The Design Problem

- Since the only thing the allocator can work with is the collection of free blocks, the management of this free space (aka the free list) is the fundamental design problem.

The Design Problem

- Since the only thing the allocator can work with is the collection of free blocks, the management of this free space (aka the free list) is the fundamental design problem.
- Four design dimensions.
    - Free list organization: How to keep track of the free blocks.
    - Free block selection: How to pick an appropriate free block to satisfy the current allocation request from the mutator.
    - Free block splitting: What to do with the remainder of a free block after allocation.
    - Free block coalescing: How to handle an allocated block that is being returned to the free list.

The Design Problem

- Since the only thing the allocator can work with is the collection of free blocks, the management of this free space (aka the free list) is the fundamental design problem.
- Four design dimensions.
    - Free list organization: How to keep track of the free blocks.
    - Free block selection: How to pick an appropriate free block to satisfy the current allocation request from the mutator.
    - Free block splitting: What to do with the remainder of a free block after allocation.
    - Free block coalescing: How to handle an allocated block that is being returned to the free list.
- The “free list” is conceptual.
    - Everything is just a byte array, and the allocator overlays a particular interpretation of these bytes.
    - Very low-level. No conventional abstractions to help us out.

Huge Design Space

- Many policy choices for each dimension, interacting in complicated ways.

Huge Design Space

- Many policy choices for each dimension, interacting in complicated ways.
  - Free list organization: How to keep track of the free blocks.
    - Implicit, explicit, binned.
    - Singly- or doubly-linked.
    - Null-terminated or circular.
    - Returned blocks maintained in LIFO, address-sorted, or size-sorted order.

Huge Design Space

- Many policy choices for each dimension, interacting in complicated ways.
  - Free list organization: How to keep track of the free blocks.
    - Implicit, explicit, binned.
    - Singly- or doubly-linked.
    - Null-terminated or circular.
    - Returned blocks maintained in LIFO, address-sorted, or size-sorted order.
  - Free block selection: How to pick an appropriate free block to satisfy the current allocation request from the mutator.
    - Best-fit, worst-fit, first-fit, next-fit.

Huge Design Space

- Many policy choices for each dimension, interacting in complicated ways.
  - Free list organization: How to keep track of the free blocks.
    - Implicit, explicit, binned.
    - Singly- or doubly-linked.
    - Null-terminated or circular.
    - Returned blocks maintained in LIFO, address-sorted, or size-sorted order.
  - Free block selection: How to pick an appropriate free block to satisfy the current allocation request from the mutator.
    - Best-fit, worst-fit, first-fit, next-fit.
  - Free block splitting: What to do with the remainder of a free block after allocation.
    - No split, always split, threshold split.
    - Allocate block from bottom or top of free block.

Huge Design Space

- Many policy choices for each dimension, interacting in complicated ways.
  - Free list organization: How to keep track of the free blocks.
    - Implicit, explicit, binned.
    - Singly- or doubly-linked.
    - Null-terminated or circular.
    - Returned blocks maintained in LIFO, address-sorted, or size-sorted order.
  - Free block selection: How to pick an appropriate free block to satisfy the current allocation request from the mutator.
    - Best-fit, worst-fit, first-fit, next-fit.
  - Free block splitting: What to do with the remainder of a free block after allocation.
    - No split, always split, threshold split.
    - Allocate block from bottom or top of free block.
  - Free block coalescing: How to handle an allocated block that is being returned to the free list.
    - Immediate, deferred, none.

The K&R Allocator

- From K&R2e, §8.7.
  - Short, portable, performant.
- Very few assumptions.
  - Does not assume that successive calls to sbrk() return contiguous memory in process's address space.
  - Only non-portability is the assumption that pointers to different blocks returned by `sbrk()` can be meaningfully compared.

The K&R Allocator: Design Choices

- Free list organization: How to keep track of the free blocks.
  - Implicit, <u>explicit</u>, binned.
  - <u>Singly-</u> or doubly-linked.
  - Null-terminated or <u>circular</u>.
  - Returned blocks maintained in LIFO, <u>address-sorted</u>, or size-sorted order.
- Free block selection: How to pick an appropriate free block to satisfy the current allocation request from the mutator.
  - Best-fit, worst-fit, <u>first-fit</u>, next-fit.
- Free block splitting: What to do with the remainder of a free block after allocation.
  - No split, <u>always split</u>, threshold split.
  - Allocate block from bottom or <u>top</u> of free block.
- Free block coalescing: How to handle an allocated block that is being returned to the free list.
  - <u>Immediate</u>, deferred, none.

The K&R Allocator: Data Structures

```c
typedef long Align; /* for alignment to long boundary */

union header { /* block header: */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size; /* size of this block */
    } s;
    Align x; /* force alignment of blocks */
};

typedef union header Header;

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

The K&R Allocator: `malloc()`

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

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) { /* no free list yet */
          base.s.ptr = freep = prevp = & base;
          base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) { /* big enough */
            if (p->s.size == nunits) /* exactly */
                prevp->s.ptr = p->s.ptr;
            else { /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
    }
    if (p == freep) /* wrapped around free list */
        if ((p = morecore(nunits)) == NULL)
            return NULL; /* none left */
   }
}
```

The K&R Allocator: `morecore()`

```c
#define NALLOC 1024 /* minimum #units to request */

/* morecore: ask system for more memory */
static Header *morecore(unsigned nu) {
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu*sizeof(Header));
    if (cp == (char *) -1) /* no space at all */
        return NULL;
    up = (Header *) = cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

The K&R Allocator: `free()`

```c
/* free: put block ap in free list */
void free(void *ap) {
    Header *bp, *p;

    bp = (Header *)ap - 1; /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p.s.ptr && (bp > p || bp < p->s.ptr))
            break; /* freed block at start or end of arena */

    if (bp + bp->s.size == p->s.ptr) { /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) { /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```