Goals for Today

Understand how the heap is managed

- Where the memory comes from
- How the heap allocator hands out memory

Explore design tradeoffs and optimizations of heap allocators

- What makes a good allocator?

Overview: Heap Allocation

Compared to `new` and `delete`

- No compiler support (type checking)
- Less convenient (no constructors, initialization)

`malloc` written in C

- Gets memory from OS in large chunks (pages)
- Tracks in-use and free blocks (metadata)
- Divides up heap between metadata and payload

Goals of a Heap Allocator

Correctness (non-negotiable)

Can't hand out blocks that overlap each other

Throughput

Handle allocation/free requests quickly

Utilization

Pack data tightly in heap; don't waste space

Challenges of a Heap Allocator

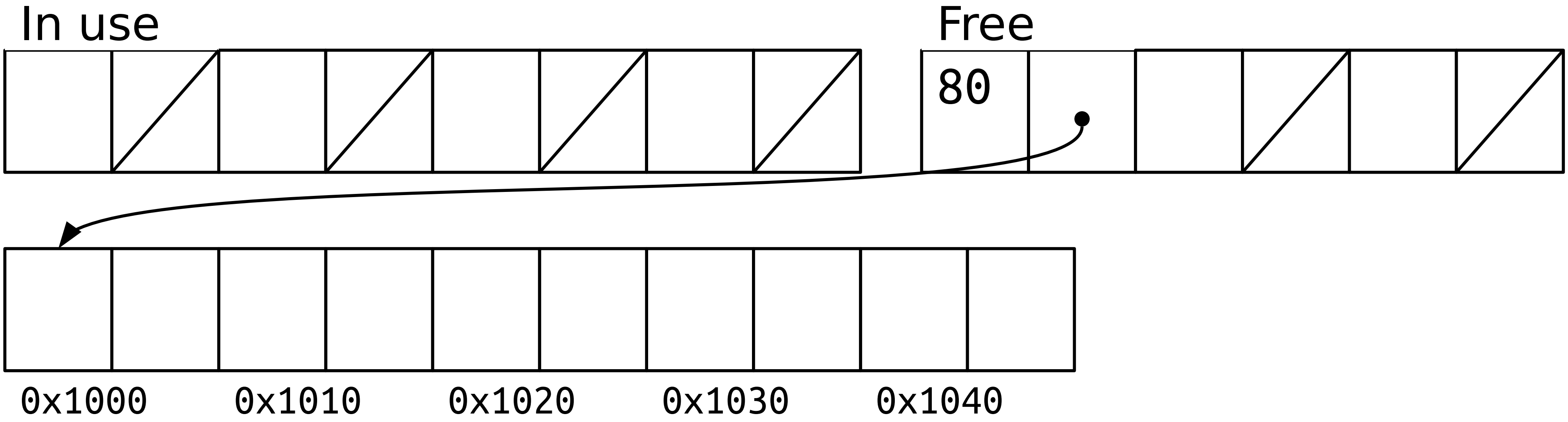
Would be easy if...

- Allocations were LIFO (like the stack), and/or
- We could predict future allocations/frees

In reality...

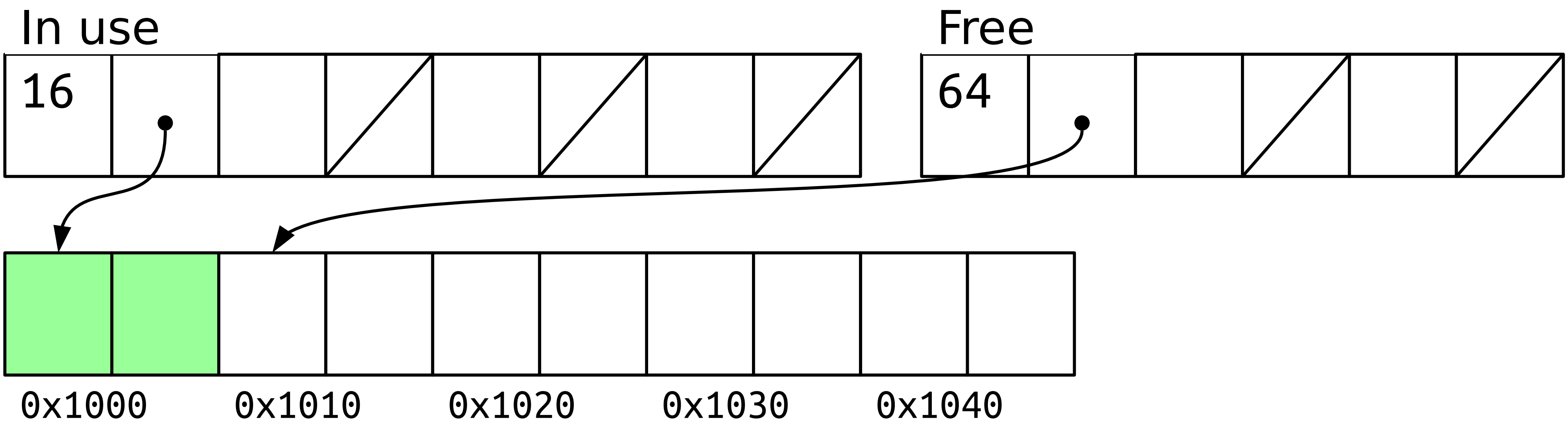
- Blocks have wildly varying lifespans
- Blocks of many different sizes
- Can't "take back" a pointer after handing it out

First Implementation



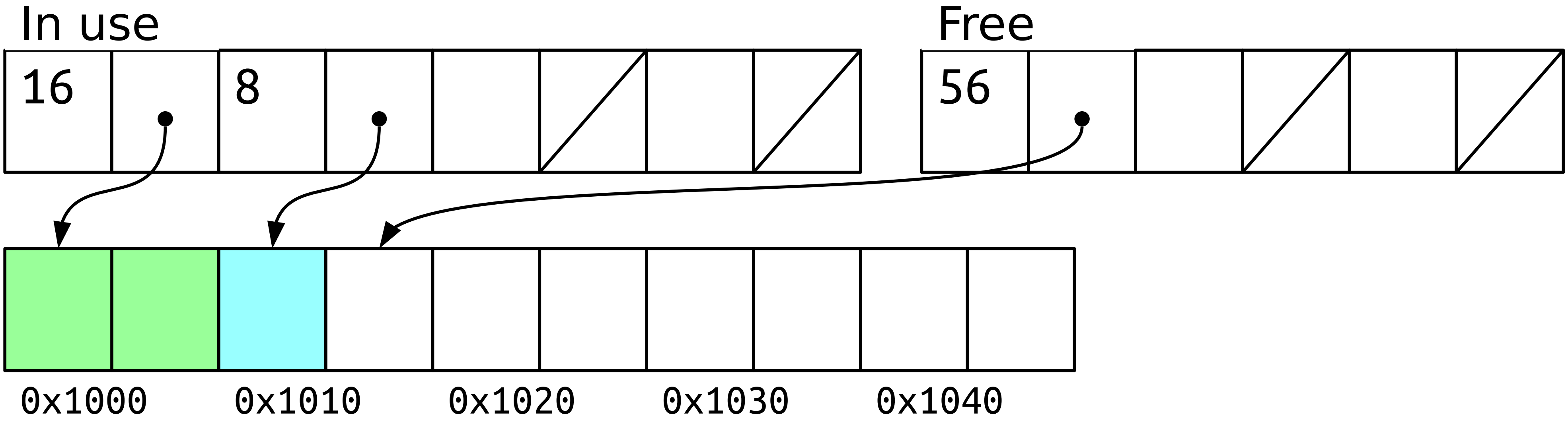
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(5);
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



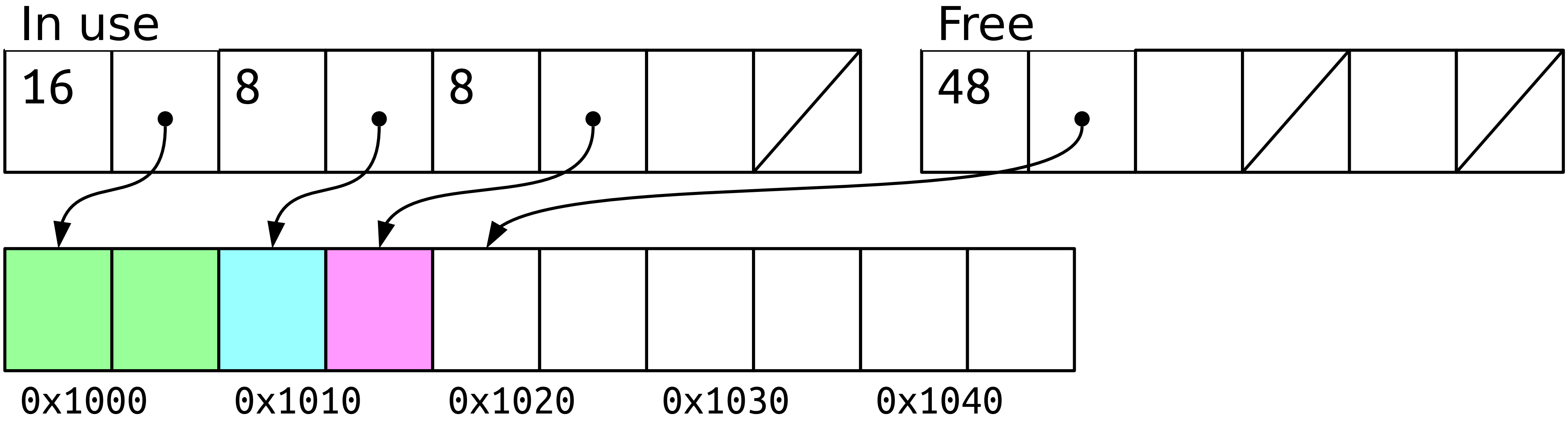
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1000
b = malloc(5);
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



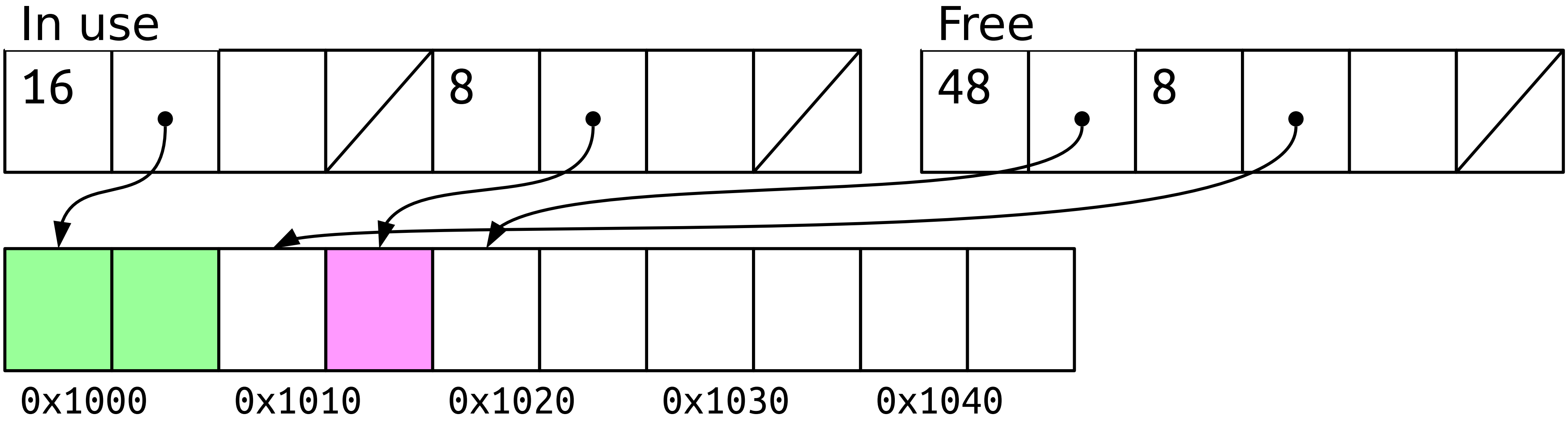
```
void *a, *b, *c, *d, *e;
a = malloc(16);       // 0x1000
b = malloc(5);        // 0x1010 (actually alloc 8)
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



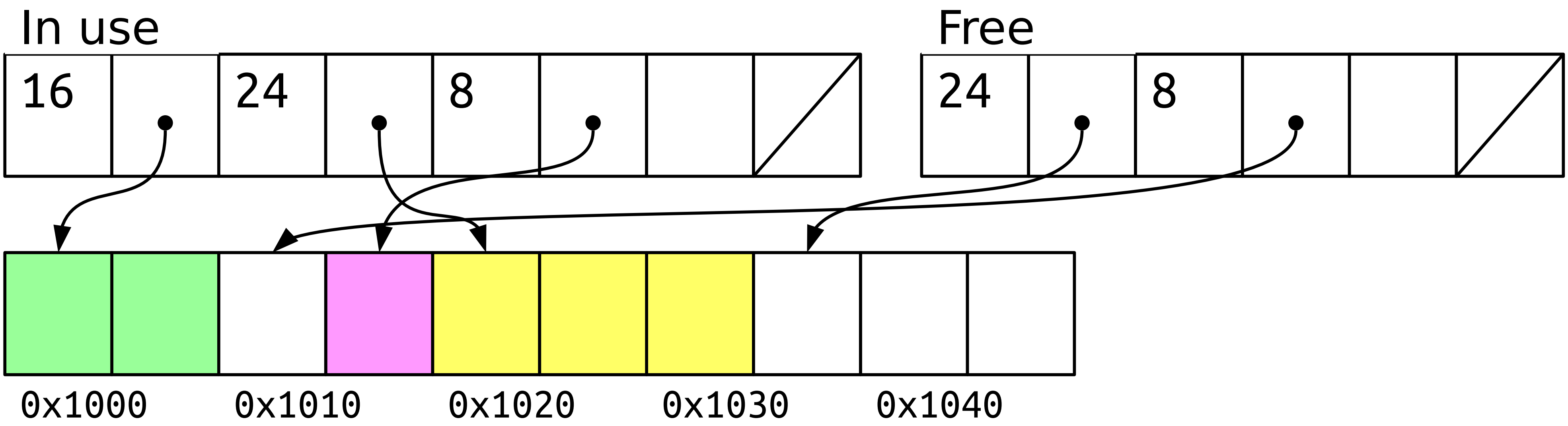
```
void *a, *b, *c, *d, *e;
a = malloc(16);       // 0x1000
b = malloc(5);        // 0x1010 (actually alloc 8)
c = malloc(8);        // 0x1018
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



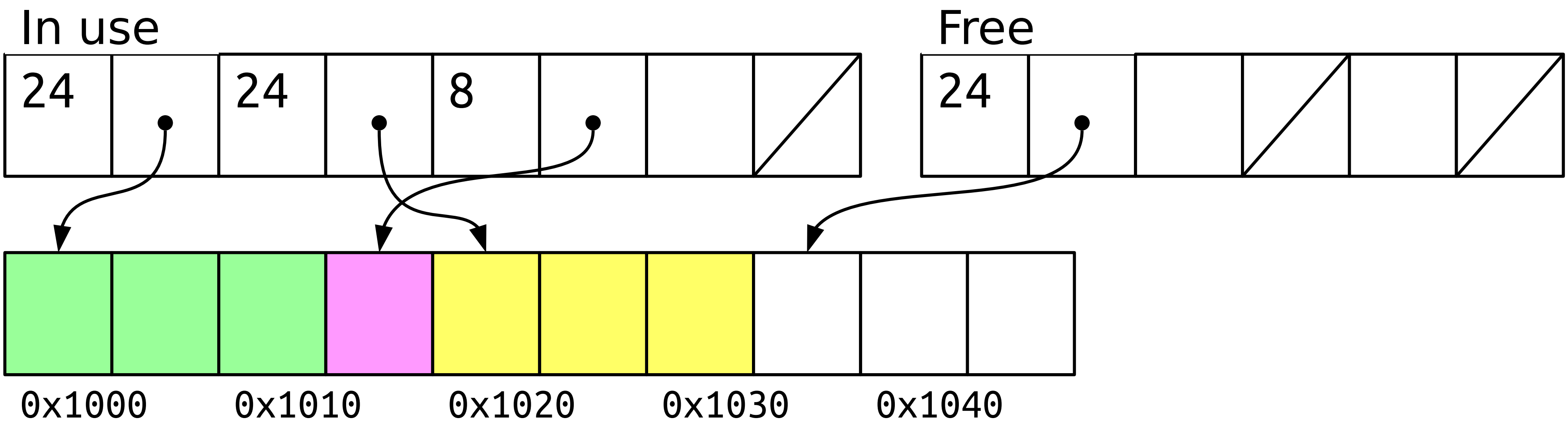
```
void *a, *b, *c, *d, *e;
a = malloc(16);       // 0x1000
b = malloc(5);        // 0x1010 (actually alloc 8)
c = malloc(8);        // 0x1018
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



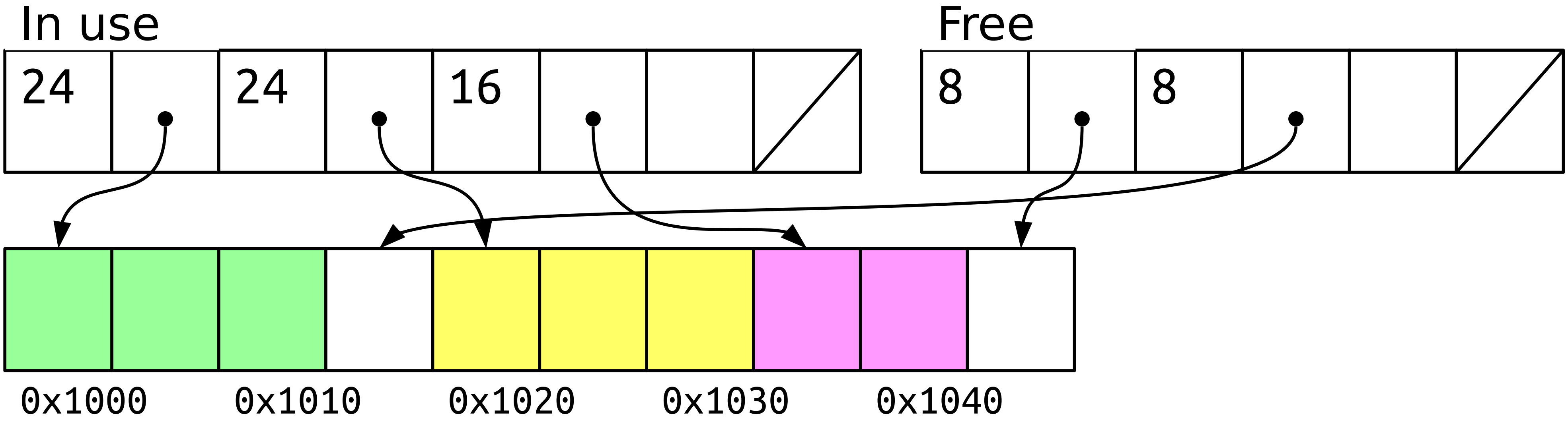
```
void *a, *b, *c, *d, *e;
a = malloc(16);     // 0x1000
b = malloc(5);      // 0x1010 (actually alloc 8)
c = malloc(8);      // 0x1018
free(b);
d = malloc(24);     // 0x1020
a = realloc(a, 24);
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



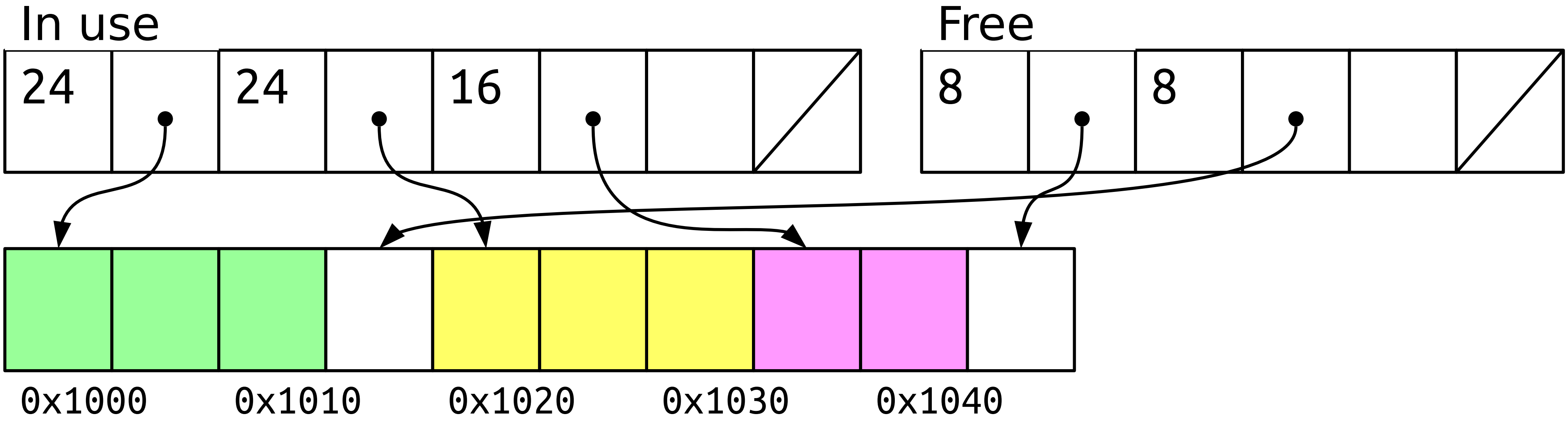
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1000
b = malloc(5);       // 0x1010 (actually alloc 8)
c = malloc(8);       // 0x1018
free(b);
d = malloc(24);      // 0x1020
a = realloc(a, 24); // 0x1000 (expand in-place)
c = realloc(c, 16);
e = malloc(16);
```

First Implementation



```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1000
b = malloc(5);       // 0x1010 (actually alloc 8)
c = malloc(8);       // 0x1018
free(b);
d = malloc(24);      // 0x1020
a = realloc(a, 24); // 0x1000 (expand in-place)
c = realloc(c, 16); // 0x1038 (moved)
e = malloc(16);
```

First Implementation



```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1000
b = malloc(5);       // 0x1010 (actually alloc 8)
c = malloc(8);       // 0x1018
free(b);
d = malloc(24);      // 0x1020
a = realloc(a, 24); // 0x1000 (expand in-place)
c = realloc(c, 16); // 0x1038 (moved)
e = malloc(16);      // NULL (no contiguous space)
```

Observations

OK to over-allocate a little

- Not good utilization of space, but still correct

Can't move blocks once allocated

- Would invalidate all pointers

Fragmentation can be a problem

- Internal: unused space inside a block
- External: free space scattered throughout heap

Limitation

- Have to keep two separate data structures
- In figure, only 4 in-use and 3 free "slots;" what if we run out?

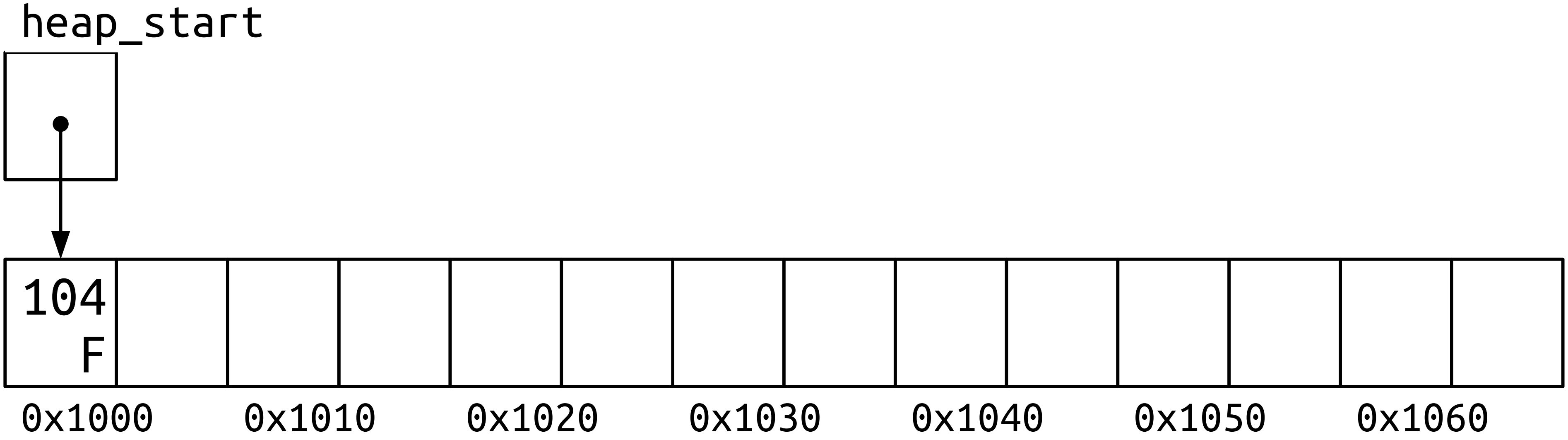
Idea: Pre-node header

- Include size and in-use/free status next to payload
- E.g. LSB=1 is free, 0 is in-use, other bits store size

Implicit Free List

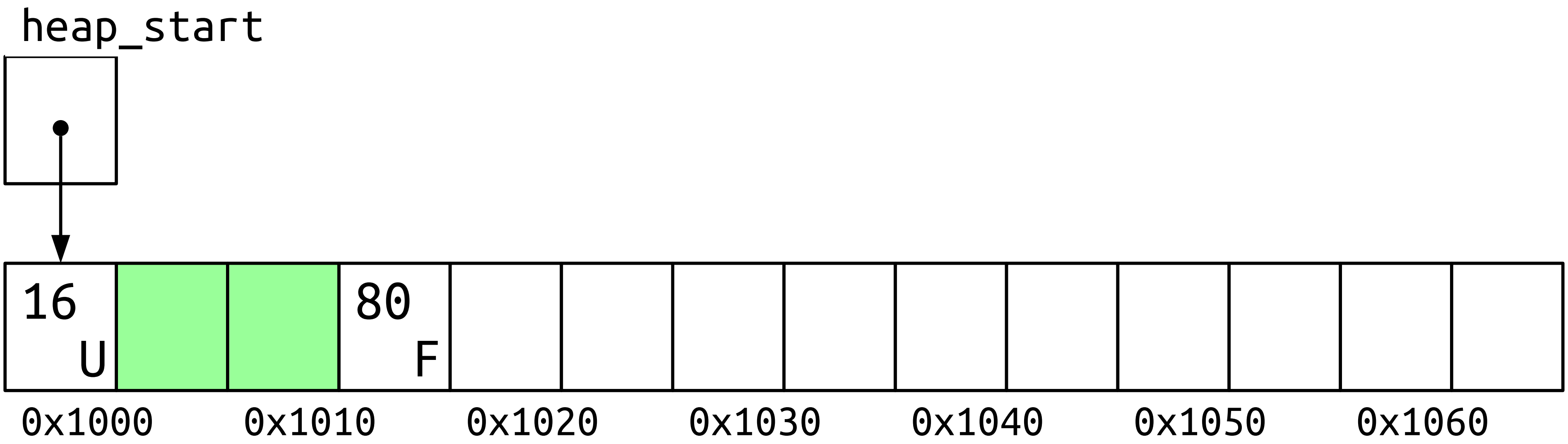
- To find a free block, traverse the heap, from one header to the next
- Check size, free status

Header and Implicit Free List



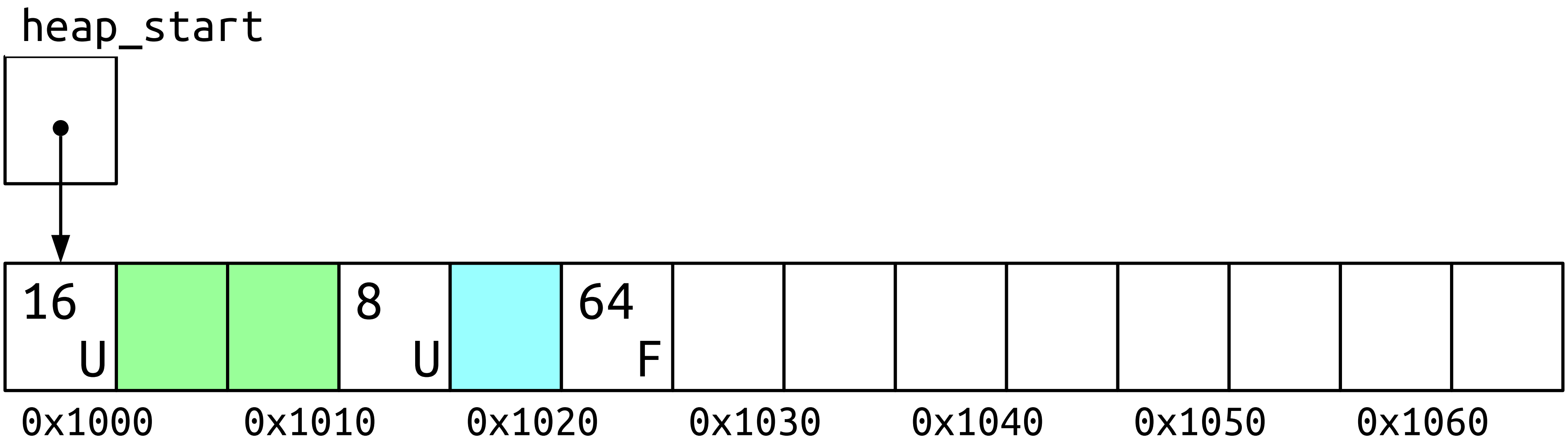
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(5);
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



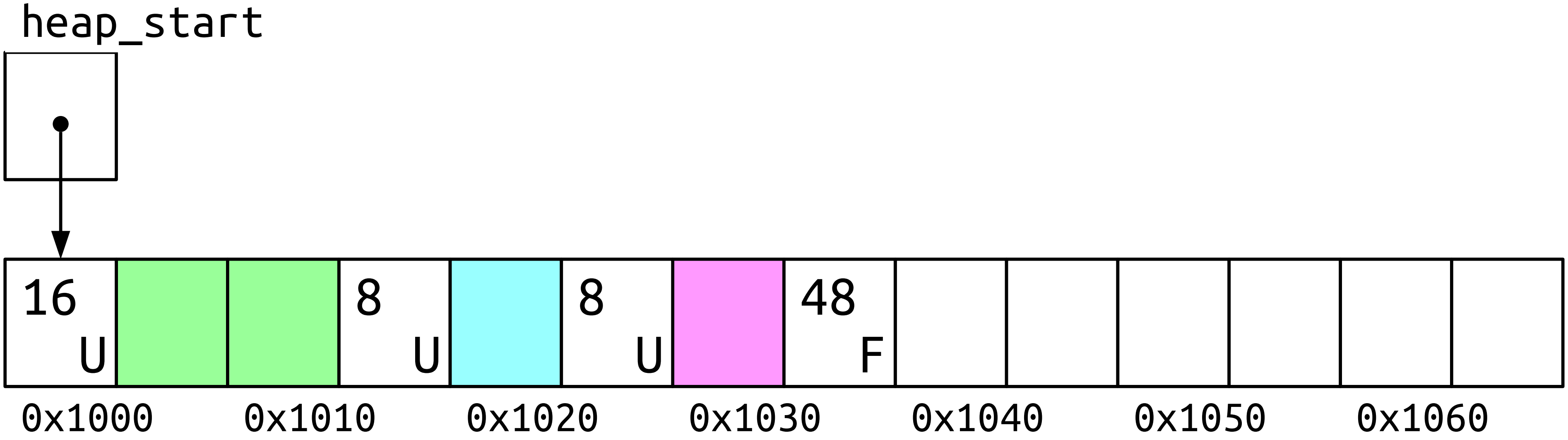
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



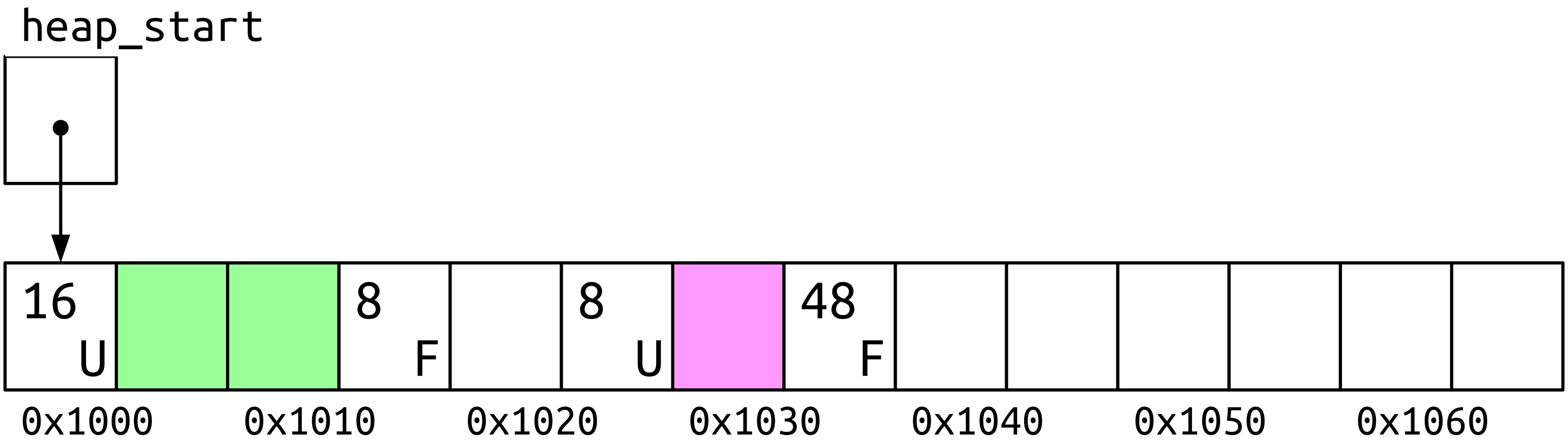
```
void *a, *b, *c, *d, *e;
a = malloc(16);       // 0x1008
b = malloc(5);        // 0x1020
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



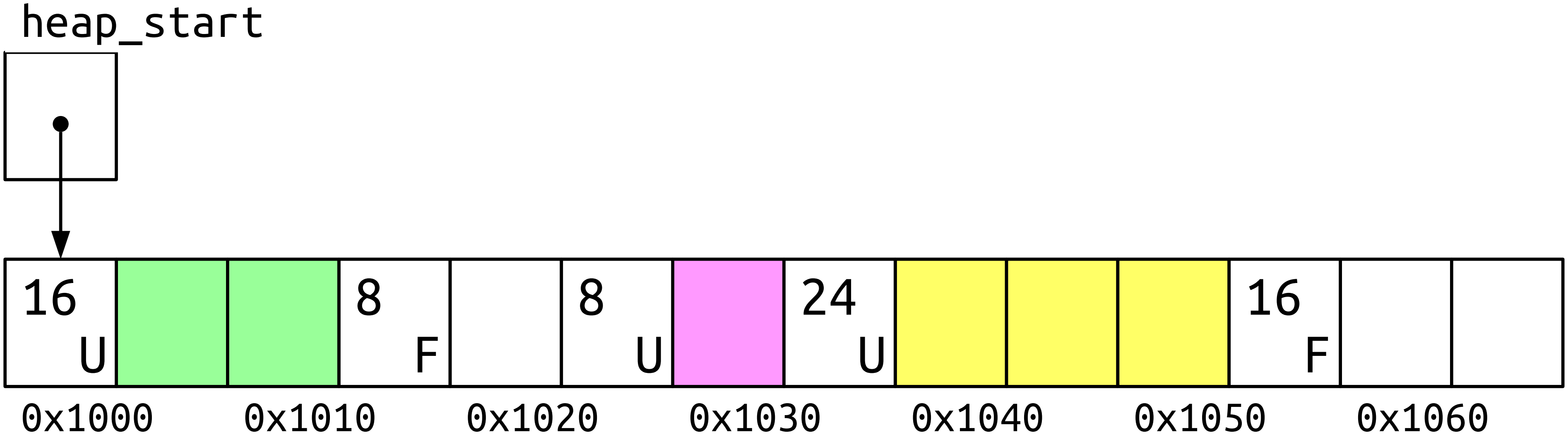
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



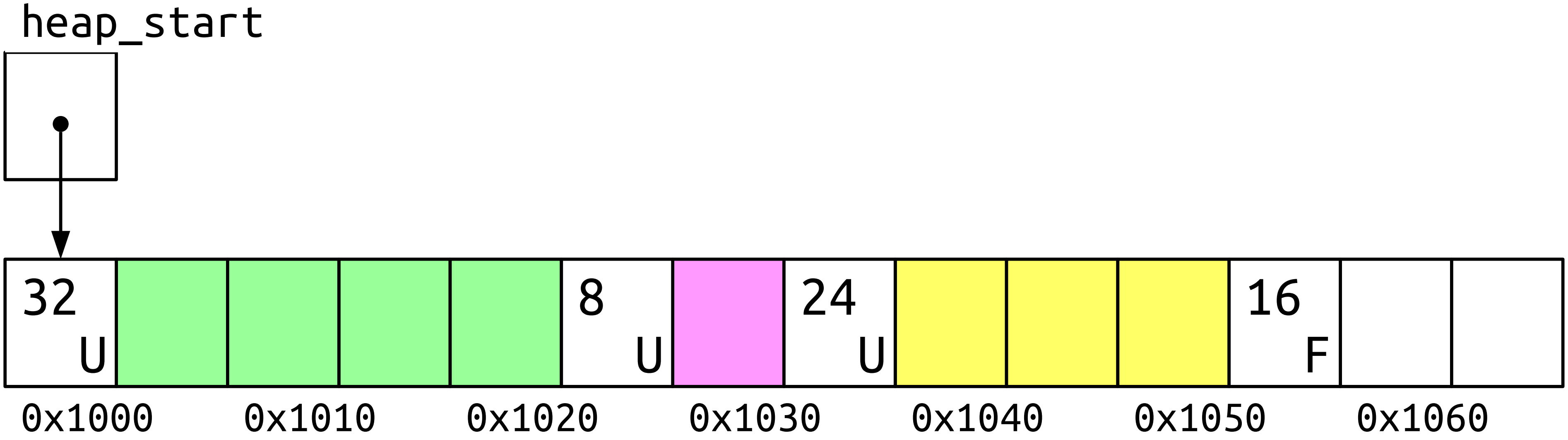
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



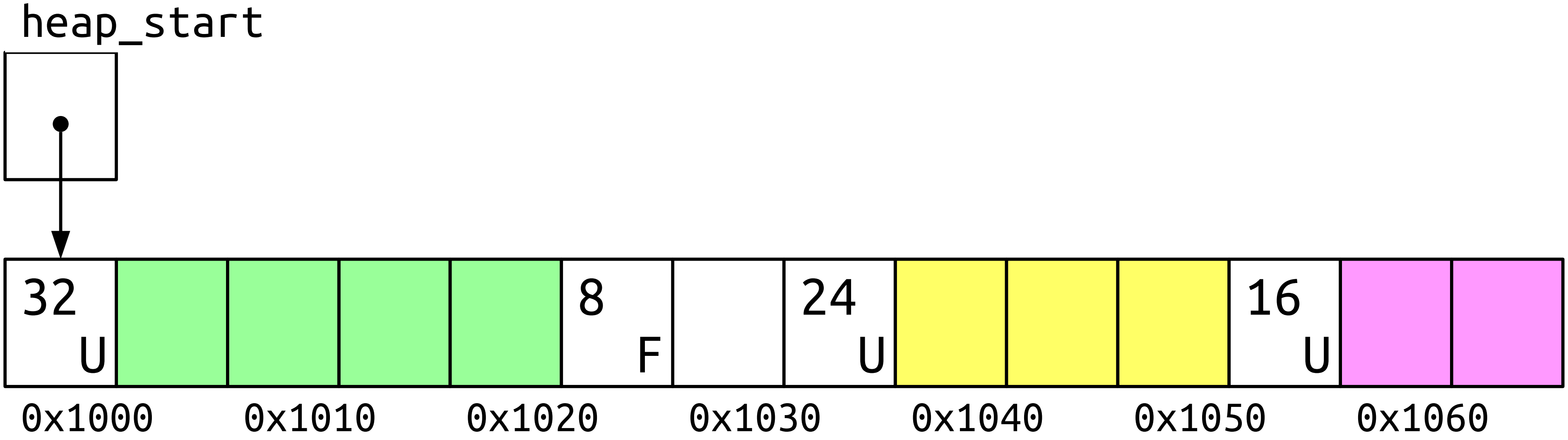
```
void *a, *b, *c, *d, *e;
a = malloc(16);        // 0x1008
b = malloc(5);         // 0x1020
c = malloc(8);         // 0x1030
free(b);
d = malloc(24);        // 0x1040
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



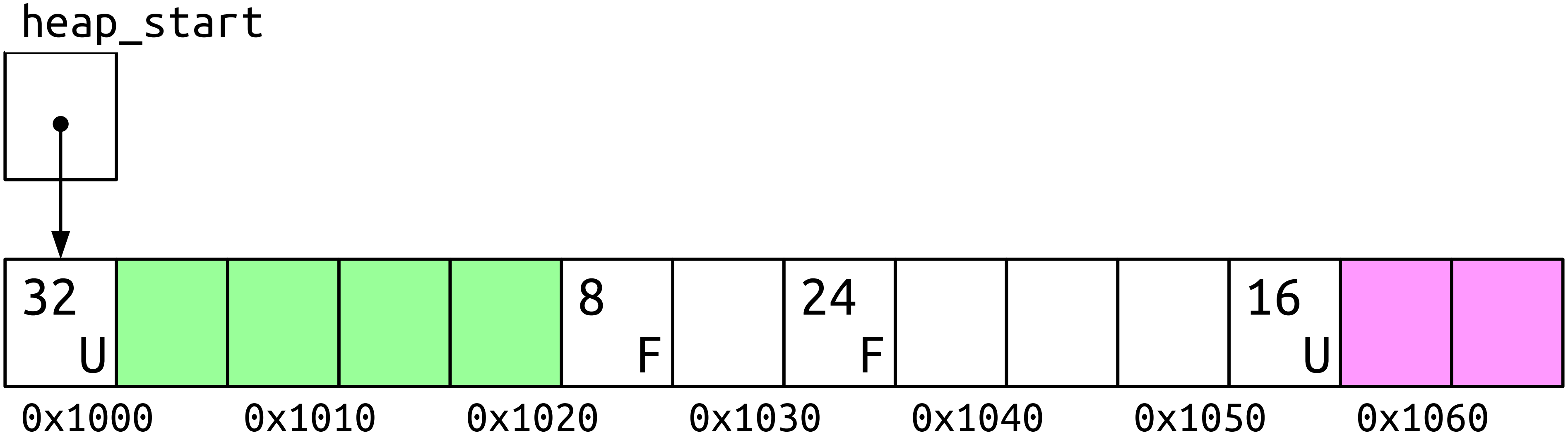
```
void *a, *b, *c, *d, *e;
a = malloc(16);       // 0x1008
b = malloc(5);        // 0x1020
c = malloc(8);        // 0x1030
free(b);
d = malloc(24);       // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16);
free(d);
```

Header and Implicit Free List



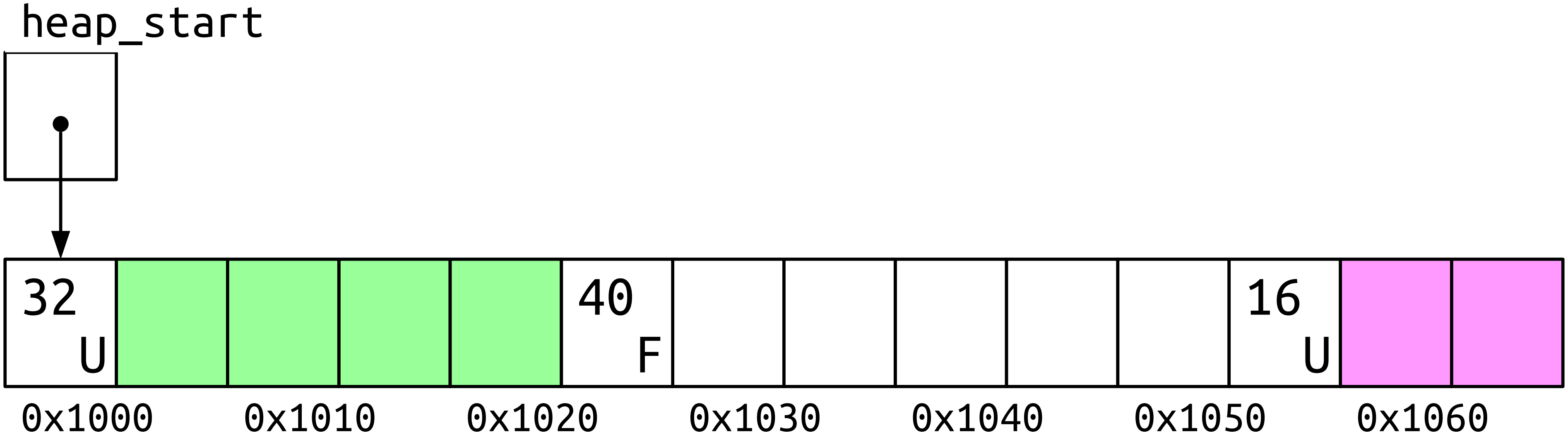
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);      // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16); // 0x1060
free(d);
```

Header and Implicit Free List



```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);      // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16); // 0x1060
free(d);
```

Coalescing Free Blocks



```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);      // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16); // 0x1060
free(d);
```

Observations

malloc/free have size information

- ...but not exposed to client (why?)

Writing past end of a block

- How much damage can we do?

Passing bad pointer to malloc/free

- E.g. pointer to middle of block

Design Decisions

Which free block to use?

- First fit: first block we find that works
- Best fit: waste as little space as possible

When to divide a block?

- Always: waste less space, more small blocks
- Rarely: fewer blocks to search through

When to coalesce?

- Immediately: potentially better utilization, slower
- Deferred: less splitting, maybe reuse small blocks

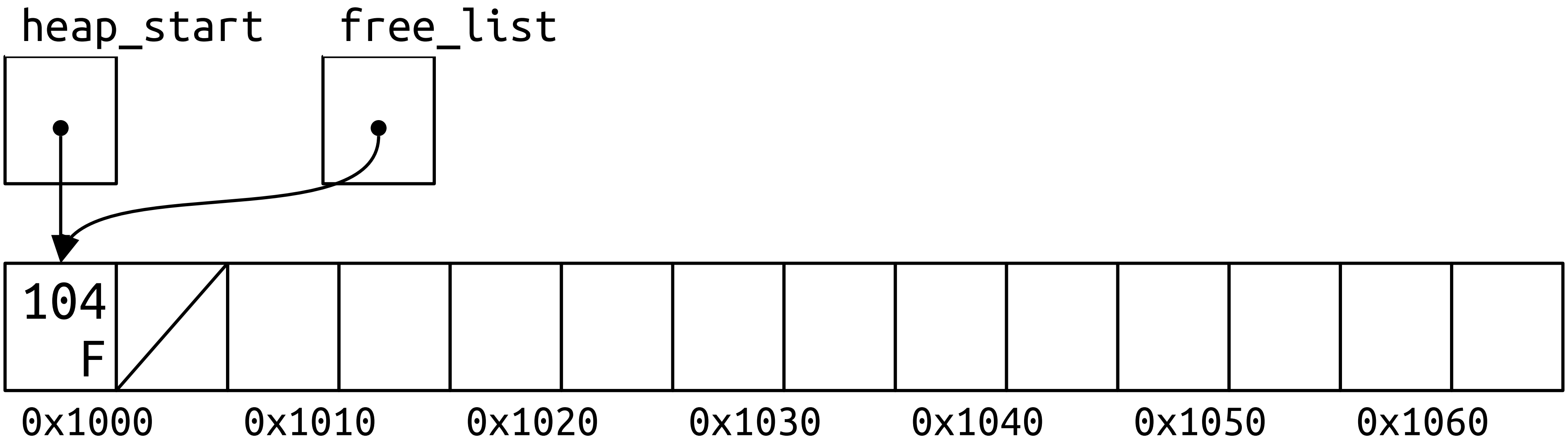
Explicit Free List

Problem: searching entire heap for free blocks is slow

Idea: explicit free list

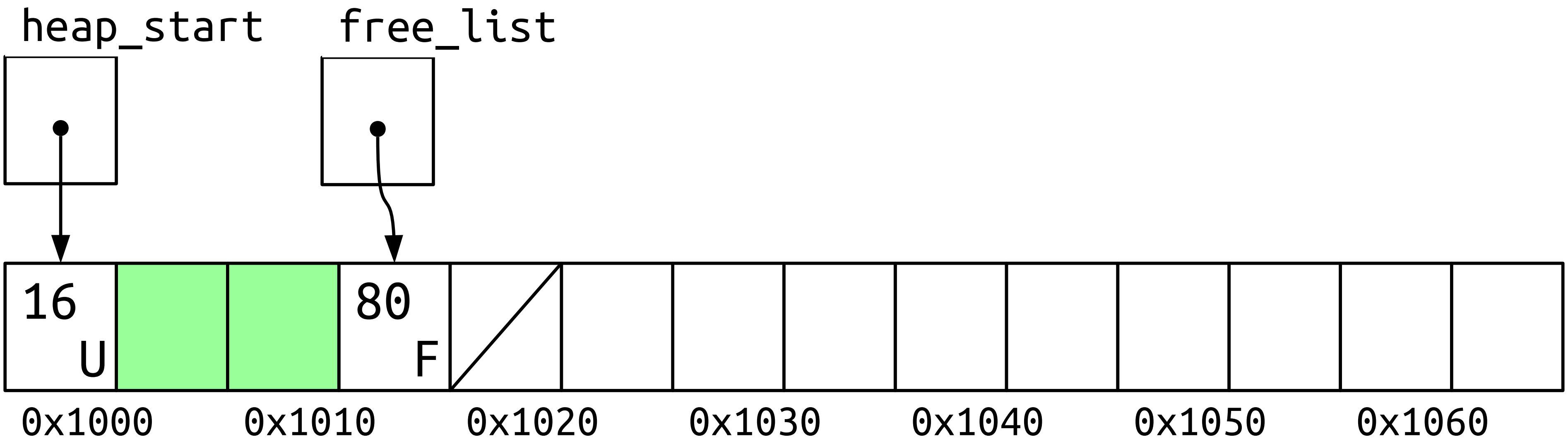
- If block is not allocated, we (the allocator) can use the payload
- Use payload to store pointer to next free block
- Creates a linked list of free blocks

Eplicit Free List



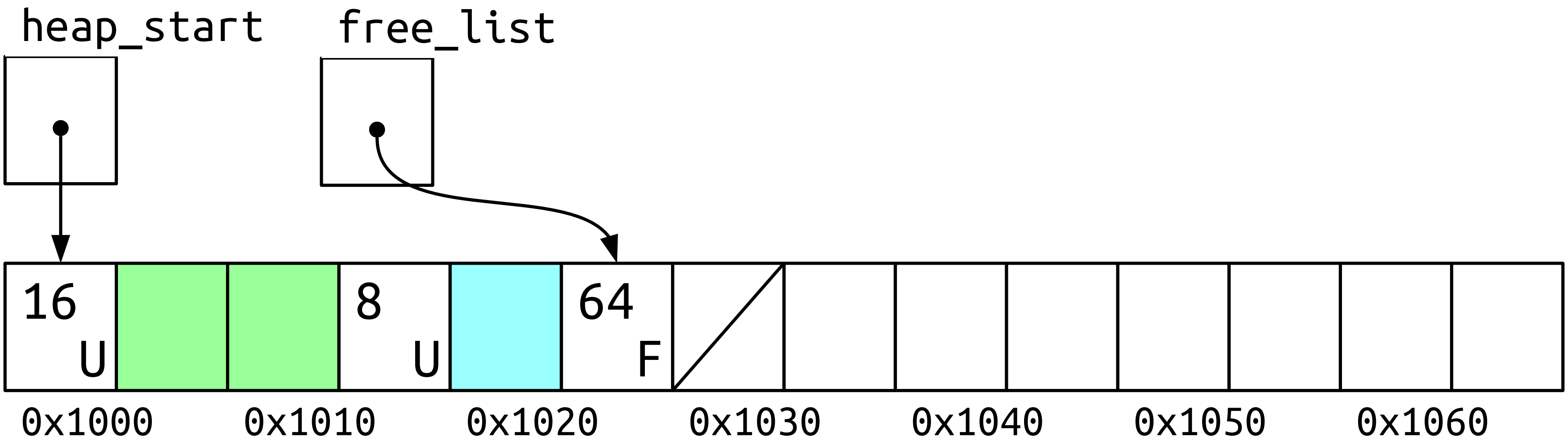
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(5);
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Explicit Free List



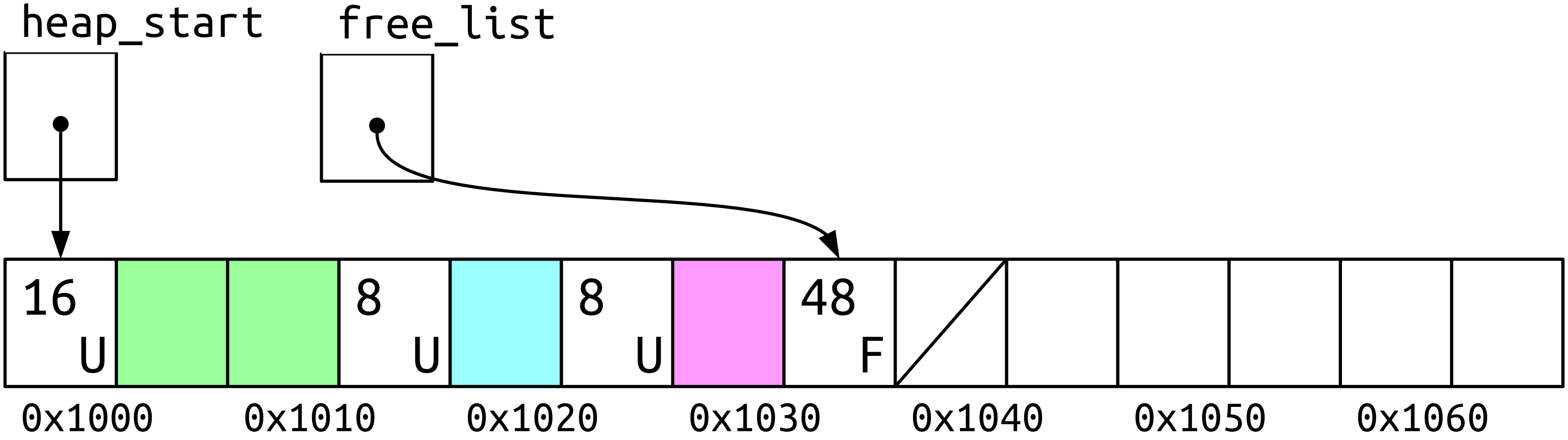
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Explicit Free List



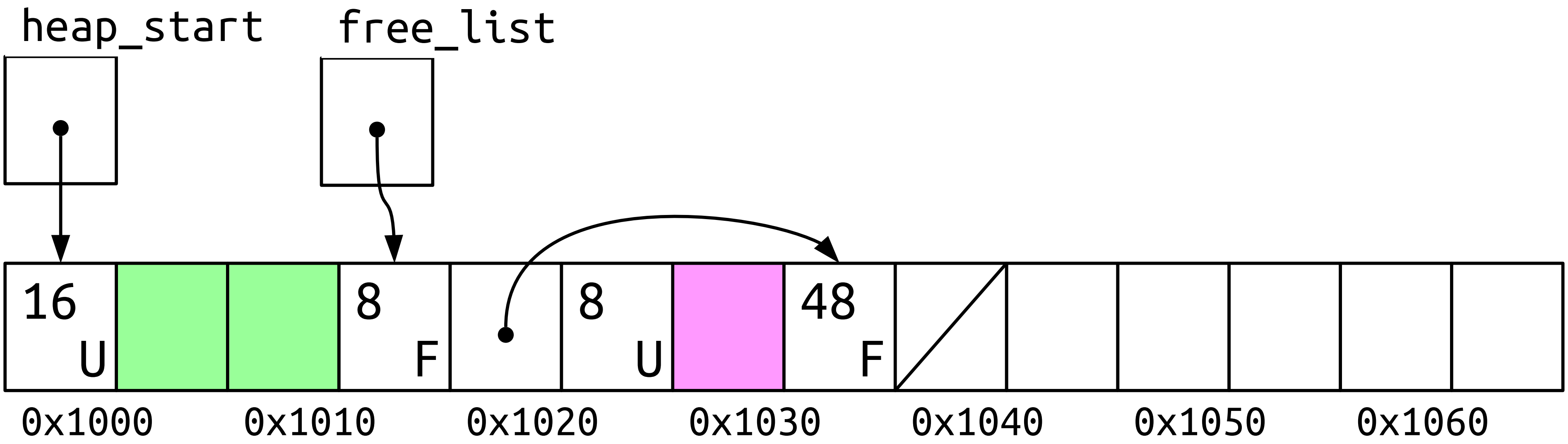
```
void *a, *b, *c, *d, *e;
a = malloc(16);        // 0x1008
b = malloc(5);         // 0x1020
c = malloc(8);
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Explicit Free List



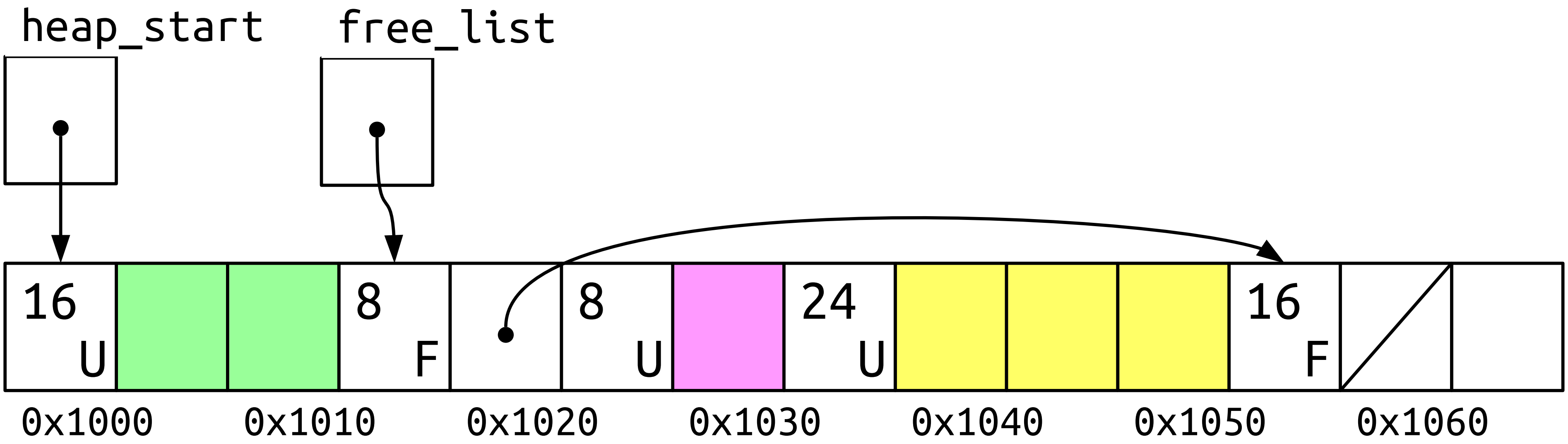
```
void *a, *b, *c, *d, *e;
a = malloc(16);         // 0x1008
b = malloc(5);          // 0x1020
c = malloc(8);          // 0x1030
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Explicit Free List



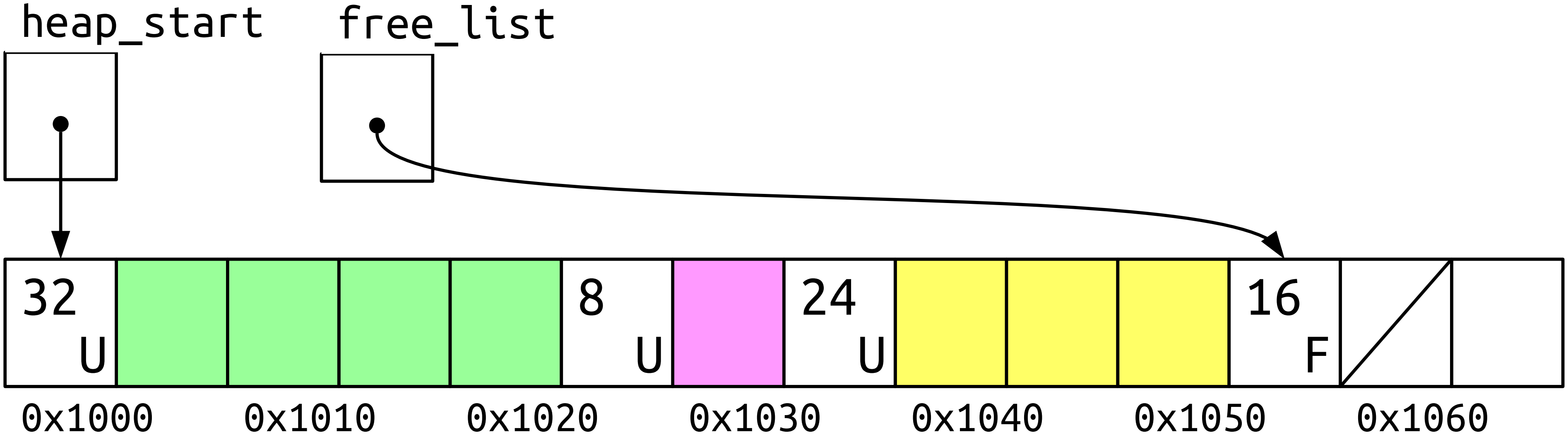
```
void *a, *b, *c, *d, *e;
a = malloc(16);         // 0x1008
b = malloc(5);          // 0x1020
c = malloc(8);          // 0x1030
free(b);
d = malloc(24);
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Explicit Free List



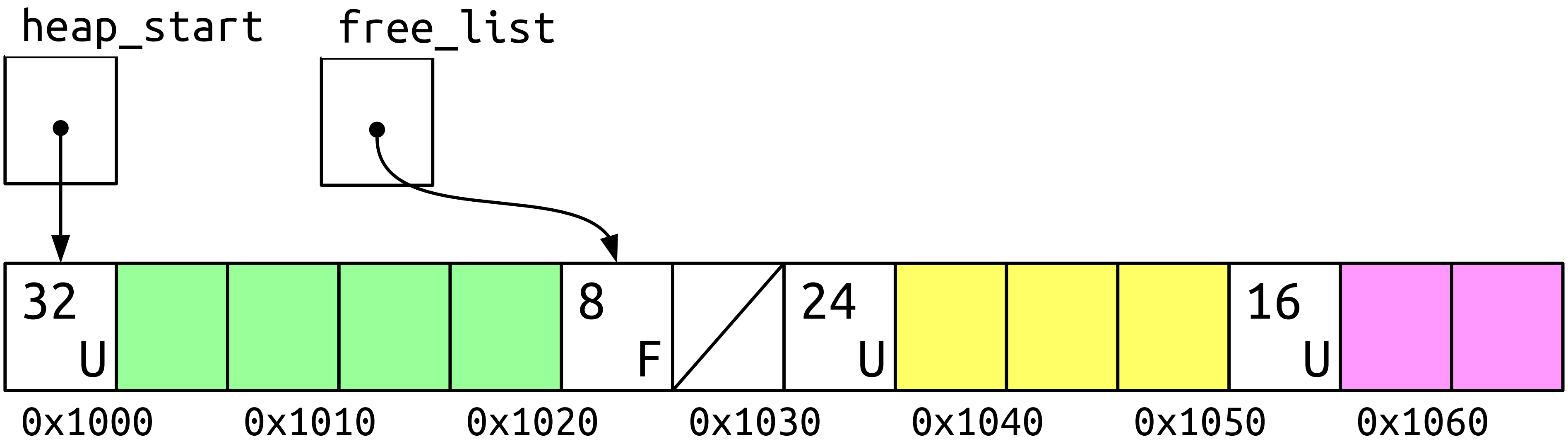
```
void *a, *b, *c, *d, *e;
a = malloc(16);        // 0x1008
b = malloc(5);         // 0x1020
c = malloc(8);         // 0x1030
free(b);
d = malloc(24);        // 0x1040
a = realloc(a, 24);
c = realloc(c, 16);
free(d);
```

Explicit Free List



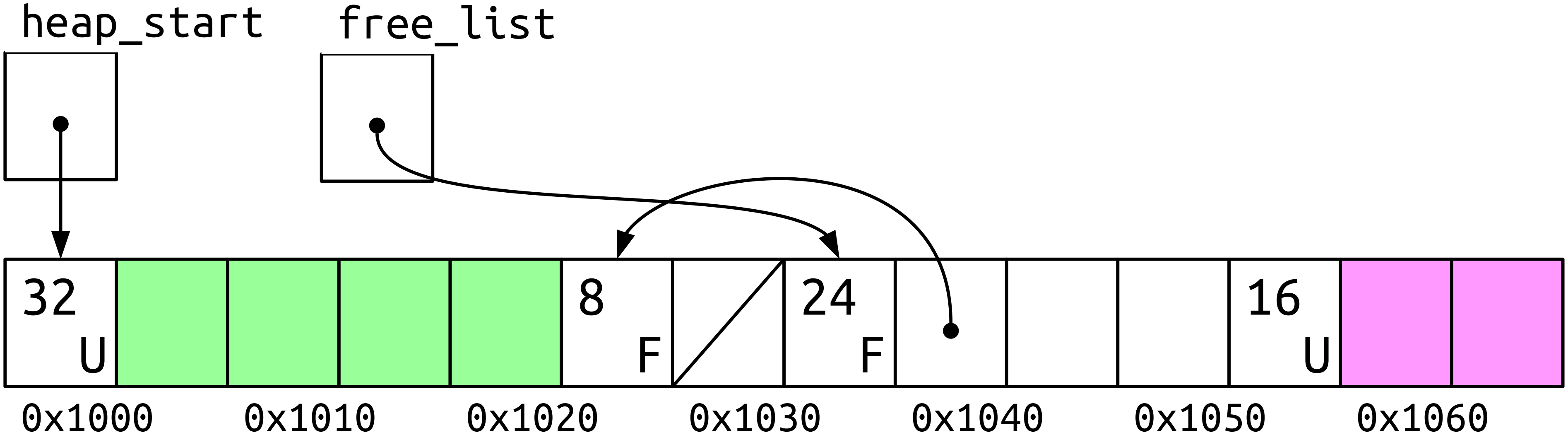
```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);      // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16);
free(d);
```

Explicit Free List



```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);      // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16); // 0x1060
free(d);
```

Explicit Free List



```
void *a, *b, *c, *d, *e;
a = malloc(16);      // 0x1008
b = malloc(5);       // 0x1020
c = malloc(8);       // 0x1030
free(b);
d = malloc(24);      // 0x1040
a = realloc(a, 24); // 0x1000
c = realloc(c, 16); // 0x1060
free(d);
```

Other Ideas

Multiple free lists

- "buckets" for different sizes

Segregated storage

- Put blocks of same size together in heap

Data structure changes

- Footers
- Doubly-linked free list

High-Level Design Issues

malloc

- Dominated by search for free block
- Tradeoff: use any block vs. use best

free

- Efficient find/update metadata
- Tradeoff: coalesce vs. put off work until later

realloc

- Most blocks don't expand
- Those that do will probably expand a lot
- Tradeoff: alloc a lot now vs. wait till client asks

Summary

Understand how the heap is managed

- Where the memory comes from
- How the heap allocator hands out memory

Explore design tradeoffs and optimizations of heap allocators

- What makes a good allocator?