Dynamic Allocation

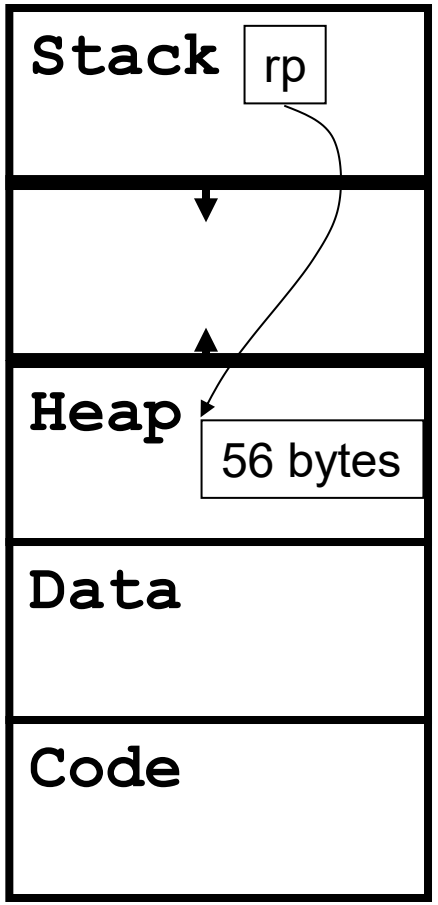
Dynamic Storage

- Example: We need space to allow the user to enter some data, but how much?
  - 80 bytes? 132? 1? 256? 10000?
- Two Friends from the C library: #include <stdlib.h>
  - malloc()
  - free()
- These friends create and manage the heap for us, allowing us to grab storage at run time

malloc()

```
struct r {
    char name[40];
    double gpa;
    struct r *next;
};
struct r *rp;
rp = malloc(sizeof(struct r));
if (rp == NULL){
    /* Handle Error! */
}
```

- Options for handling error
  - Abort
  - Ask again
  - Save user data
  - Ask for less
  - Free up something



Don’t Do This!

```
rp = malloc(sizeof struct r);

// Code foolishly inserted here!

if(rp == NULL)
```

- Be afraid.

- Using **rp** before you check it is a crash waiting to happen.

Idiomatic But Safe

```
if((rp = malloc(sizeof struct r)) == NULL){
   /* Handle Error Here */
}
```

- This is the way it's done in real life.
- Don't mix up the = and == operators!

- Alternate syntax *(because NULL==0)*:

```
if( !(rp = malloc(sizeof struct r)) ){
   /* Handle Error Here */
}
```

Escape From Type-Checking

- malloc() returns a pointer to at least as many bytes as we requested.
- But what is the type of this pointer and how do we use it to store items of a different type?
- malloc() is declared as “void *malloc(unsigned long)”
- C uses the idiom “pointer to void” for a generic pointer
- To be safe, you should cast this pointer into the correct type so that type-checking can work for you again!

```
int *ip;
ip = (int *)malloc(sizeof int);
```

- Otherwise, the compiler will silently cast your “void *” pointer into any other kind of pointer without checking

Done With a Chunk of Storage

- When you're done with a chunk of storage, you use **free()** to make it available for reuse.
- Remember, C doesn't do garbage collection
- From our previous example

```
free(rp);
```

  returns the memory back to the heap for re-use by someone else

- You **must not** use the value in rp after the call to **free()**, nor may you dereference the memory it points to!
- There's no guarantee what's at *rp after you call free() – assume it is garbage data!
- With modern C libraries, free(NULL) does nothing but isn't an error

After free()

- From our previous example

```
free(rp);
```

- The variable rp still exists.
  - It is a pointer to struct r
- But what it points to is now garbage data.
  - We should never dereference rp again: *rp
- We can, however, assign a new value to rp
  - That is okay – make it point somewhere else
- The compiler will NOT help you with this.
  - Mistakes will cause run-time errors

Other memory allocation functions

- void *malloc(size_t n);
  - Allocates (at least) n bytes of memory on the heap, returns a pointer to it
  - Assume memory contains garbage values

- void *calloc(size_t num, size_t size);
  - Allocates (at least) num*size bytes of memory on the heap, returns a pointer to it
  - Memory will be zero'ed out.

- void *realloc(void *ptr, size_t n);
  - Reallocates (at least) n bytes of memory on the heap, returns a pointer to it
  - Copies the data starting at ptr that was previously allocated
  - Often used to expand the memory size for an existing object on the heap

Pointer Video

- https://www.youtube.com/watch?v=5VnDaHBi8dM

Handling Persistent Data

Persistent Data I

```
char *foo(void)
{
   static char ca[10];
   return ca;
}
```

- Anyone calling this function now has access to ca in this block. Could be dangerous. Why?

- Note that this approach is not dynamic

Example

```
char *strFromUnsigned(unsigned u)
{
    static char strDigits[] = “?????” ;
    char *pch;
    pch = &strDigits[5];
    do
        *--pch = (u % 10) + '0';
    while((u /= 10) > 0);

    return pch;
}
```

Problem in use

```
strHighScore =
   strFromUnsigned(HighScore);

.

.

.

strThisScore =
   strFromUnsigned(ThisScore);
```

Persistent Data II

```
char *foo(void)

{

    char ca[10];

    return ca;

}
```

- Since ca was allocated on stack during function call pointer returned is now pointing to who-knows-what

Bad

Persistent Data III

```
char *foo(void)
{
   char *ca = malloc(...);
   /* error checking but no free */
   return ca;
}
```

- This actually works, but the caller needs to know that they're responsible for the free()

Memory Leaks

- Memory leaks occur when the programmer loses track of memory allocated by malloc or other functions that call malloc

```
void foo(void)
{
        char *ca = malloc(...);
        /* no free */
        return;
}
```

- Bad

Memory Management

- Some functions that call malloc
  - calloc
  - strdup
  - regcmp
  - others...
- C doesn't do automatic memory management for efficiency reasons
  - If you want to manage memory...do it yourself!

calloc()

```
void *calloc(size_t num, size_t size);
```

- Call malloc() to find space for **num** new allocations
- Initialize the space to zero (0)
- Not much to discuss…

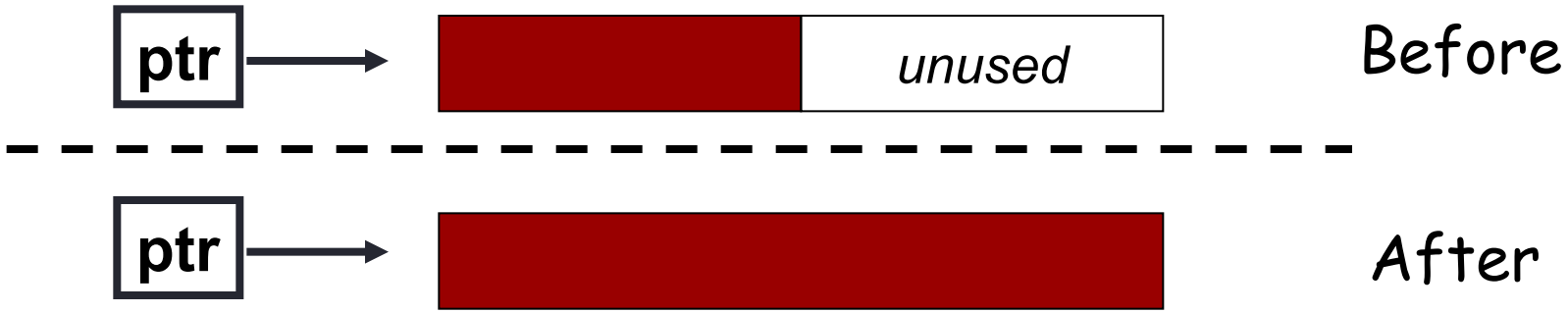
realloc()

```
ptr = realloc(ptr, num_bytes);
```

- What it does (conceptually)
  - Find space for new allocation
  - Copy original data into new space
  - Free old space
  - Return pointer to new space

realloc()

realloc: What might happen



realloc

- Realloc may return
  - same pointer
  - different pointer
  - NULL
- Is this a good idea?

```
cp = realloc(cp, n);
```

1. Yes
2. No
3. Sometimes

Realloc

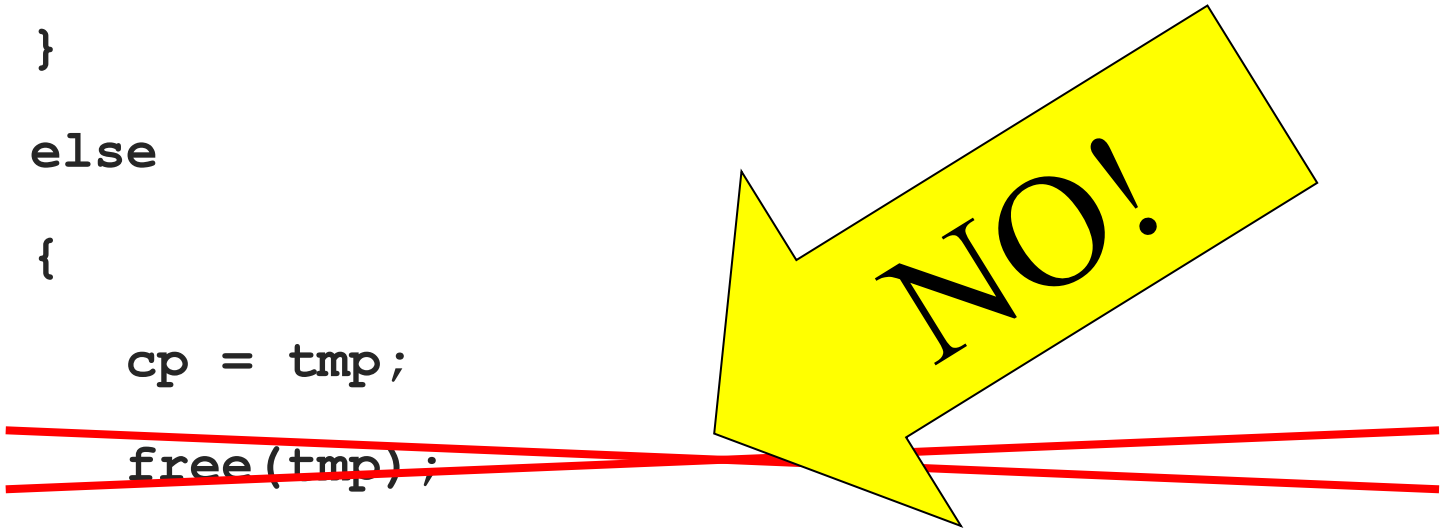
- Is this a good idea?

```
cp = realloc(cp, n);
```

- No!
- If realloc returns NULL cp is lost
- Memory Leak!

How to do it properly

```
void *tmp;

if((tmp = realloc(cp,...)) == NULL)

{

   /* realloc error */

}

else

{

   cp = tmp;

   free(tmp);

}
```



Additional Edge Cases

- realloc(NULL, n) ≡ malloc(n);
- realloc(cp, 0) ≡ free(cp); // only on some compilers
- These can be used to make realloc work in a single loop design to build a dynamic structure such as a linked list.

Example

```
int size = 0;      /* Size of "array" */
int *ip = NULL;    /* Pointer to "array" */
int *temp;
int i;
char buffer[80];
while(fgets(buffer, 80, stdin) != NULL) {
    size++;
    if((temp = realloc(ip, size*sizeof(*temp))) == NULL){
        fprintf(stderr, "Realloc failure\n");
        exit(EXIT_FAILURE);
    }
    ip = temp;
    ip[size-1] = strtol(buffer, NULL, 10);
}
```

Dynamic Allocation: What can go wrong?

- Allocate a block and lose it by losing the value of the pointer
- Allocate a block of memory and use the contents without initialization
- Read or write beyond the boundaries of the block
- Free a block but continue to use the contents
- Call realloc to expand a block of memory and then – once moved – keep using the old address
- FAIL TO NOTICE ERROR RETURNS

Questions?

Question

Is there anything wrong with the following code that frees all the nodes in a linked list?

```
for (struct node *p = Head; p != NULL; p = p->next)
    free(p);
```

A. The head of the list doesn't get freed

B. The code is satisfactory as it stands

C. The reinitialization uses memory that has been freed 

D. The code uses free() multiple times on the same pointer

```
struct node *t;
for (struct node *p = Head; p != NULL; p = t) {
    t = p->next
    free(p);
}
```

Malloc Implementation

Outline

- K&R Malloc Implementation
  - Headers
  - Heap Layout
  - Allocating the Correct Amount of Memory
  - **`malloc()`**: Getting the memory for work
  - **`free()`**: Recycling the memory when done
  - **`morecore()`**: OS requests for real memory

Memory Management

- The K&R implementations of malloc() and free() are one of many ways to implement these.
- We've refactored the code a bit. It was written for compactness and efficiency in the 1970s C environment; things have changed and we can afford to be a little bit more verbose for clarity.
- It's not necessarily the most efficient, and certainly not the only way:
  - http://codinghighway.com/2013/07/13/the-magic-of-wrappers/
  - http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf
  - https://danluu.com/malloc-tutorial/

How K&R Malloc Works

- Linked list to track free memory
  - Located in the Heap
  - *Circular* Linked List

  - This is the ***Free List***
    - Contains memory available to be malloc'ed
    - Once memory is malloc'ed, we don't track it until it is free'd

How K&R Malloc Works

- `void *malloc(size_t n);`
  - Delete a node from the free list
  - Return a pointer to that node to the program
- `void free(void *ptr)`
  - Insert the node ptr into the free list

- Note: We've renamed them _malloc() and _free() in our code to keep from conflicting with the malloc() and free() from the C library
  - There are a number of C library functions that use malloc() and free(); we don't want override the library versions until our versions work properly!

Question

In the K&R implementation of malloc(), the data structure in the heap is a:

A. Binary Tree
B. Hash Table
C. Circular Linked List 
D. Array of Structs

Question

In the K&R implementation of the heap management, free() does the following:

A. Deletes a node from the free list

B. Inserts a node in the free list 

C. Zeroes out the bytes on the heap

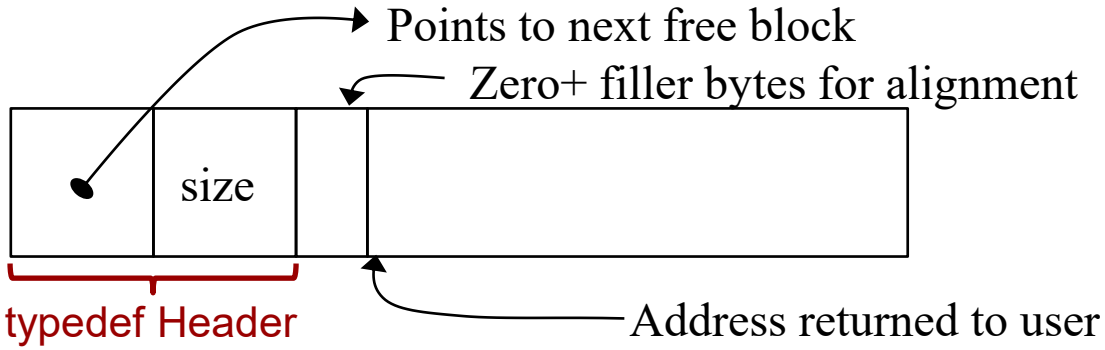
D. Reduces the size of the heap

How K&R Malloc Works

- Each node on the free list is available memory
  - Everything on the free list is in heap memory areas that malloc has requested from the system
  - Other functions can request heap space from the system, but that space will never show up on malloc's free list

- Consider this:
  - We want each block in the free list to be as large as possible
    - So it is more likely to have enough bytes to satisfy a malloc() call
    - Therefore when we free a block adjacent to our other memory on the heap, we should merge that block into the adjacent block.

A Node in the Free List



A block returned by malloc

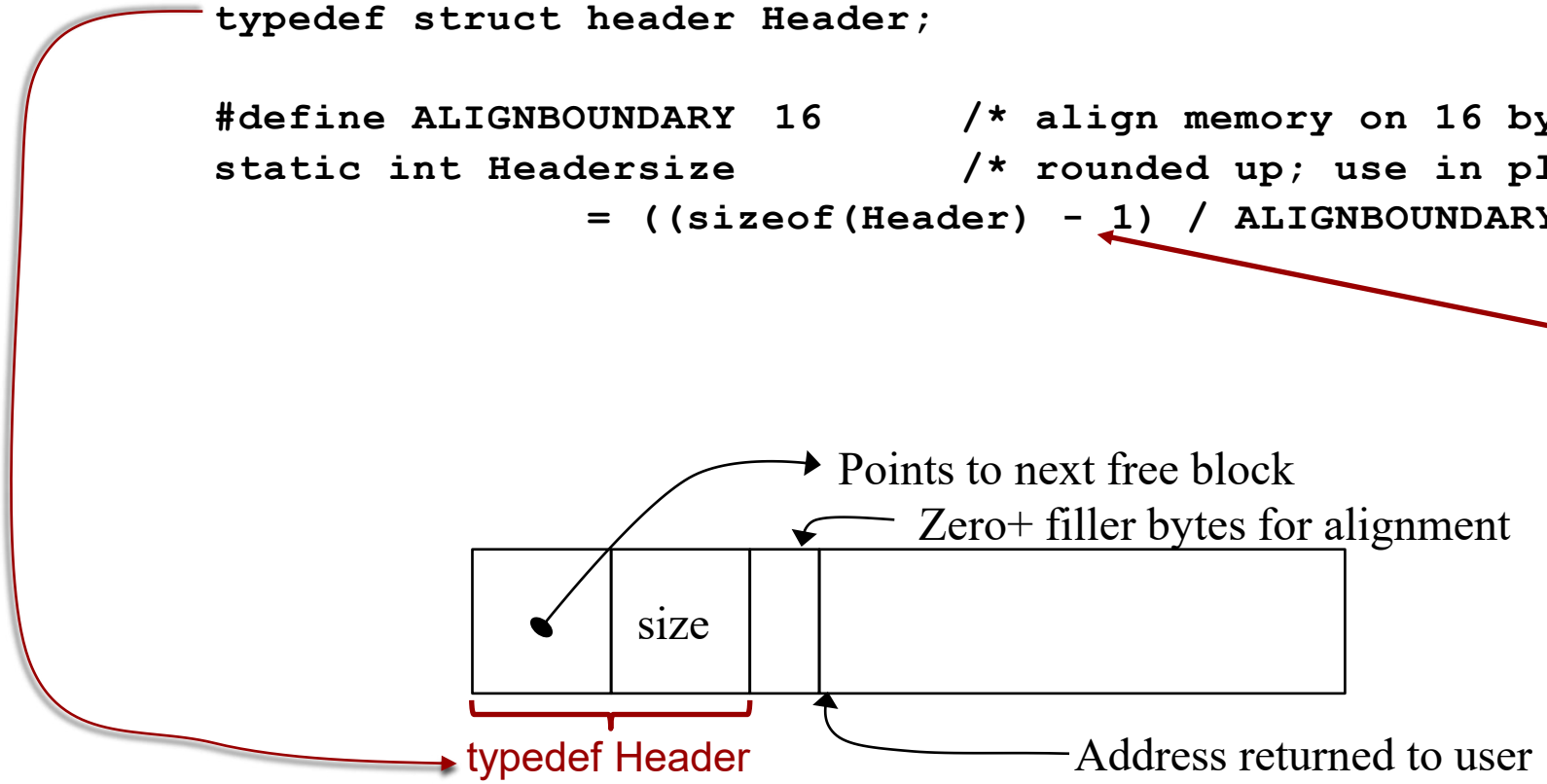
How K&R Malloc Works

- Design choices
  - Linked List is ordered by memory address (in the heap)
  - We could also order the linked list from largest to smallest size
  - There are many other ways we could implement malloc()

- Size in K&R malloc is number of *units*
  - Not number of bytes
  - A block is sizeof(Header) rounded up to the strictest alignment boundary
  - A block is 16 bytes in this implementation

Section of K&R Code

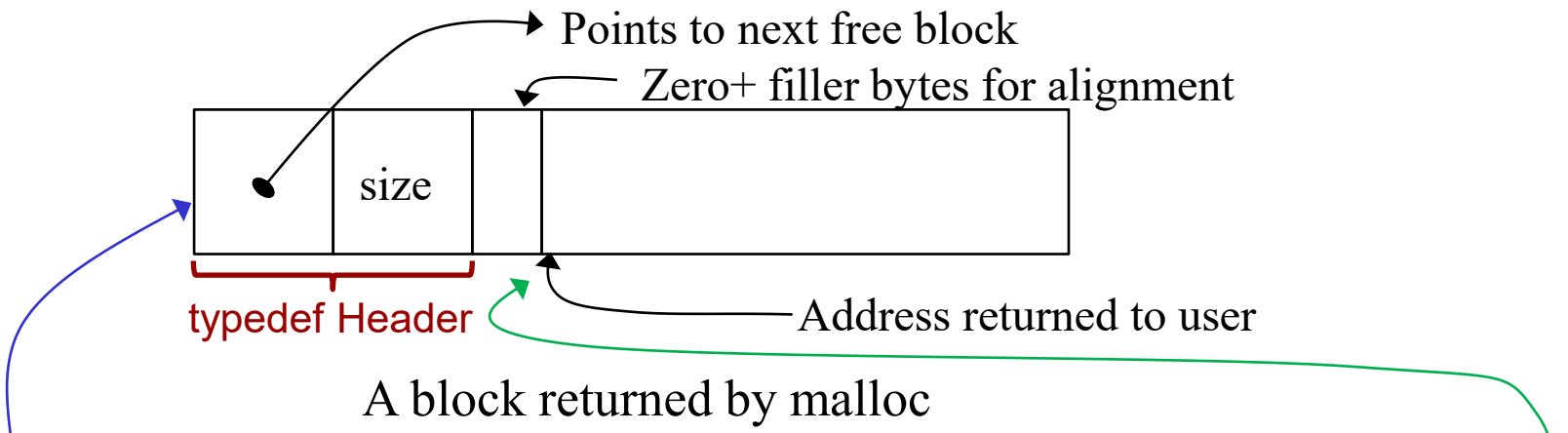
```
struct header {                 /* block header */
    struct header *next;        /* next block if on free list */
    unsigned size;              /* size of this block in Headersize units */
};
typedef struct header Header;

#define ALIGNBOUNDARY  16       /* align memory on 16 byte boundaries */
static int Headersize           /* rounded up; use in place of sizeof(Header) */
                = ((sizeof(Header) - 1) / ALIGNBOUNDARY + 1) * ALIGNBOUNDARY;
```



A block returned by malloc

sizeof(Header)	Headersize
8	16
12	16
16	16
20	32

Returned by Malloc



- If the variable

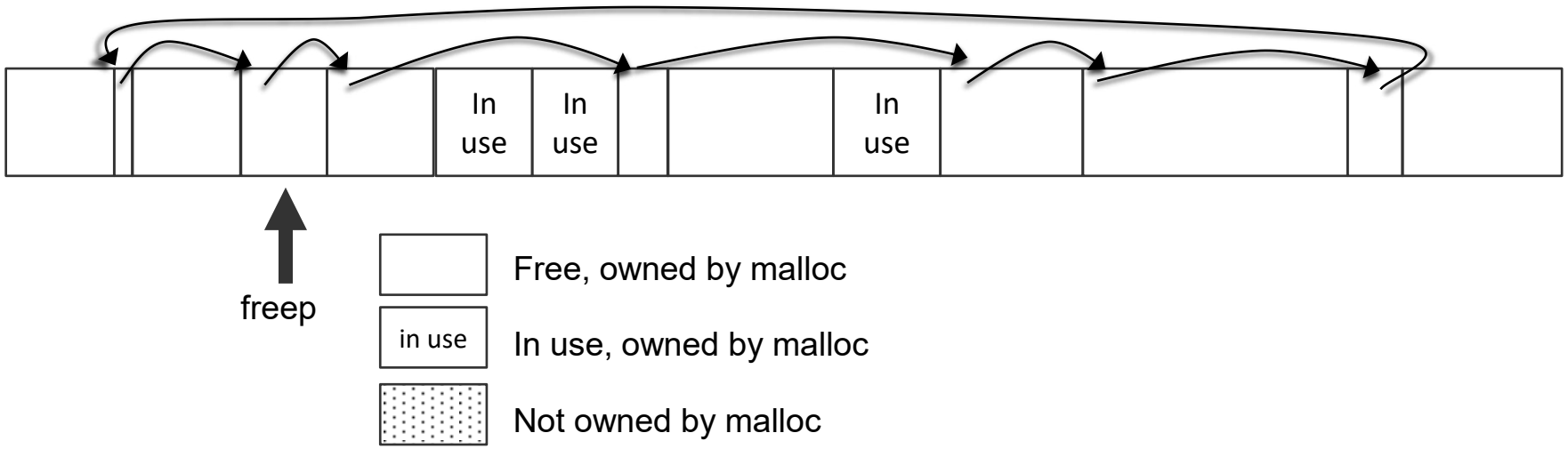
  `Header *p;`

  points to the beginning of a block, where does

  `(char *)p + Headersize`

  point?

Heap Layout



- K&R ch. 8
- One of those free blocks has user data size 0 and isn't in the heap, so it is never handed over by malloc().

Section of K&R Code

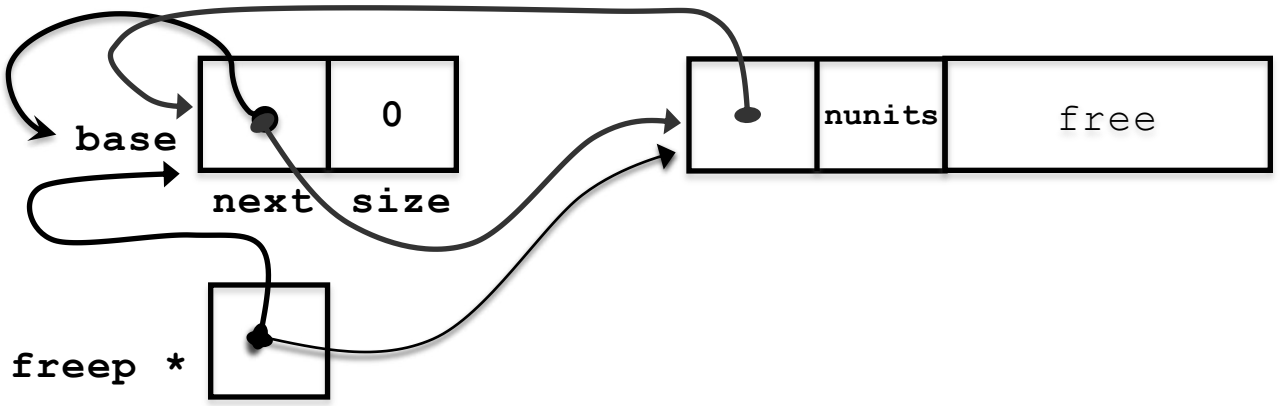
```
static Header base;                 // empty list to get started with
static Header *freep = NULL;         // start of free list

/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    unsigned long nunits;

    /* Allocate memory in Headersize units, so round up request */
    nunits = (nbytes + Headersize - 1) / Headersize + 1;
```

- The key here is alignment with the Header's size
- You need (just) enough bytes to cover the program's request and the header information
- Suppose that the Header consists of 16 bytes [Headersize = 16]
- Take a look at different requests for sizes as represented by nbytes

nbytes	7	15	16	17	32	33
nunits	2	2	2	3	3	4

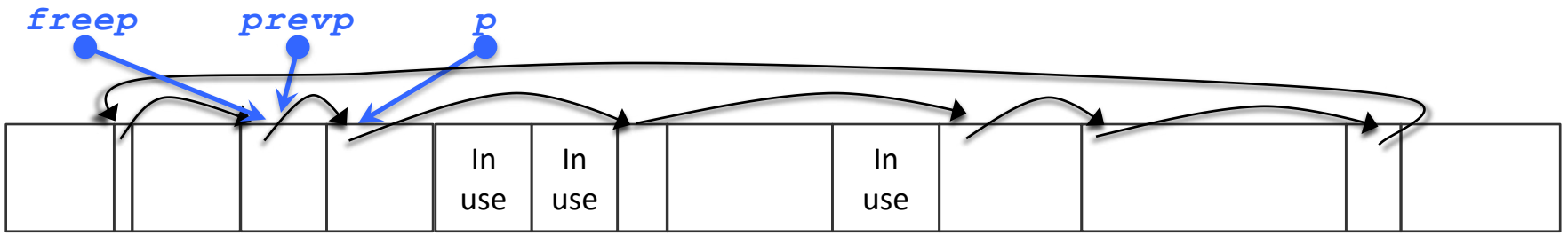
Section of malloc() Code

```
if (freep  == NULL) {
    freep = &base;
    base.size = 0;
    base.next = &base;
    if (morecore(nunits) == NULL)
        return NULL;    /* none left; give up */
    }
```



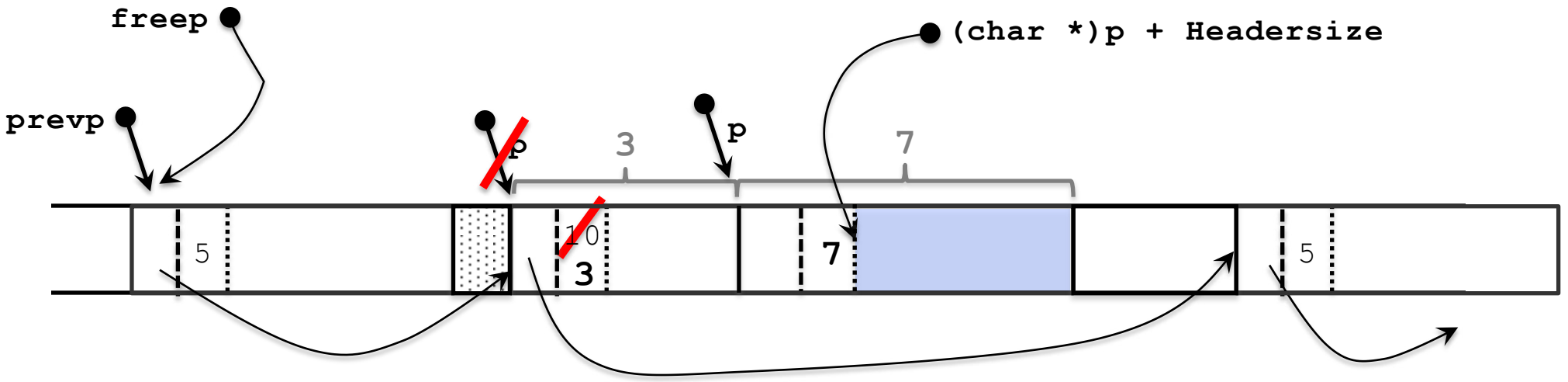
Section of malloc() Code

```
for (prevp = freep, p = prevp->next; p->size < nunits;
                                        prevp = p, p = p->next) {
      if (p == freep)
          if ((p = morecore(nunits)) == NULL)
              return NULL;    /* none left; give up */
  }
```



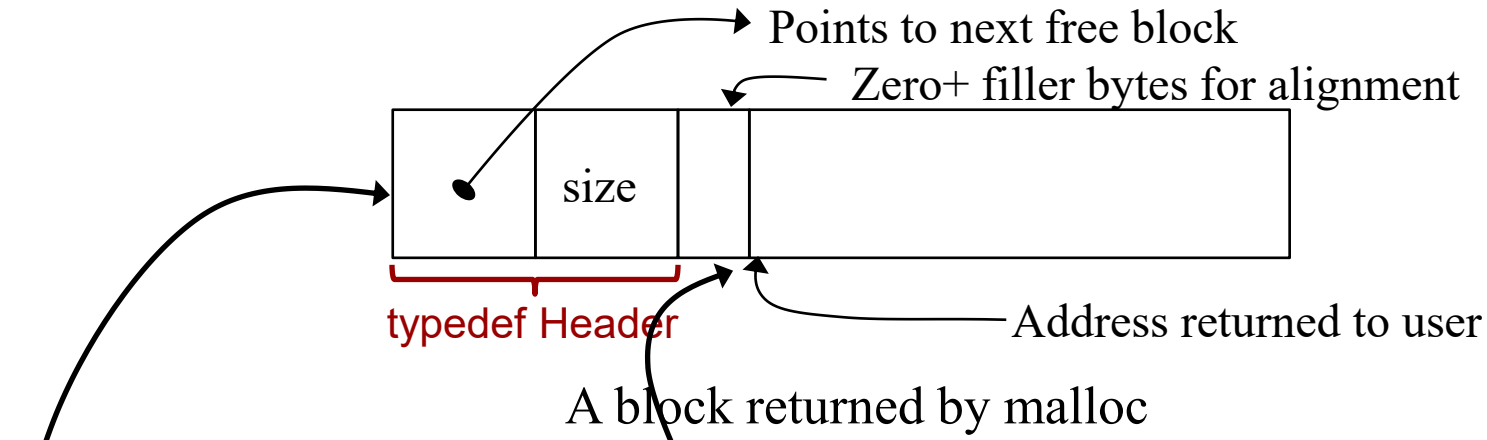
Section of malloc() Code

```
if (p->size == nunits)
    prevp->next = p->next;
else {
    p->size -= nunits;
    p = (char *)p + p->size * Headersize;
    p->size = nunits;
    }
freep = prevp;
return (char *)p + Headersize;
```



Let nunits = 7, and follow how the free block is identified and returned

Pointer Sent to free()



- If the variable

  `char *ap;`

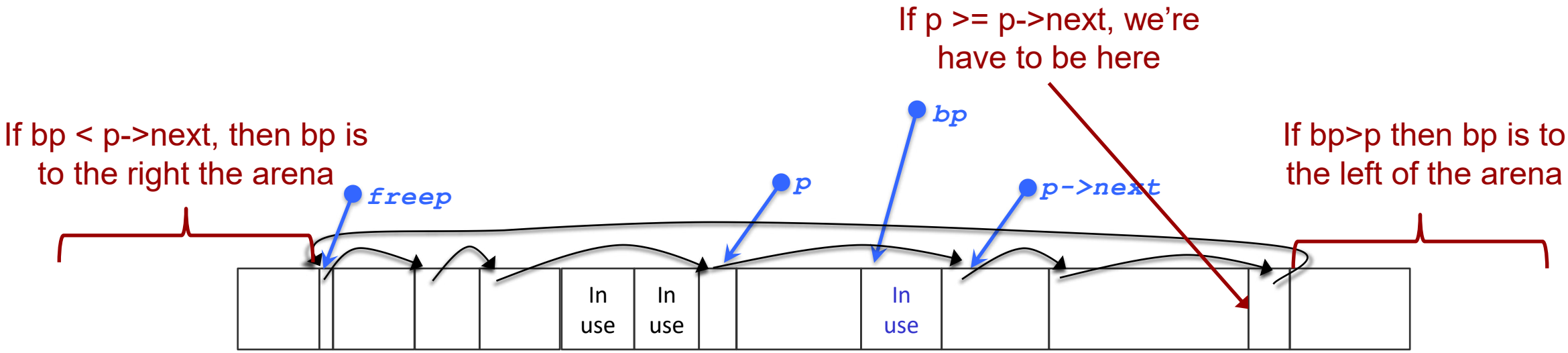
  points to the address returned to the user, where does

  `(Header *)((char *)ap - Headersize)`

  point?

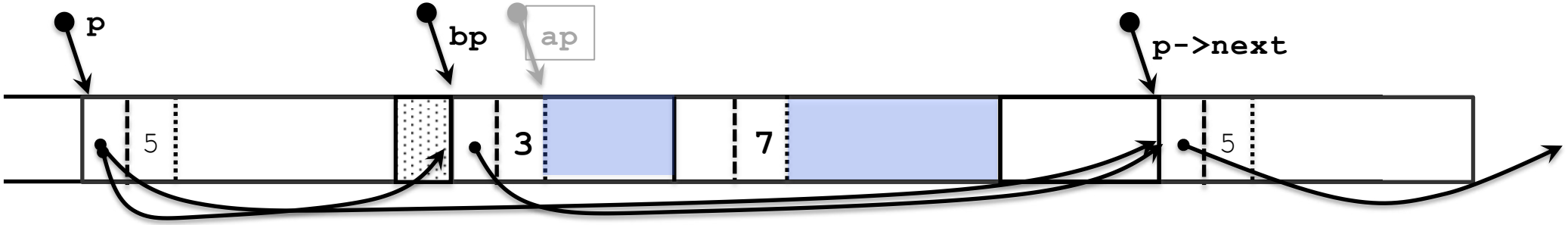
Section of free() Code

```
void _free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)(ap - Headersize);

    /* Find p such that the freed block is between p and p->next */
    for (p = freep; !(p < bp && bp < p->next); p = p->next)
        /* is *bp at either end of the arena? */
        if (p >= p->next && (bp > p || bp < p->next))
            break;
```



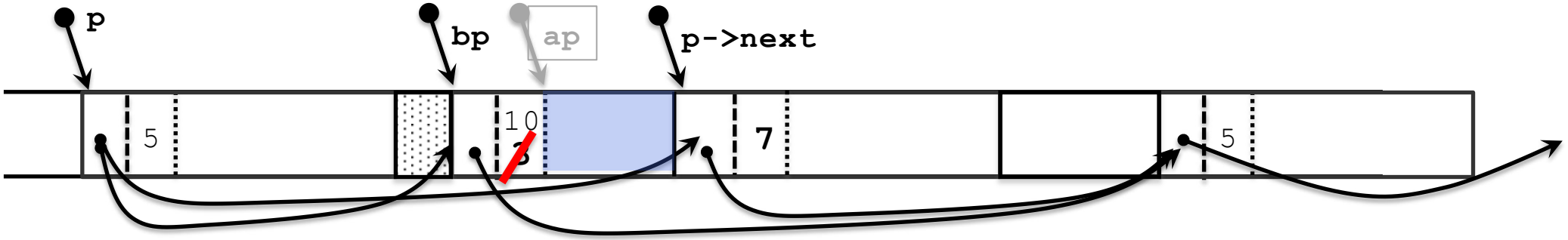
Free(): Below and Above In Use

```
 /* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == (char *)p->next) {
    bp->size += p->next->size;
    bp->next = p->next->next;
} else
    bp->next = p->next;

 /* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == (char *)bp) {
    /* add the freed block to the block at p */
    p->size += bp->size;
    p->next = bp->next;
} else
    p->next = bp;
freep = p;
```



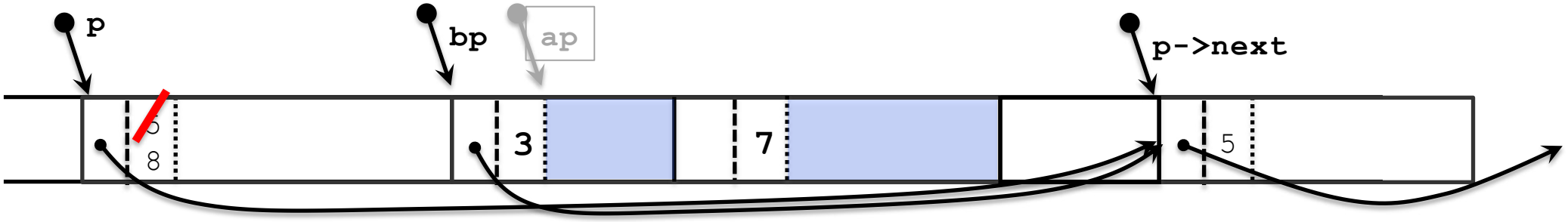
Free(): Below In Use

```
 /* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == p->next) {
    bp->size += p->next->size;
    bp->next = p->next->next;
} else
    bp->next = p->next;

 /* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == bp) {
    /* add the freed block to the block at p */
    p->size += bp->size;
    p->next = bp->next;
} else
    p->next = bp;
freep = p;
```



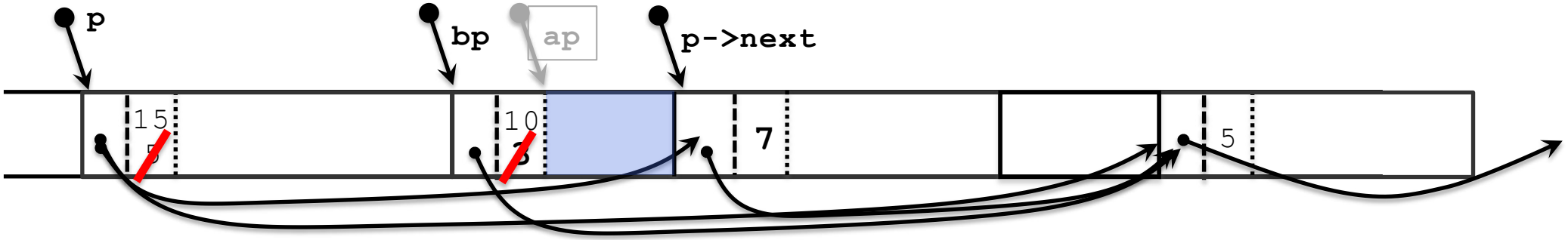
Free(): Above In Use

```
 /* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == p->next) {
    bp->size += p->next->size;
    bp->next = p->next->next;
} else
    bp->next = p->next;

 /* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == bp) {
    /* add the freed block to the block at p */
    p->size += bp->size;
    p->next = bp->next;
} else
    p->next = bp;
freep = p;
```



Free(): Above and Below Free

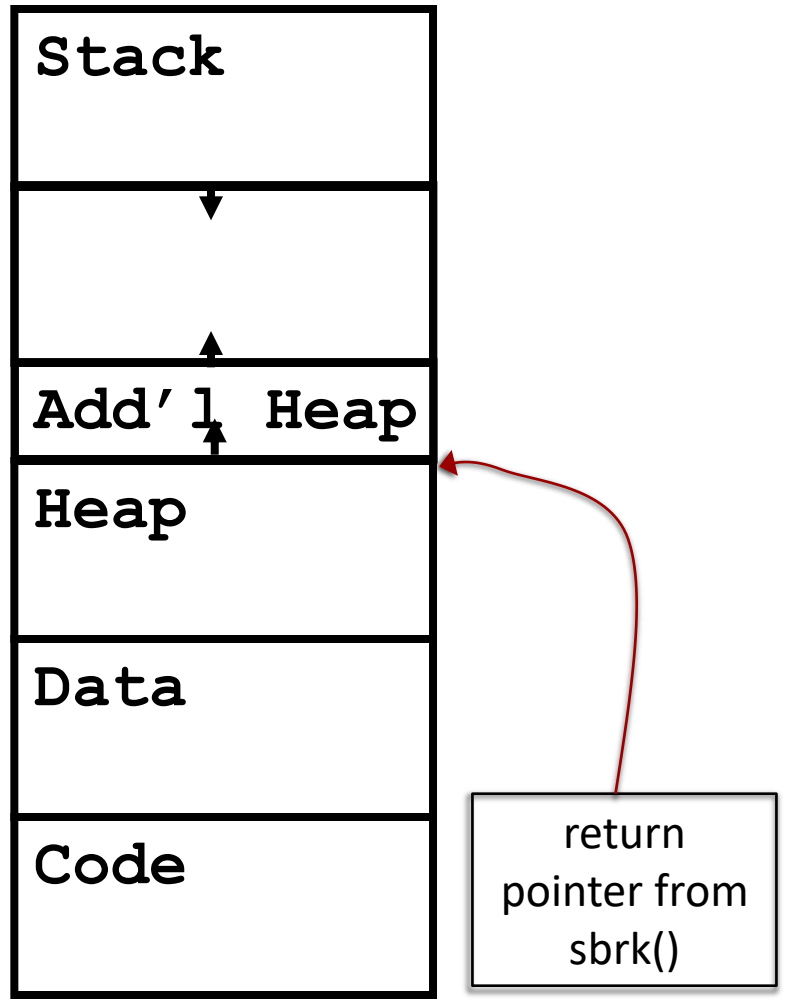
```
 /* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == p->next) {
    bp->size += p->next->size;
    bp->next = p->next->next;
} else
    bp->next = p->next;

 /* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == bp) {
    /* add the freed block to the block at p */
    p->size += bp->size;
    p->next = bp->next;
} else
    p->next = bp;
freep = p;
```



Section of morecore() Code 2

```
/* Don't ask for any less than NALLOC */
if (nu < NALLOC)
    nu = NALLOC;
cp = sbrk(nu * Headersize);
if (cp == (char *) -1) /* no space */
    return NULL;

/* Take the new block, add a header, and free() it */
up = (Header *) cp;
up->size = nu;
_free((char *)up + Headersize);
return freep;
```

- sbrk() is a Linux system call to obtain more memory.
- It works at the end of the data segment/heap (a.k.a. “the break”) to acquire at least n bytes more space from the OS
- It returns an aligned pointer to the new space or -1 if no space is available
- The -1 is an odd historical artifact



Question

After the execution of this code,

```
double *mp = &d[3];
char *xp = (char *)mp + 16;
```

to what memory address does *xp* point?

A. 0x10 bytes past the address of *d[3]*

B. 10 bytes past the beginning of *d*

C. 10 bytes before the beginning of *d*

D. 16 bytes before the beginning of *d*

Today's number is
16,661

Memory Management

- What are the equivalents of malloc and free in Java? Python?

- How can we “automatically” collect garbage?
  - Mark & Sweep Techniques
  - Reference Counting Techniques
  - And many, many others...