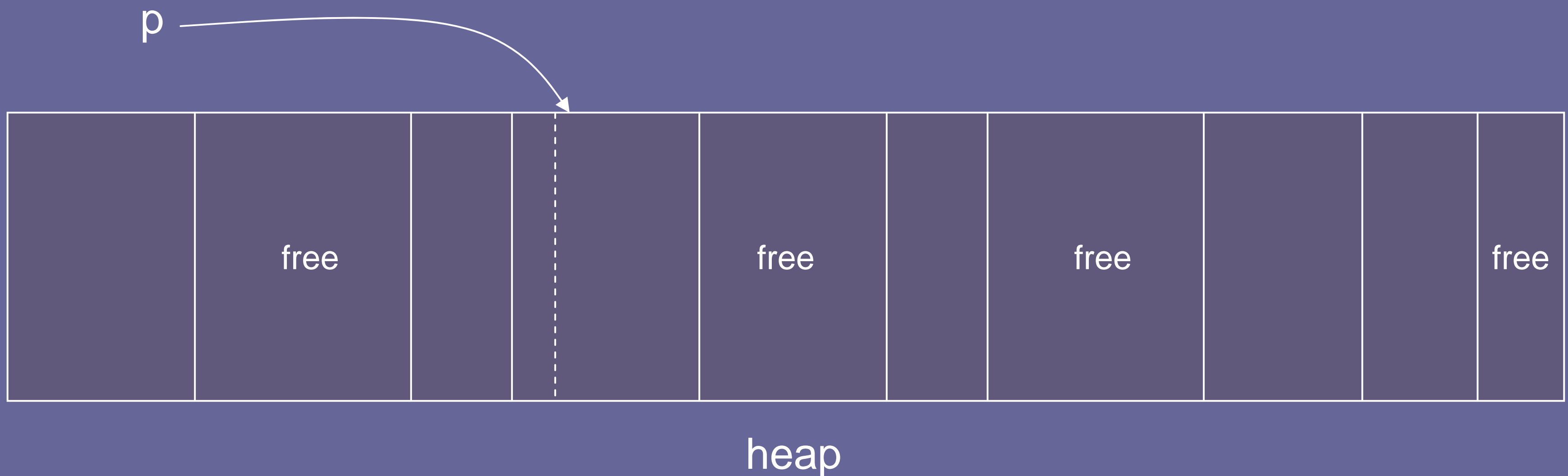
Sketch of malloc

```
struct memblock *b = find_a_free_block(sz);
/* b->size >= sz; b no longer in free list. */
return &(b.data);
```



Sketch of free

```
unsigned char *p = (unsigned char *) arg;
struct memblock *b = (struct memblock *) (p - headersize);
add_to_free_list(b);
```



Performance Issues

- Speed.
- Fragmentation – unusable memory due to size issues (wastes space).
  - Internal fragmentation:
    Allocator returns larger block than caller requested.
  - External fragmentation:
    Free blocks all too small to satisfy request.

- These issues affect design of allocation algorithms, data structures.
  - First-fit / best-fit, coalescing, more…

Memory Management Pitfalls

- C memory management is simple – too simple.
  - Implementation is simple;
  - Semantics is simple;
  - Using it correctly is not easy!*
- Many mistakes are possible.
- Some are "silly" – easy to fix.
  Others are serious and subtle bugs.
- This is why modern languages generally do things differently.

*See Wikipedia:Worse is better

“Silly” memory errors

- Size argument of malloc.
  - Example:
    Assuming the declaration:

    ```
    typedef struct listnode *list;
    ```

    which of these should you write?

    ```
    list L = (list) malloc(sizeof(list));
    list L = (list) malloc(sizeof(struct listnode));
    ```

  - Example:

    ```
    char *new = (char *) malloc(strlen(old));
    strcpy(new,old);
    ```

“Silly” memory errors

- Size argument of malloc.
  - Example:
    Assuming the declaration:
    ```
    typedef struct listnode *list;
    ```
    which of these should you write?
    ```
    list L = (list) malloc(sizeof(list));
    list L = (list) malloc(sizeof(struct listnode));
    ```
  - Example:
    ```
    char *new = (char *) malloc(strlen(old));
    strcpy(new,old);
    ```

“Silly” memory errors

- Uninitialized pointer.
  - Example:

```
struct listnode n;       /* Initially contains junk. */
n.next->key = 3;
```

Serious Error: Dangling Pointer

- Using a pointer that has been freed.

```
p = malloc(…);
…
free(p);
…
p->key = 3;
```

Serious Error: Dangling Pointer

- Using a pointer that has been freed.

```
p = malloc(…);
…
free(p);
…
p->key = 3;
```

- (Equivalently, freeing an object while it's still needed.)
- Problem: memory at address p may have been re-allocated.
- This example looks silly, but can happen in non-silly ways.

Serious Error: Memory Leak

- Forgetting to free an object that's no longer needed.

```
for (i = 0; i < 10; i++) {
    p = malloc(20);
}
free(p);
```

- Objects allocated by first 9 iterations never freed.
  - And *can never* be, since pointers to them were lost. (But that's not the point!)
- Symptom: Program gradually uses more and more memory, eventually dies.

Automated Memory Management

- Knowing when to free an object is hard.
  - Free too soon: dangling pointer.
  - Wait too long: memory leak.
- Plan: let memory management system detect no-longer-needed objects and reclaim them.
  - No need to free.
  - No dangling pointer problems (by definition).
  - Fewer memory leaks (but not zero – will revisit).
- First needed/developed for LISP (1959).
  - In LISP, cons allocates – nothing deallocates.

Obvious Idea

- Reference counts: Each object tracks how many copies of its address exist.

Obvious Idea

- Reference counts: Each object tracks how many copies of its address exist.
- When copying a pointer, adjust reference counts.

```
p = q;
```

Obvious Idea

- Reference counts: Each object tracks how many copies of its address exist.
- When copying a pointer, adjust reference counts.

```
p->refcount--;
q->refcount++;
p = q;
```

Obvious Idea

- Reference counts: Each object tracks how many copies of its address exist.
- When copying a pointer, adjust reference counts.
- If reference count drops to zero, object is garbage.

```
if (--(p->refcount) == 0) {
        free(p);
}
q->refcount++;
p = q;
```

Problems

- *Why isn't this a good general solution?*

- If data structures have cycles, counts will never reach zero.
- Can be useful:
  - in special situations where there are definitely no cycles.
  - when combined with some other strategy for detecting useless cycles.

- General solutions: Garbage Collection (next time)

Problems

- *Why isn't this a good general solution?*

- If data structures have cycles, counts will never reach zero.
- Can be useful:
  - …in special situations where there are definitely no cycles.
  - …when combined with some other means of detecting useless cycles.

- General solutions: Garbage Collection (next time).