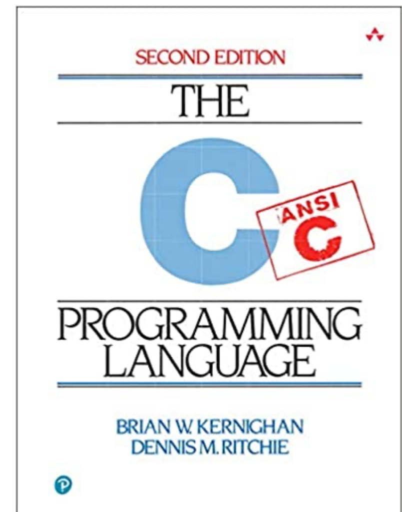
NTNU | Norwegian University of Science and Technology

Operating Systems

Discussion of PE3 – 4.03.2021

Michael Engel

Memory allocation

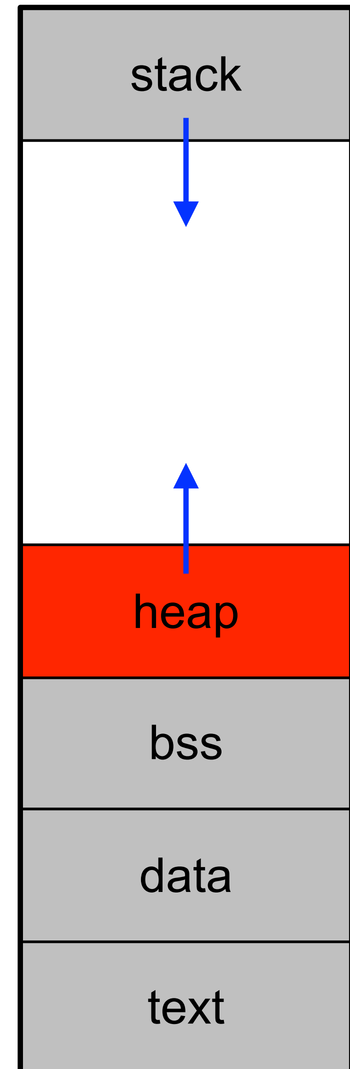


- Understanding how the heap is managed
  - Malloc: allocate memory
  - Free: deallocate memory
- K&R implementation (2nd edition, section 8.7)
  - Free list
    - Free block with header (pointer and size) and user data
    - Aligning the header with the largest data type
    - Circular linked list of free blocks
  - Malloc
    - Allocating memory in multiples of header size
    - Finding the first element in the free list that is large enough
    - Allocating more memory from the OS, if needed
  - Free
    - Putting a block back in the free list
    - Coalescing with adjacent blocks, if any

Memory layout: Heap

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

char* f(void)
{
  char* p;
  iSize = 8;
  p = malloc(iSize);
  return p;
}
```



Using malloc and free

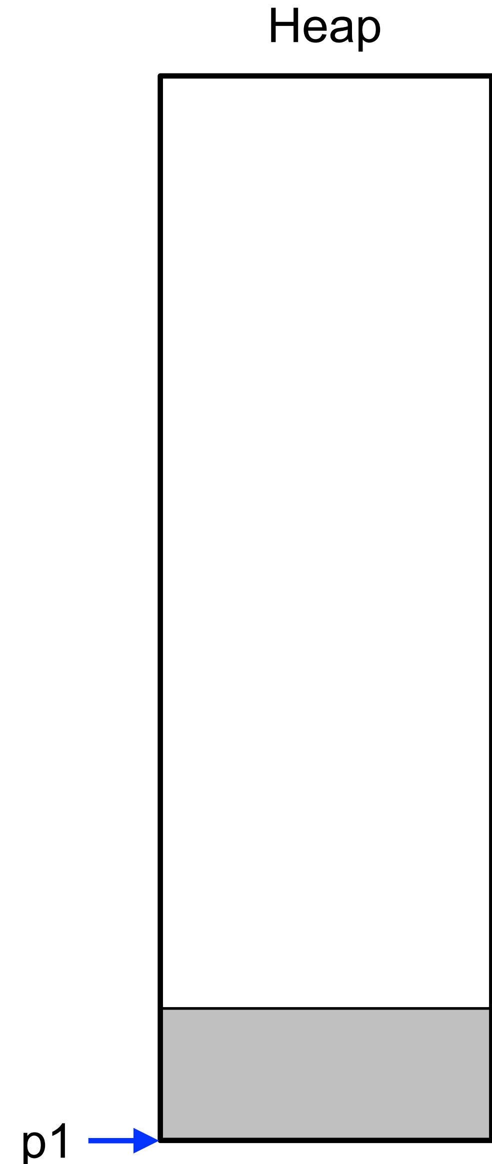
- Types
  - `void*`: generic pointer to any type
    (can be converted to other pointer types)
  - `size_t`: unsigned integer type returned by `sizeof()`
- `void *malloc(size_t size)`
  - Returns a pointer to space of size size
  - ... or `NULL` if the request cannot be satisfied
  - e.g., `int* x = (int *) malloc(sizeof(int))`
- `void free(void *p)`
  - Deallocate the space pointed to by the pointer `p`
  - Pointer `p` must be pointer to space previously allocated
  - Do nothing if `p` is `NULL`

Example heap allocations

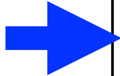
```
#include <stdlib.h>
void* malloc(size_t size);
void free(void *ptr);

...

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);
```



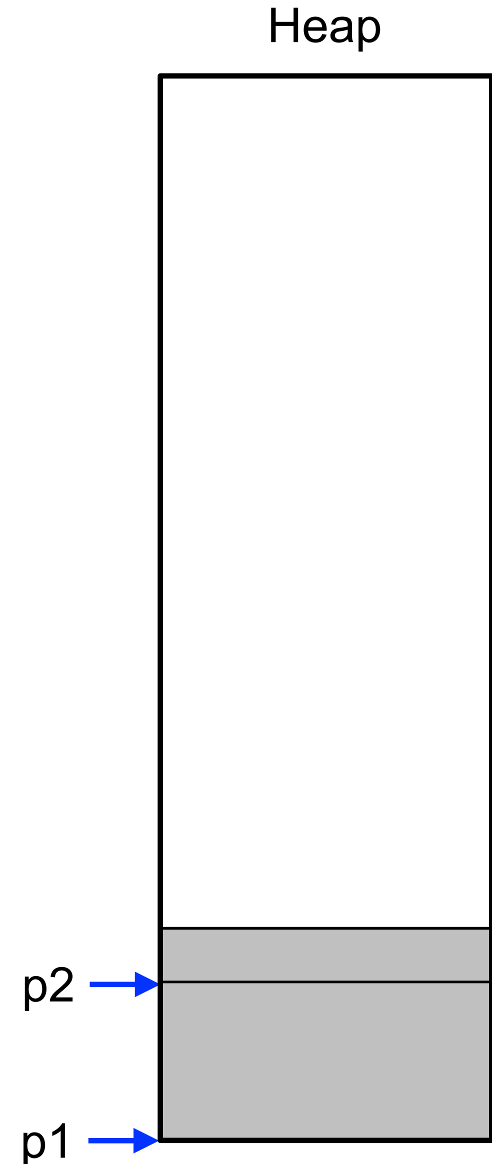
Example heap allocations



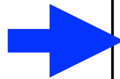
```
#include <stdlib.h>
void* malloc(size_t size);
void free(void *ptr);


…


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);
```



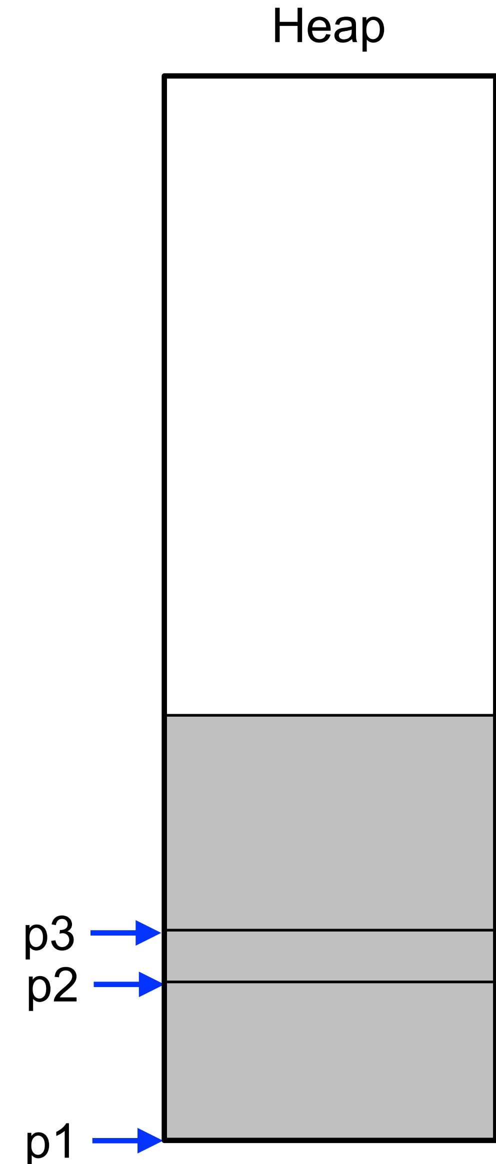
Example heap allocations



```
#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);
```

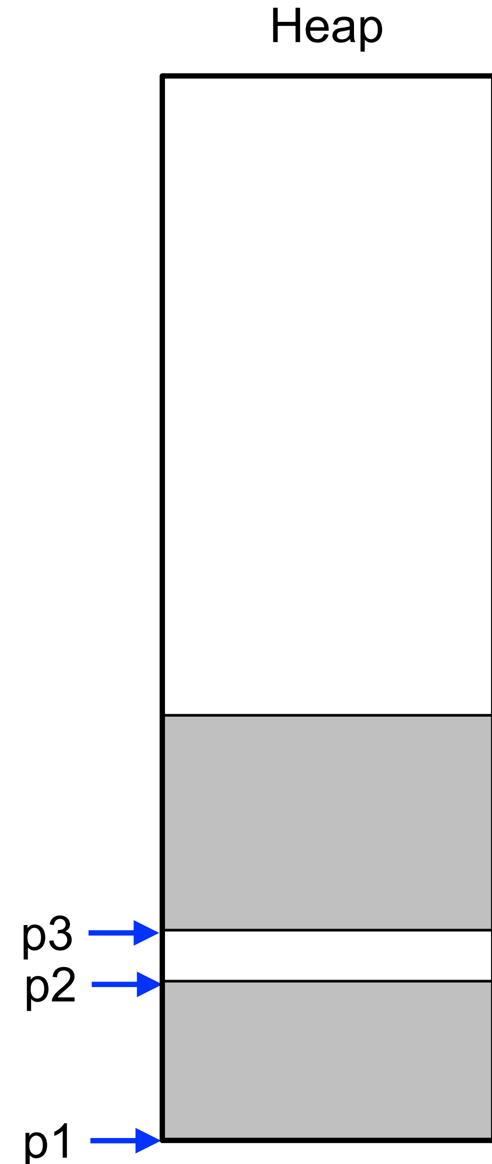


Example heap allocations

```c
#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);
```

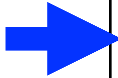


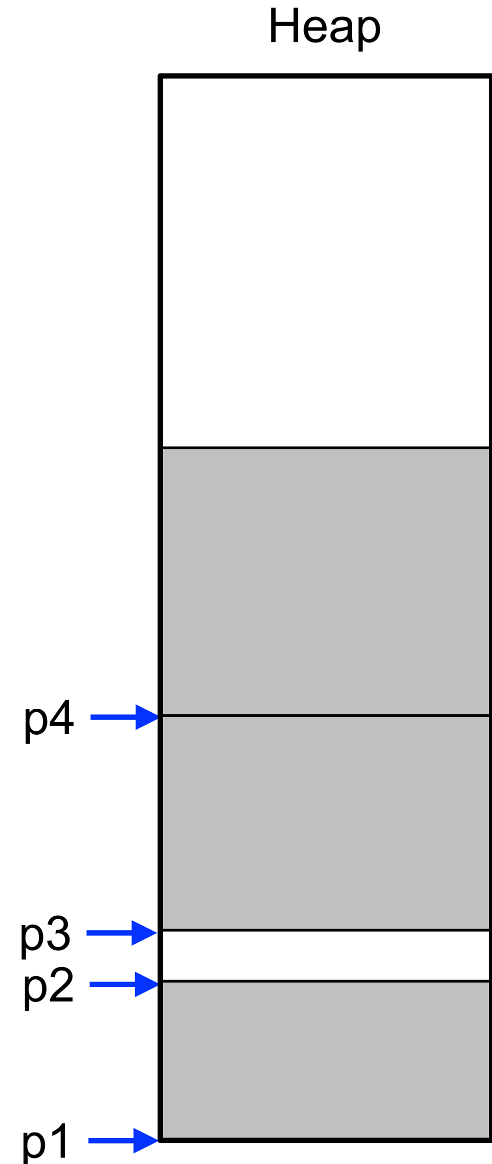
Example heap allocations

```
#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);
```



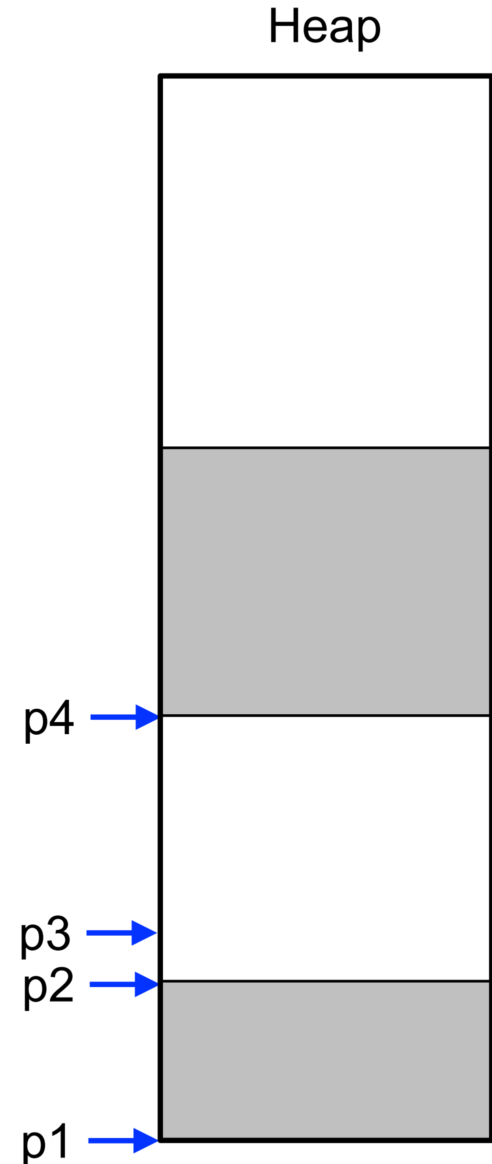


Example heap allocations

```
#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);
```

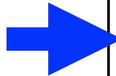


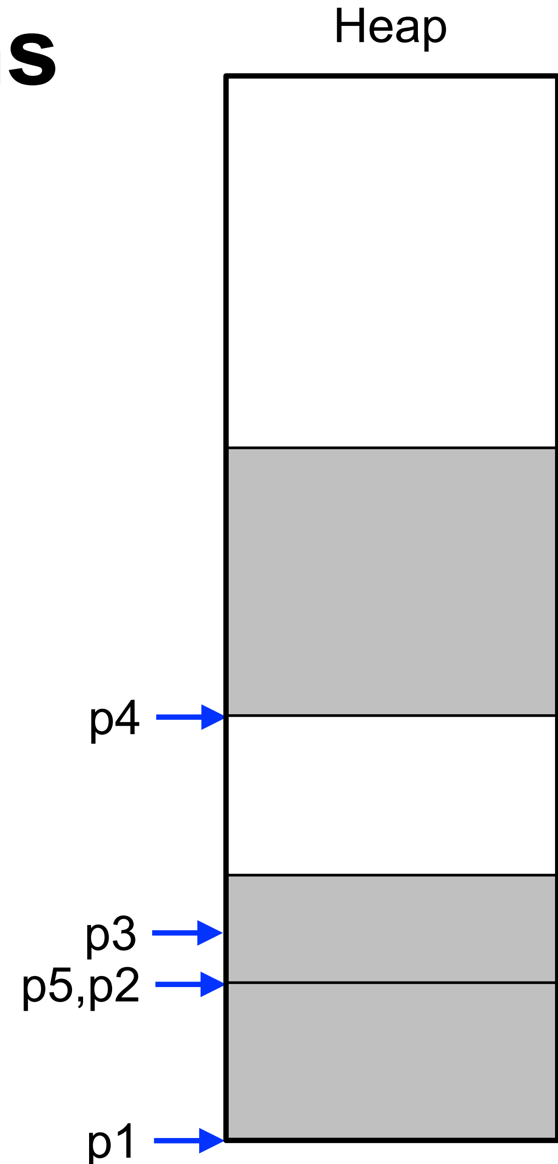
Example heap allocations

```
#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);
```



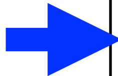


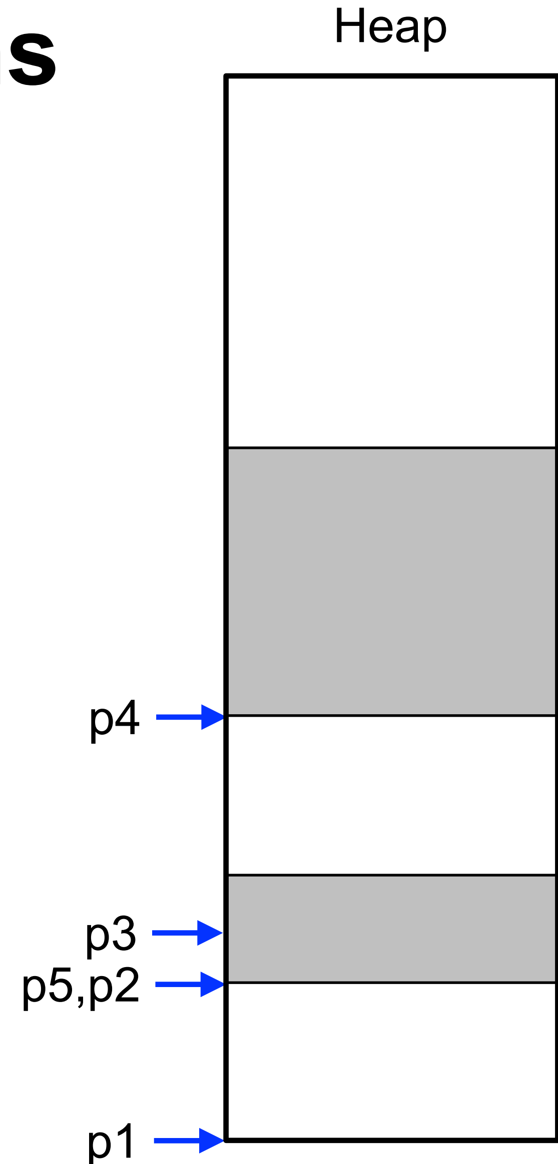
Example heap allocations

```
#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);
```



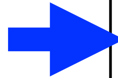


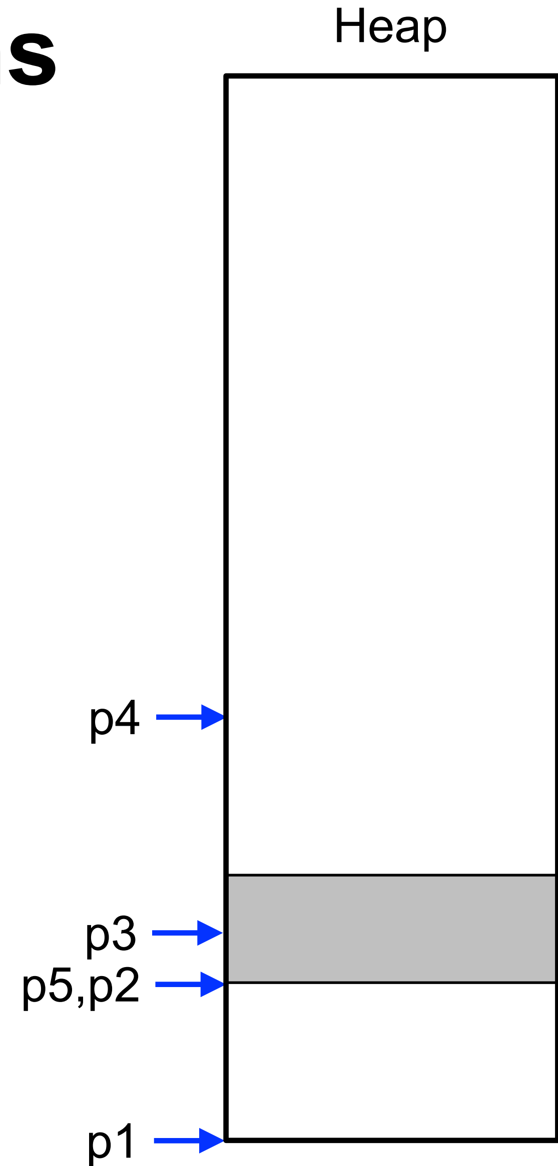
Example heap allocations

```
#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);
```



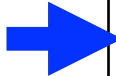


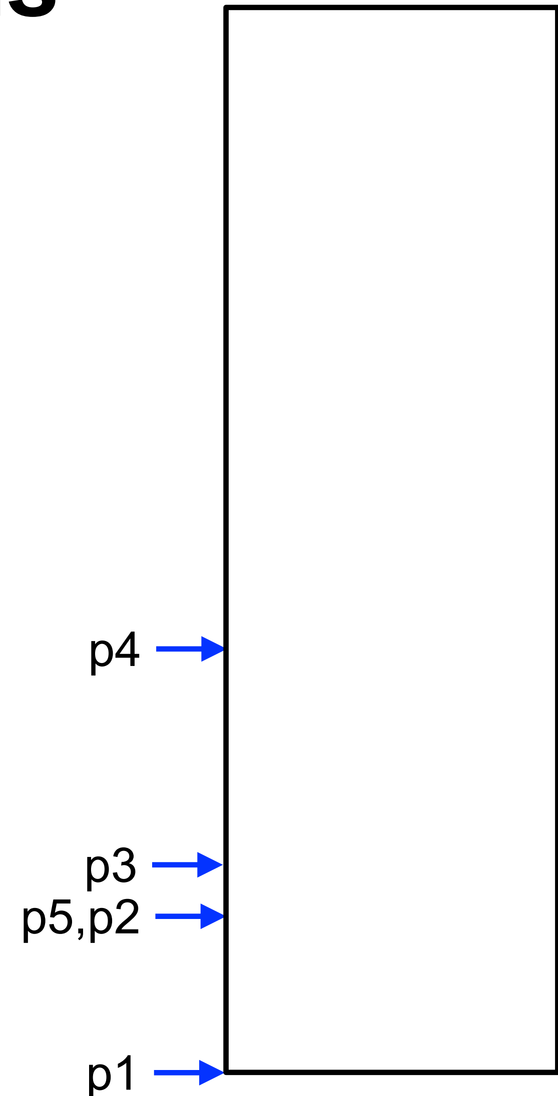
Example heap allocations

```c
#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);
```





How to use the heap, then?

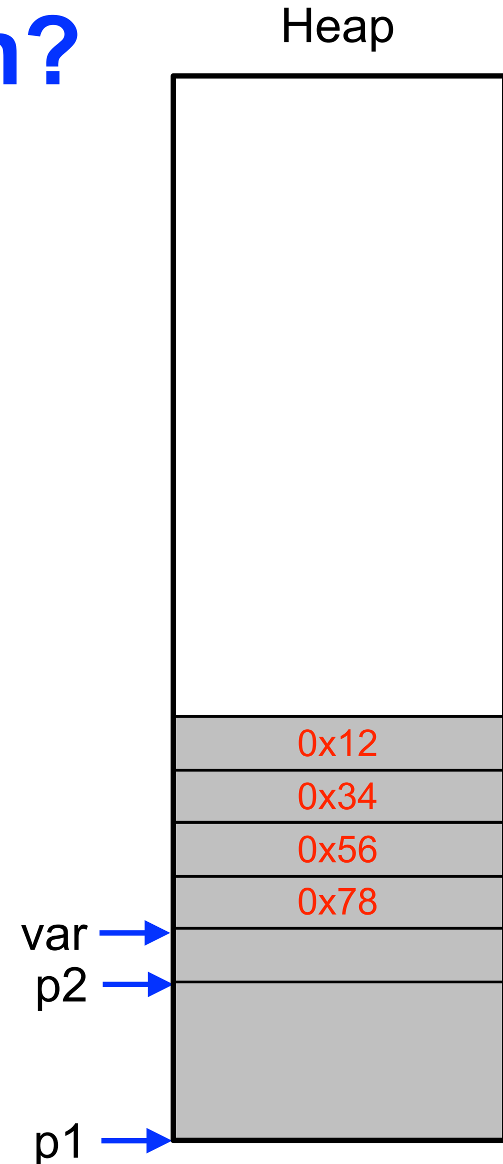
```
#include <stdlib.h>
void* malloc(size_t size);
void free(void *ptr);

…
int * var;
char* p1 = malloc(3);
char* p2 = malloc(1);
var = malloc(4);

// little endian byte order!
*var = 0x12345678;

// we no longer need the value
free(var);
```

This example does not consider our specific heap implementation that stores meta data inside the heap!



The biggest problem? *Pointer Arithmetics*

- We can "compute" using pointer and array identifiers:

```
char text[] = "quark";
char *c = text+1;
*c = 'w';                 /* "qwark" */
*(text+4) = 'b';          /* "qwarb" */
*(c-1) = 'z';             /* "zwarb" */
```

  - `text[4]` is another expression for `*(text+4)`
  - `text+1` can be written as `&(text[1])`
  - even `c[-1]` is possible instead of `*(c-1)`!

From the C crash course (slide 54 ff.)…

Pointer Arithmetics (2)

- This code outputs “quark” three times

```
char text[]=”quark”;

int i;
char *c;

for (i=0;i<7;i++)             /* normal array access */
   printf(“%c”,text[i]);

for (i=0;i<7;i++)             /* using pointer arithmetics */
   printf(“%c”,*(text+i));
                              /* more pointer arithmetics */
for (c=text;c<=&text[6];c++)
   printf(“%c”,*c);
```

p: pointer, ***s***: scalar value

```
   p+s   is equal to   &(p[s])
*(p+s)   is equal to     p[s]
   p++   is equal to     p=&(p[1])
```

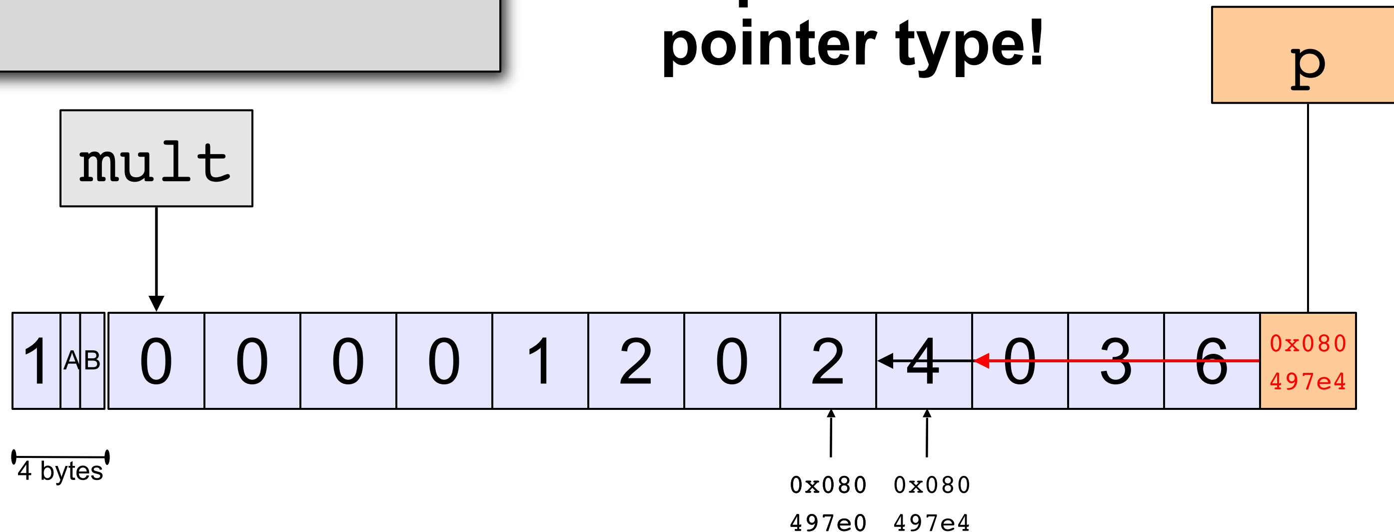
Pointer Arithmetics (3)

```
short int dummy = 1;
char bla='A',blb='B';
int mult[4][3] = { {0,0,0},
                   {0,1,2},
                   {0,2,4},
                   {0,3,6} };
int *p = &mult[2][1];

int main() {
    p++;
    return 0;
}
```

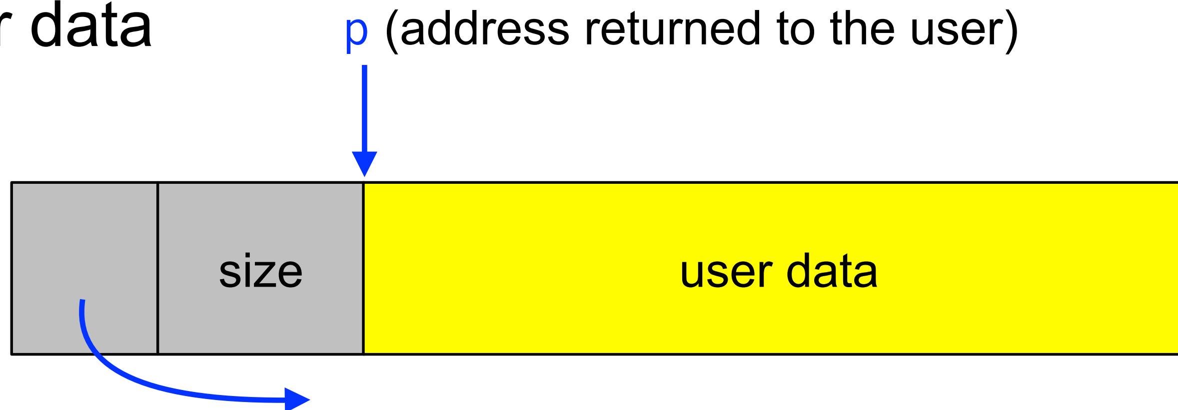
- Also works for arrays which are *not* of type `char`
- since *p*+*s* = `&(p[s])`, p is *not* **incremented** by 1 (Byte), but rather **by 4**!

→ **Address difference depends on the pointer type!**



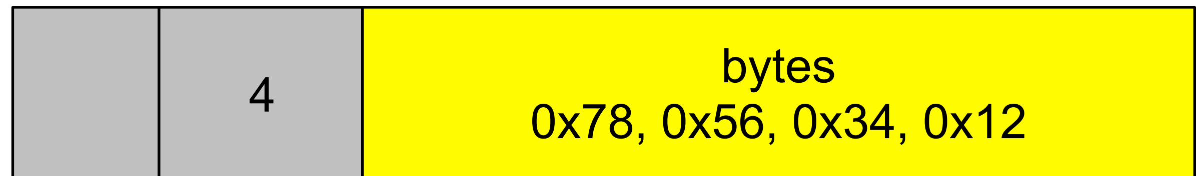
Memory blocks: pointer, size, data

- Representation of blocks in memory
  - pointer to the next block
  - size of the block
  - user data



Four our example

```
var = malloc(4);
*var = 0x12345678;
```



Free block: memory alignment

- Define a structure s for the header
  - Pointer to the next free block (`ptr`)
  - Size of the block (`size`)
- To simplify memory alignment
  - Make all memory blocks a multiple of the header size
  - Ensure header is aligned with largest data type (e.g., `long`)
- Union: C technique for forcing memory alignment
  - Variable that may hold objects of different types and sizes
  - Made large enough to hold the largest data type, e.g.,

```
union Tag {
    int   ival;
    float fval;
    char *sval;
} u;
```

Free block: memory alignment

```
/* align to long boundary */
typedef long Align;

union header { /* block header */
   struct {
     union header *ptr;
     unsigned size;
   } s;
   Align x;     /* Force alignment */
};

typedef union header Header;
```

In fact, "x" is never used, it's just there to enforce the alignment of struct s!

much more information about alignment and padding:
http://www.catb.org/esr/structure-packing/

Allocate memory in units

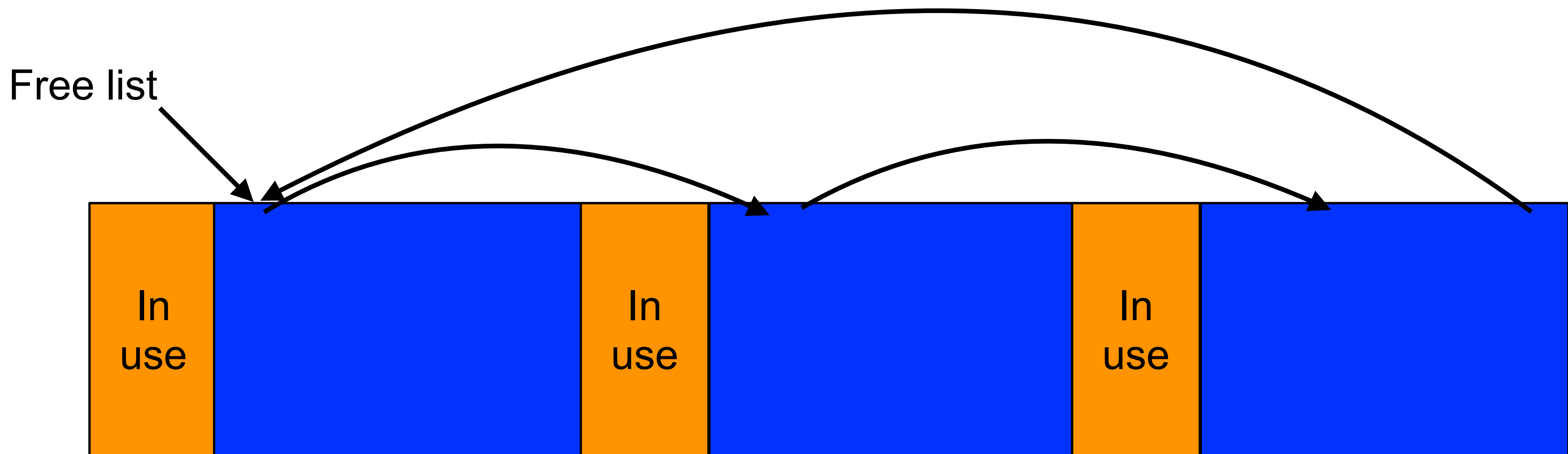
- Keep memory aligned
  - Requested size is rounded up to multiple of header size
- Rounding up when asked for nbytes
  - Header has size `sizeof(Header)`
  - Round: `(nbytes + sizeof(Header) – 1)/sizeof(Header)`
- Allocate space for user data, plus the header itself

```
void *malloc(unsigned int nbytes) {
   unsigned int nunits;

   nunits = (nbytes + sizeof(Header) - 1) /
                 sizeof(Header) + 1;
   …
}
```

Free List: Circular Linked List

- Free blocks, linked together
  - Example: circular linked list
- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks



Allocation algorithms

- 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
- Which block to return?
  - First-fit algorithm
    - Keep a linked list of free blocks
    - Search for the ***first*** one that is big enough
  - Best-fit algorithm
    - Keep a linked list of free blocks
    - Search for the ***smallest*** one that is big enough
    - Helps avoid fragmenting the free memory

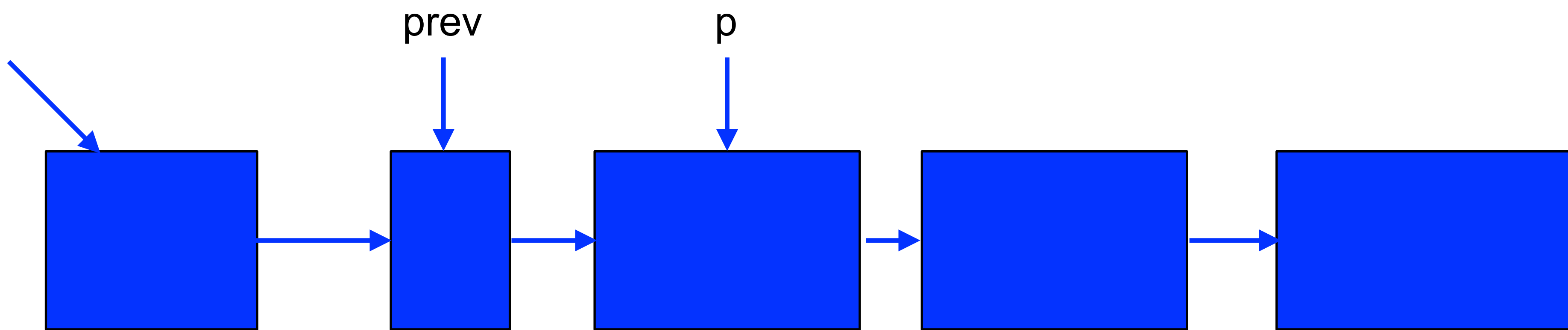
3a. Memory management

- The file mymalloc.c contains a C program skeleton for a heap allocator. To keep things a bit more simple, the heap is implemented here as a simple array of bytes with 64 kB.
- In this exercise, you will implement your own heap memory management functions `void *mymalloc(int size)` to allocate a block of memory of the given size from the heap and void `myfree(void *p)` to release the block of memory pointed to by the pointer parameter
- Example code:

```
void *p;
p = mymalloc(42);
if (p != (void *)0) {
  // do something
  myfree(p);
} else {
  printf("mymalloc failed!\n");
}
```

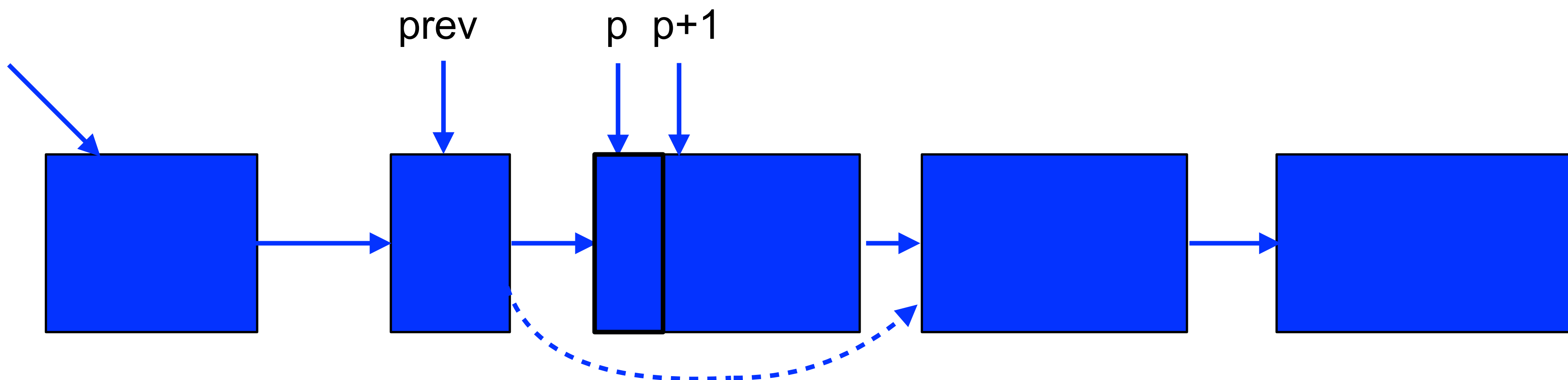
3a. Malloc: First-fit algorithm

- Start at the beginning of the list
- Sequence through the list
  - Keep a pointer to the previous element
- Stop when reaching the first block that is big enough
  - Patch up the list
  - Return a block to the user



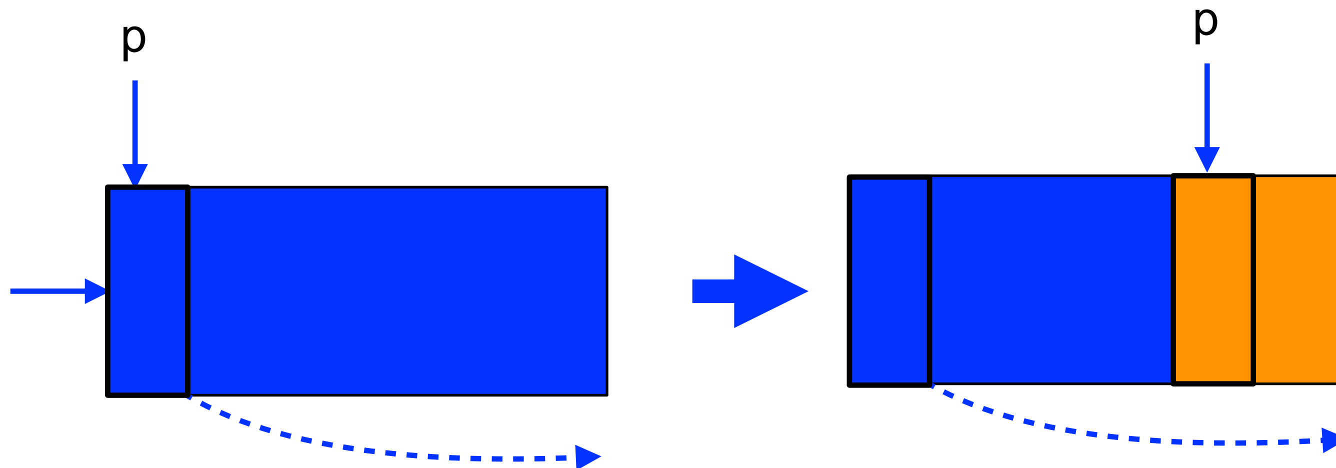
Case 1: a perfect fit

- Suppose the first fit is a perfect fit
  - Remove the element from the list
  - Link the previous element with the next element
    ```
    prev->s.ptr = p->s.ptr;
    ```
  - Return the current element to the user (skipping header)
    ```
    return (void *)(p+1);
    ```



Case 2: block is too big

- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list
    ```
    prev->s.size -= nunits;
    ```
  - Allocate the second block to the user
    ```
    p += p->s.size;
    p->s.size = nunits;
    ```



Combining the two cases

```
prevp = freep;  /* start at the beginning */

for (p = prevp->s.ptr; /* */ ; prevp = p, p = p->s.ptr) {

  if (p->s.size >= nunits) {

    if (p->s.size == nunits) { /* fit */
       prevp->s.ptr = p->s.ptr;

    } else {                    /* too big, split in two */
       p->s.size -= nunits;     /* #1 */
       p += p->s.size;          /* #2 */
       p->s.size = nunits;      /* #2 */
    }

    return (void *)(p+1);
  }
}
```

Start of the free list

- Benefit 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
  - Optimization: make head be where last block was found

```
prevp = freep;  /* start at the beginning */

for (p = prevp->s.ptr; /* */ ; prevp = p, p = p->s.ptr) {

  if (p->s.size >= nunits) {

    /* Do stuff from the previous slide! */
    …
    freep = prevp;    /* move the head! */
    return (void *)(p+1);
  }
}
```

No block is big enough!

- Cycling completely through the list
  - Check if the “for” loop returns back to the head of the list

```
prevp = freep;  /* start at the beginning */

for (p = prevp->s.ptr; /* */ ; prevp = p, p = p->s.ptr) {

  if (p->s.size >= nunits) {

    /* Do stuff from the previous slide! */
    …
  }
  if (p == freep)       /* wrapped around! */
    Handle the error…
}
```

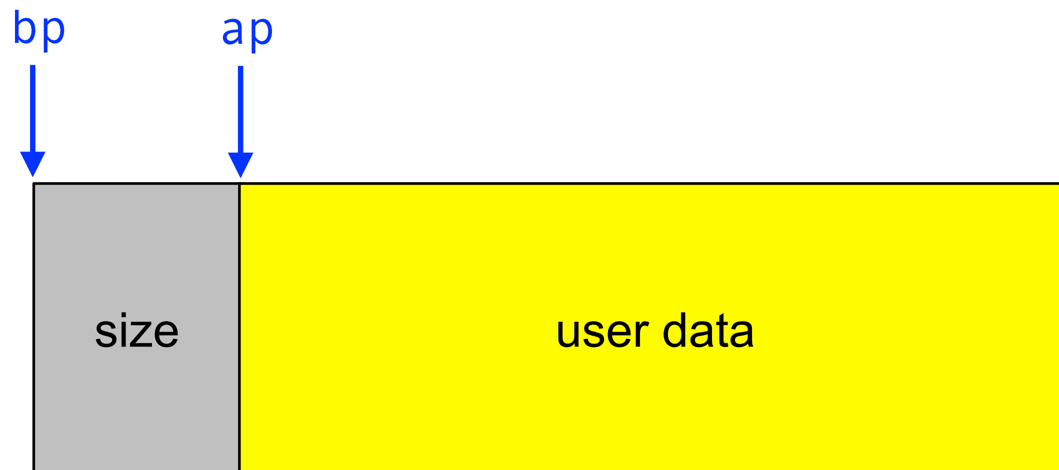
What to do when you run out

- In our example: **fail**, because we have a fixed memory area
- In general: ask the operating system for additional memory
  - …and insert the new chunk into the free list
  - …and then try again, this time successfully
- Operating system dependent
  - on Unix: `sbrk(2)` system call

```
if (p == freep)        /* wrapped around! */
  if ((p = sbrk(nunits)) == NULL)
    return NULL;       /* could not get more memory from */
                       /* the operating system, so fail… */
```

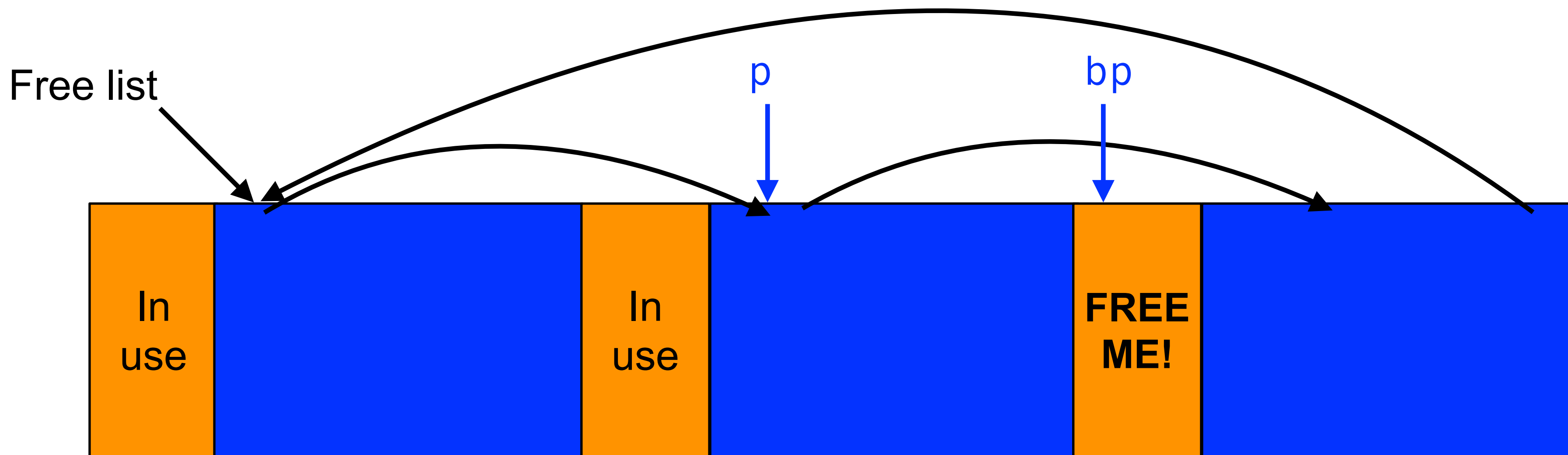
3b. Free

- User passes a pointer to the memory block
  - `void free(void *ap);`
- Free function inserts block into the list
  - Identify the start of entry: `bp = (Header *)ap - 1;`
  - Find the location in the free list
  - Add to the list, coalescing entries, if needed



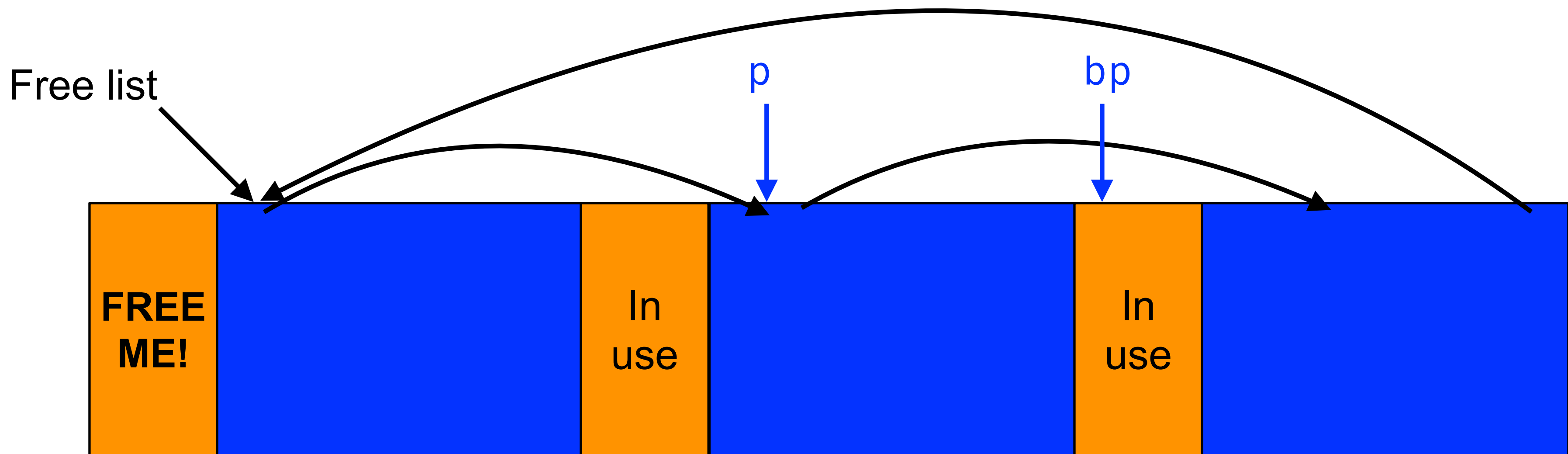
Scanning free list for the spot

- Start at the beginning: `p = freep;`
- Sequence through the list: `p = p->s.ptr;`
- Stop at last entry before the to-be-freed element
  - `(bp > p) && (bp < p->s.ptr);`



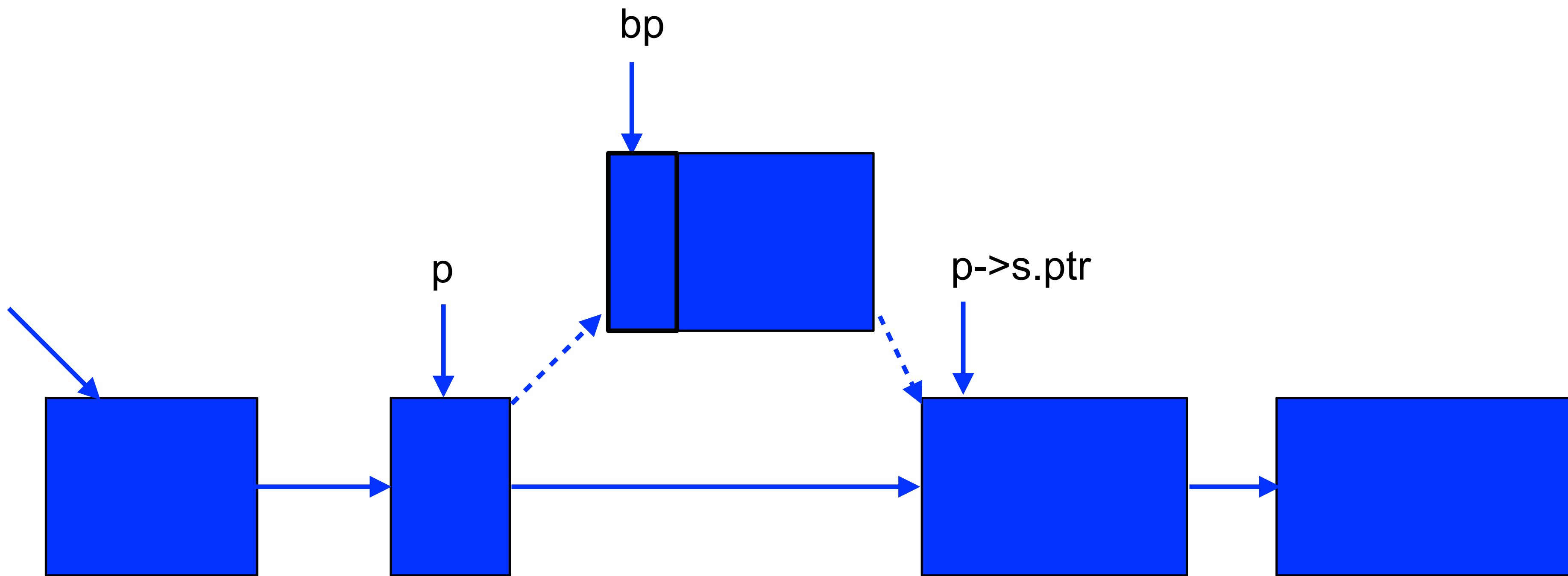
Corner cases: beginning or end

- Check for wrap-around in memory:
  `p >= p->s.ptr;`
- See if to-be-freed element is located there:
  `(bp > p) || (bp < p->s.ptr)`



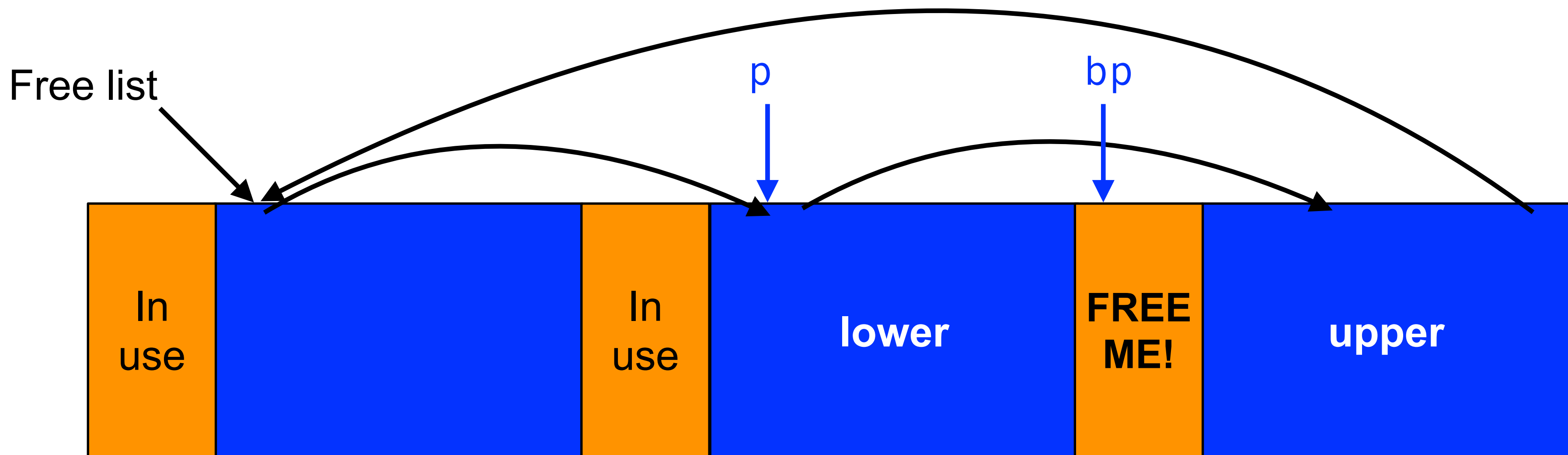
Inserting into free list

- New element to add to free list: `bp`
- Insert in between `p` and `p->s.ptr`:
  - `bp->s.ptr = p->s.ptr;`
  - `p->s.ptr = bp;`



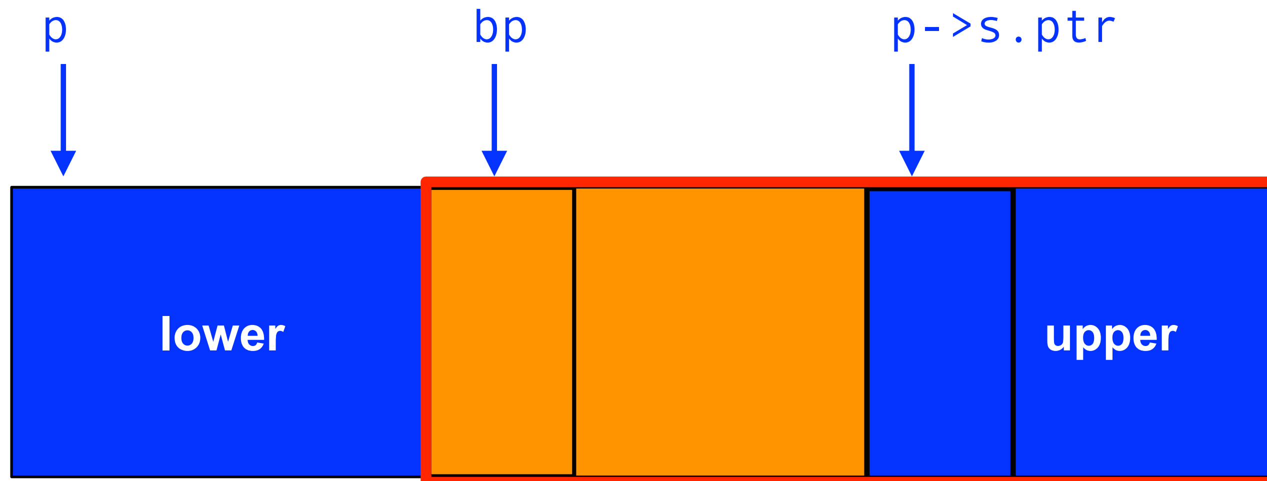
Coalescing with neighbors

- Scanning the list finds the location for inserting
  - Pointer to to-be-freed element: bp
  - Pointer to previous element in free list: p
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors



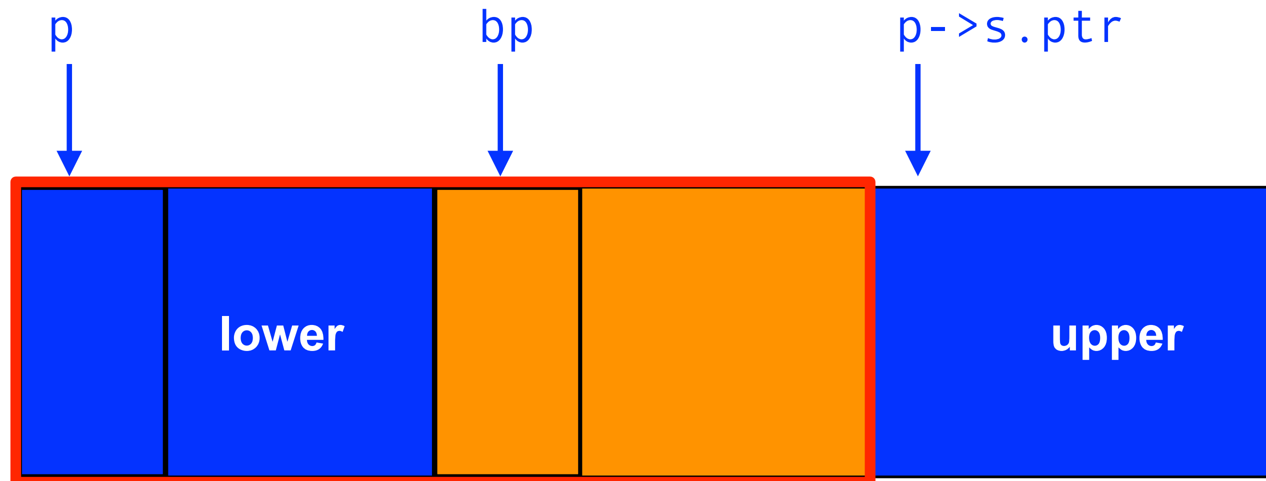
Coalescing with upper neighbor

- Check if next part of memory is in the free list:
  - `if (bp + bp->s.size == p->s.ptr)`
- If so, make both into bigger block:
  - Larger size: `bp->s.size += p->s.ptr->s.size;`
  - Copy next pointer: `bp->s.ptr = p->s.ptr->s.ptr;`
- Else, simplify point to the next free element:
  - `bp->s.ptr = p->s.ptr;`



Coalescing with lower neighbor

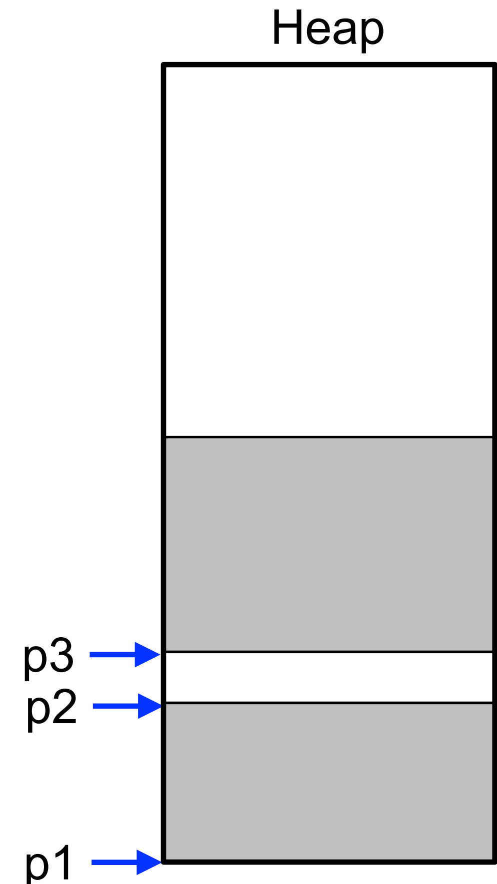
- Check if previous part of memory is in the free list:
  - `if (p + p->s.size == bp)`
- If so, make both into bigger block:
  - Larger size: `p->s.size += bp->s.size;`
  - Copy next pointer: `p->s.ptr = bp->s.ptr;`



3c. Design and implement test cases

- The test cases should be implemented in the main function. Describe which situations can occur and what you do to check for the correct behavior of your memory allocator.

```
void* mymalloc(size_t size);
void  myfree(void *ptr);
…
char* p1 = mymalloc(3);
char* p2 = mymalloc(1);
char* p3 = mymalloc(4);
myfree(p2);
char* p4 = mymalloc(6);
myfree(p3);
char* p5 = mymalloc(2);
myfree(p1);
myfree(p4);
myfree(p5);
```



3c Design and implement test cases

- Test interesting corner cases:
  - Allocate more memory than available
  - Try to free a non-allocated block
    - Special case: free a previously freed block
  - Create allocation with only small gaps left using alternating malloc and free calls, then try to allocate a block that doesn't fit
  - Check coalescing with upper, lower or both neighbors
  - Malloc the block at the start of the free list
  - …more ideas?

Conclusions

- Elegant simplicity of K&R malloc and free
  - Simple header with pointer and size in each free block
  - Simple 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
- Testing
  - Complete test coverage is not that simple…