CS 107
Lecture 13: Managing the Heap Part I

Monday, February 27, 2023

Computer Systems
Winter 2023
Stanford University
Computer Science Department

Reading: Course Reader: Managing the Heap Language, Textbook: Chapter 9.9

Lecturer: Chris Gregg

```
malloc()
calloc()
realloc()
free()
```

Today's Topics

- Reading: Chapter 9.9
- Programs from class: `/afs/ir/class/cs107/samples/lect13`
- Logistics
    - Bank vault — how is it going?
- Program Address Space
- What does it mean to allocate memory?
- The Heap, under the hood
    - Why do we have both stack and heap allocation?
    - Refresher on `malloc`, `free`, and `realloc`.
    - Allocator Requirements
    - Allocator Goals
    - Tracing the heap
    - How do we track heap allocations?
    - Placement: first-fit, next-fit, best-fit (throughput -vs- utilization)
    - Two different free lists: implicit and explicit
    - Splitting / Coalescing

Feedback

Thank you for the excellent feedback!

- Yes, office hours are busy, and I know that can be frustrating. Please use Ed for a bit less synchronous help.

  "CS106A has electronic exams and CS107 previously had them, this testing method seems like a regression."

- I agree. I have had electronic exams for many years, and like them. The problem is that too many students were cheating on electronic exams, so we went back to paper. If you want to help create an electronic exam that is impervious to cheating, please reach out to me!

Feedback

> "At a high level this class seems obsessed with having students experience unpleasant tasks that were common 20 years ago but have since become very uncommon as programmers collectively developed powerful tools to automate these tasks. Working in C, decoding assembly, using CLI tools that lack modern productivity enhancements, working with the x86 instruction sets, manual memory management. I hope the faculty is developing a version of this class which covers the same academic topics: bitwise manipulation, strings, pointers, heap vs stack, generics, floating point numbers, assembly, disassembly all in the context of the modern world of computing. This is the only course I've taken at Stanford where I've felt that a majority of the work was pointless outside of this particular classroom."

- These tasks have *not* become uncommon for systems-level work, believe it or not. I'm sorry you don't believe this, but understanding at this level and knowing how to work with these tools *is* important for operating systems developers, embedded code developers, and others. But, systems is not all of CS. We do want you to have experience at this level, and we are not planning on removing it from the curriculum.

What does it mean to allocate memory?

As we have discussed, your programs have two areas of main memory: the stack and the heap.

Your program has (by default) 8MB of stack space that it must manage based on the conventions we discussed when learning assembly code.

The heap, on the other hand, is ultimately controlled by the operating system, and a "heap allocator" (your final project!) maintains the heap as a collection of contiguous memory *blocks* that are either *free* or *allocated*.

An *allocated* block has been reserved for a particular application. When you call **`malloc()`**, you now have access to an allocated block, and only your program can modify or read the values in that block. Allocated blocks remain allocated for the rest of your program, or until you **`free()`** them. If your program ends, the heap allocator frees the block.

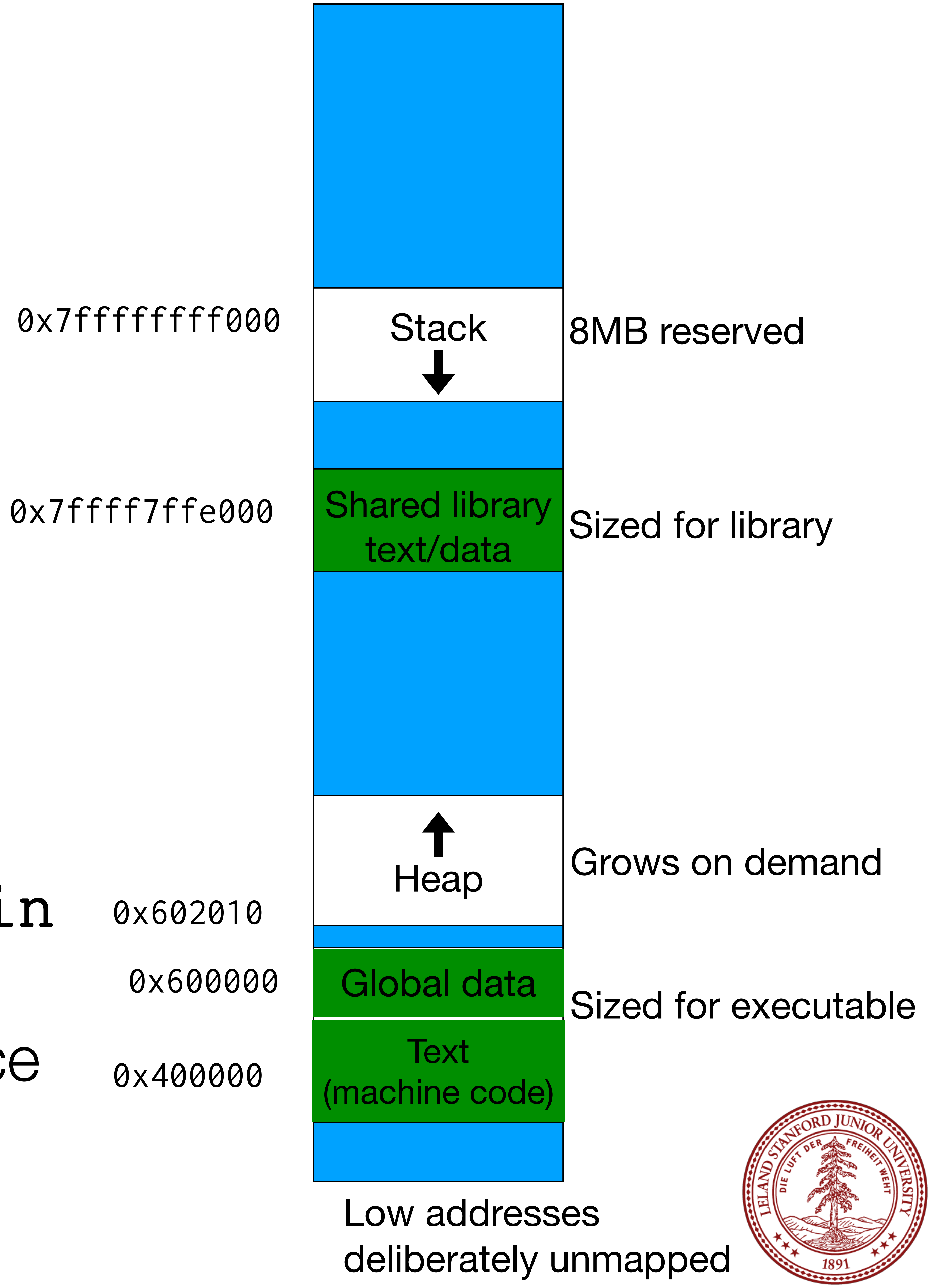
Program Address Space

Ever wonder what happens when you type the following?

```
./program_name
```

The ***OS loader*** handles this — it does the following:

1. Creates a new process
2. Sets up address space/segments
3. Reads executable file, loads instructions, global data
   Mapped from file into green segments
4. Libraries loaded on demand
5. Sets up and reserves the 8MB stack
   Reserves stack segment, initializes `%rsp`, calls `main`
6. malloc written in C, will init self on use
7. Asks OS for large memory region, parcels out to service requests



`0x7ffffffff000` Stack 8MB reserved

`0x7ffff7ffe000` Shared library text/data Sized for library

Heap Grows on demand

`0x602010`

`0x600000` Global data

`0x400000` Text (machine code)

Sized for executable

Low addresses deliberately unmapped

Why do we have both stack and heap allocation?

As we have discussed before, stack memory is limited and serves as a scratch-pad for functions, and it is continually being re-used by your functions. Stack memory isn't persistent, but because it is already allocated to your program, it is fast.

Heap memory takes more time to set up (you have to go through the heap allocator), but it is unlimited (for all intents and purposes), and persistent for the rest of your program.

`malloc`, `free`, and `realloc` refresher

This is what your heap allocator is going to do!

`void *malloc(size_t size)`

Return pointer to memory block >= requested size

(failure returns `NULL` and sets errno)

`void free(void *p)`

Recycle memory block

`p` must be from previous malloc/realloc call

`void *realloc(void *p, size_t size)`

Changes size of block `p`, returns pointer to block (possibly same)

Contents of new block unchanged up to min of old and new size

If the new pointer isn't the same as the old pointer, the old block will have been free'd

Allocator Requirements

The heap allocator must be able to service arbitrary sequence of malloc() and free() requests

`malloc` must return a pointer to contiguous memory that is equal to or greater than the requested size, or `NULL` if it can't satisfy the request.

The *payload* contents (this is the area that the pointer points to) are unspecified — they can be 0s or garbage.

If the client introduces an error, then the behavior is undefined

- If the client tries to free non-allocated memory, or tries to use free'd memory.

The heap allocator has some constraints:

It can't control the number, size, or lifetime of the allocated blocks.

It must respond immediately to each malloc request

I.e., it can't reorder or buffer `malloc` requests — the first request must be handled first.

It *can* defer, ignore, or reorder requests to `free`

Allocator Requirements (continued)

Other heap allocator constraints:

The allocator must align blocks so they satisfy all alignment requirements

i.e., 16 byte alignment for GNU `malloc` (libc `malloc`) on 64-bit Linux (for your assignment, we only ask that you align on an 8-byte boundary).

The allocated payload must be maintained *as-is*

The allocator *cannot* move allocated blocks, such as to compact/coalesce free.

- Why not?

All of the programs with allocated memory would have corrupted pointers!

- The allocator *can* manipulate and modify free memory

Allocator Goals

The allocator should first and foremost attempt to service **`malloc`** and **`free`** requests *quickly.*

Ideally, the requests should be handled in *constant time* and should not degrade to linear behavior (we will see that some implementations can do this, some cannot)

The allocator must try for a *tight space utilization*.

Remember, the allocator has a fixed block of memory to dole out smaller parts — it must try to allocate efficiently

The allocator should try to minimize *fragmentation*.

It should try to group allocated blocks together.

There should be a small overhead relative to the payload (we will see what this mean soon!)

Allocator Goals (continued)

It is desirable for a heap allocator to have the following properties:

Good locality

- Blocks are allocated close in time are located close in space
- "Similar" blocks are allocated close in space

Robust

- Client errors should be recognized
  - What is required to detect and report them?

Ease of implementation and maintenance

- Having `*(void **)` all over the place makes for hard-to-maintain code. Instead, use structs, and typedef when appropriate.
- The code is necessarily complex, but the more efforts you put into writing clean code, the more you will be rewarded by easier-to-maintain code.

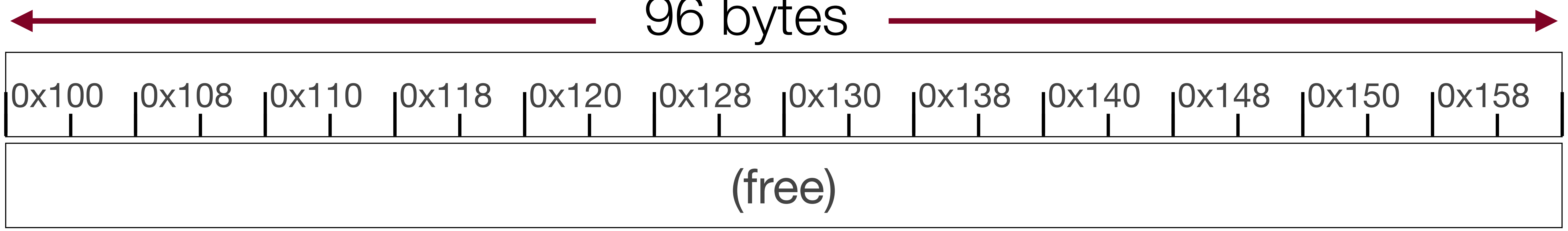
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x0**
a	0xffffe800	**0xbeef**

heap



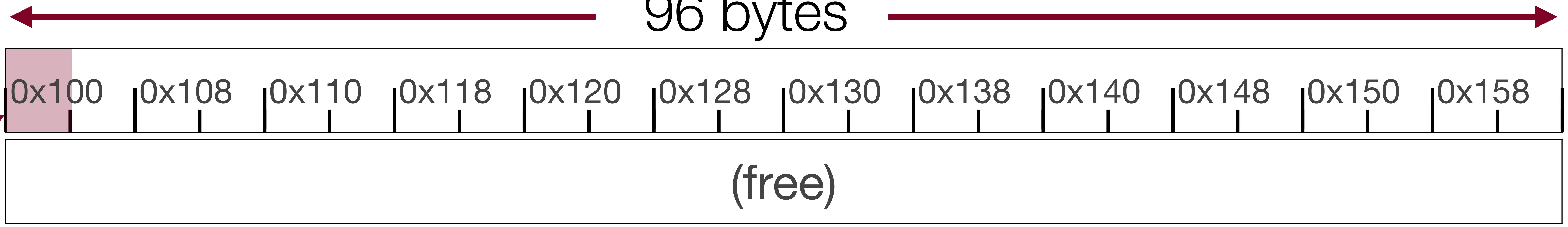
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0xabcde`**
c	`0xffffe810`	**`0xf0123`**
b	`0xffffe808`	**`0x0`**
a	`0xffffe800`	**`0xbeef`**

heap



Each section represents 4 bytes

Tracing the Heap (possible implementation)

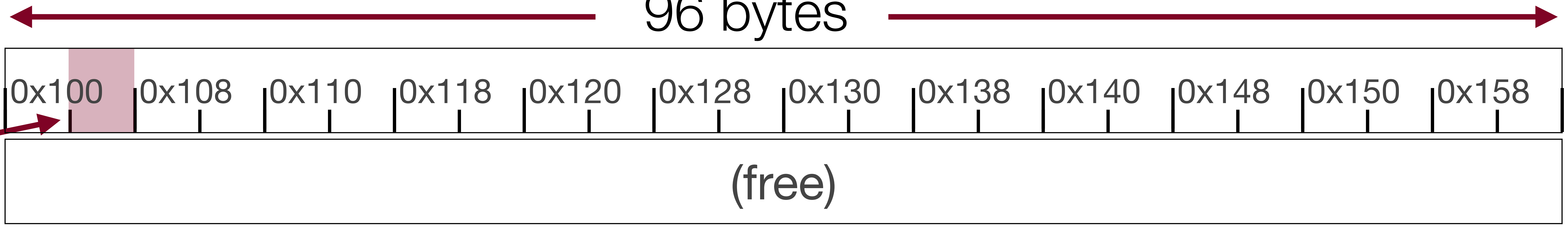
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0xabcde`**
c	`0xffffe810`	**`0xf0123`**
b	`0xffffe808`	**`0x0`**
a	`0xffffe800`	**`0xbeef`**

heap

96 bytes



Each section represents 4 bytes

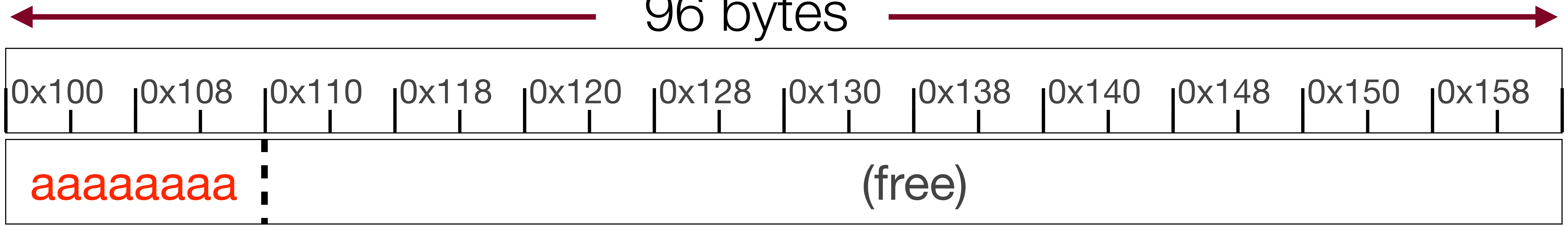
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0xabcde`**
c	`0xffffe810`	**`0xf0123`**
b	`0xffffe808`	**`0x0`**
a	`0xffffe800`	**`0x100`**

heap



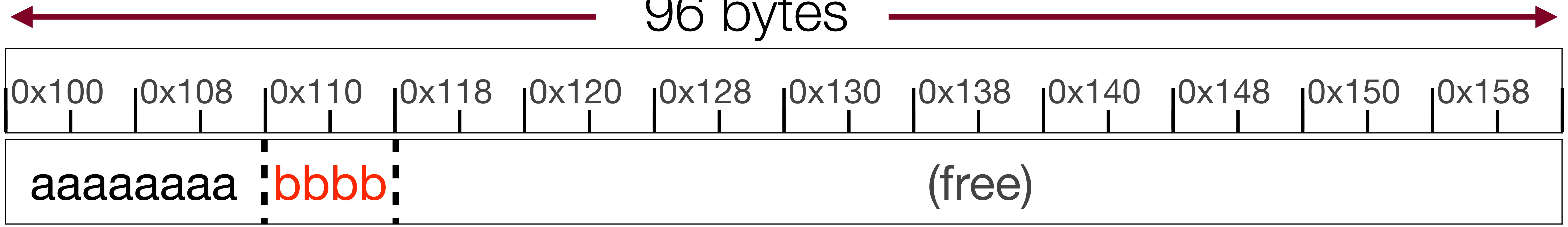
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0xabcde`**
c	`0xffffe810`	**`0xf0123`**
b	`0xffffe808`	**`0x110`**
a	`0xffffe800`	**`0x100`**

heap



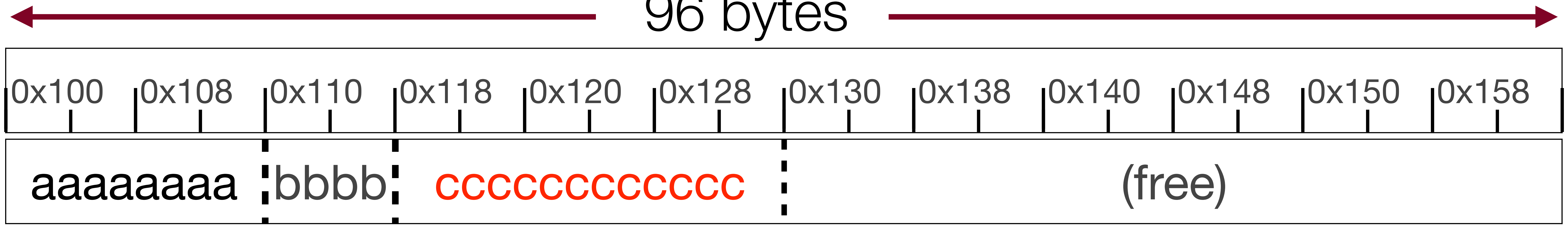
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0xabcde`**
c	`0xffffe810`	**`0x118`**
b	`0xffffe808`	**`0x110`**
a	`0xffffe800`	**`0x100`**

heap



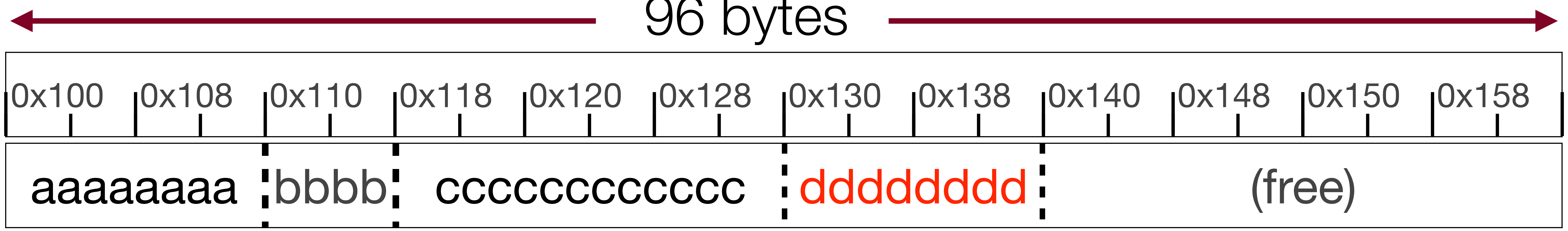
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**

heap



96 bytes

0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x138	0x140	0x148	0x150	0x158

aaaaaaaa | bbbb | cccccccccccc | dddddddd | (free)

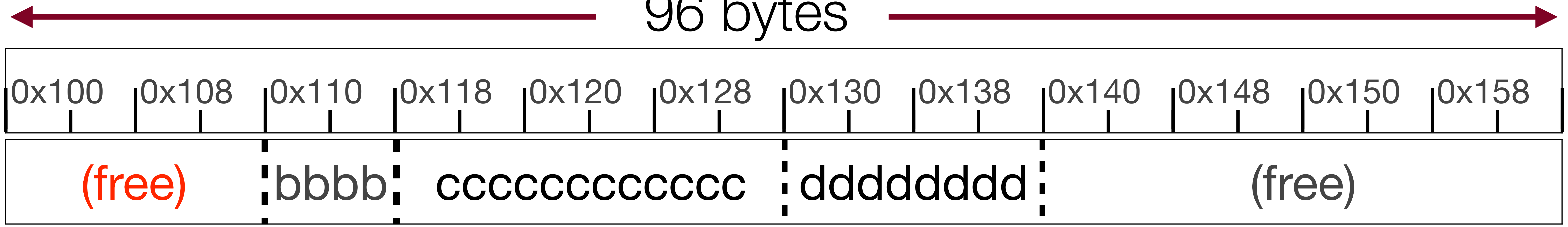
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0x130`**
c	`0xffffe810`	**`0x118`**
b	`0xffffe808`	**`0x110`**
a	`0xffffe800`	**`0x100`**

heap



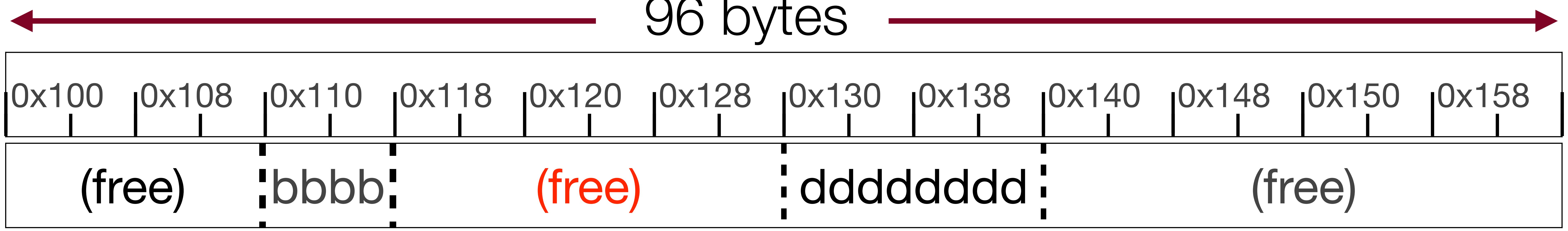
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	**`0x0`**
d	`0xffffe818`	**`0x130`**
c	`0xffffe810`	**`0x118`**
b	`0xffffe808`	**`0x110`**
a	`0xffffe800`	**`0x100`**

heap



96 bytes

0x100 0x108 0x110 0x118 0x120 0x128 0x130 0x138 0x140 0x148 0x150 0x158

(free) bbbb (free) dddddddd (free)

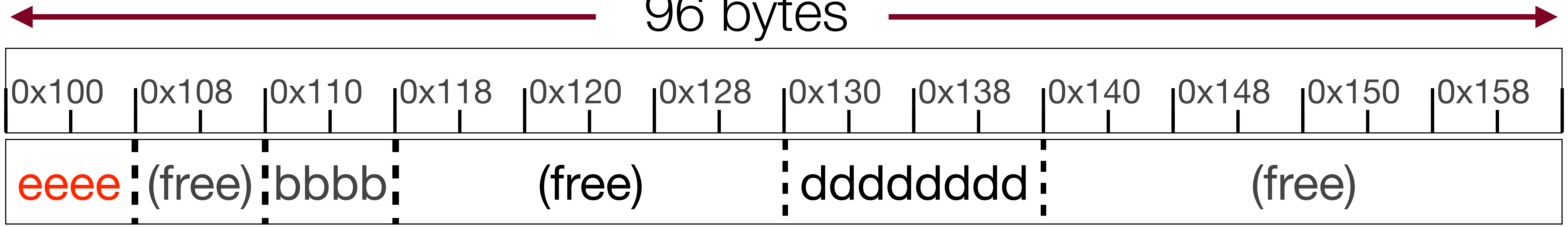
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	0xffffe820	0x100
d	0xffffe818	0x130
c	0xffffe810	0x118
b	0xffffe808	0x110
a	0xffffe800	0x100

heap



96 bytes

0x100 0x108 0x110 0x118 0x120 0x128 0x130 0x138 0x140 0x148 0x150 0x158

eeee (free) bbbb (free) dddddddd (free)

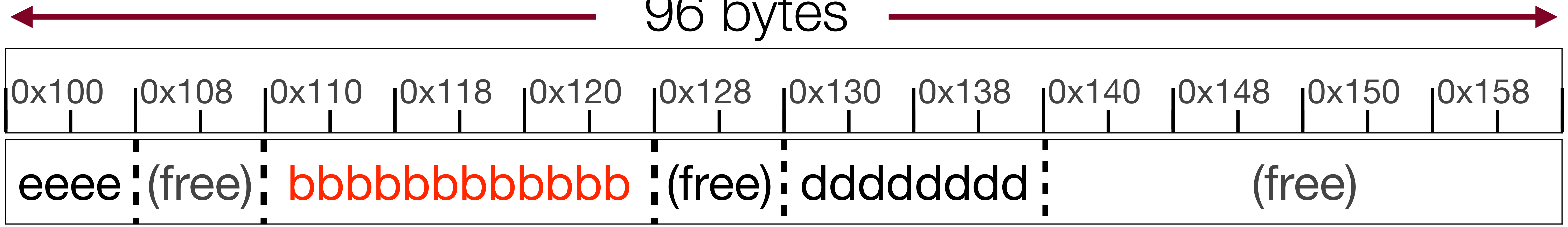
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	`0xffffe820`	`0x100`
d	`0xffffe818`	`0x130`
c	`0xffffe810`	`0x118`
b	`0xffffe808`	`0x110`
a	`0xffffe800`	`0x100`

heap


96 bytes

0x100 | 0x108 | 0x110 | 0x118 | 0x120 | 0x128 | 0x130 | 0x138 | 0x140 | 0x148 | 0x150 | 0x158

eeee | (free) | bbbbbbbbbbbb | (free) | dddddddd | (free)

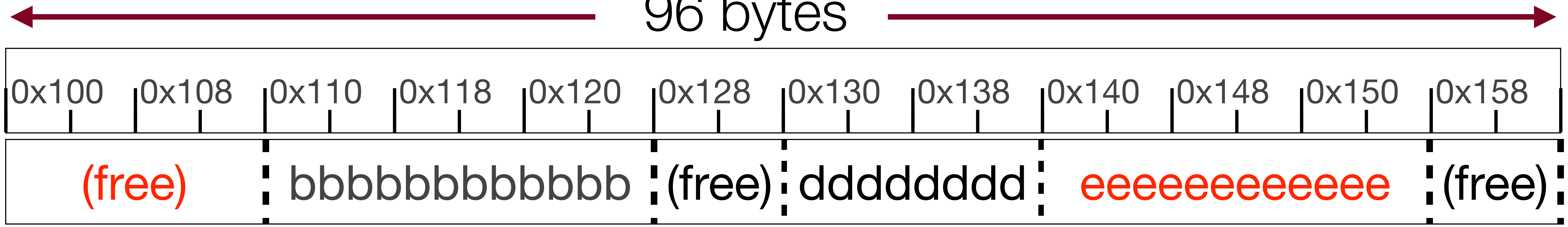
Tracing the Heap (possible implementation)

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

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x140**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**

heap


96 bytes

0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x138	0x140	0x148	0x150	0x158
(free)		bbbbbbbbbbbb			(free)	dddddddd		eeeeeeeeeeee			(free)

Tracing the Heap (possible implementation)

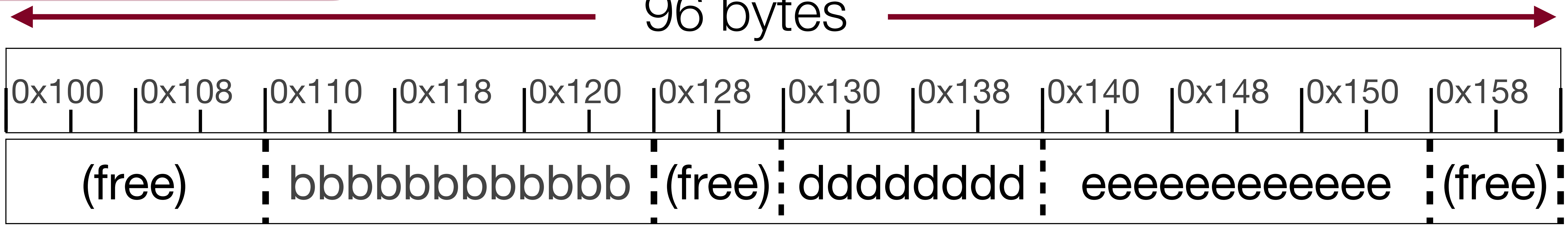
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

Returns `NULL`

	Address	Value
e	0xffffe820	**0x140**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**
f	0xffffe7f0	**0x0**

heap



Take a Break!

Three Minute Break

Heap Allocator Implementation Issues

- How do we track the information in a block?
    - Remember, `free()` is only given a pointer, not a size

- How do we organize/find free blocks?

- How do we pick which free block from available options?

- What do we do with excess space when allocating a block?

- How do we recycle a freed block?

One possibility: Separate list / table

- We could have a separate list or table that holds the free and in-use information.
  - Given an address, how do we look up the information?
  - How do we update the list or table to service `malloc`s and `free`s?
  - How much overhead is there per block?

- The separate list approach could be a reasonable approach (we would have to answer all of the above questions…), but it is not often used in practice, although there are some exceptions:
  - There are some special-case allocators that use this
  - Valgrind uses this, because it needs to keep track of lots more information than just the used / free blocks.

Another Possibility

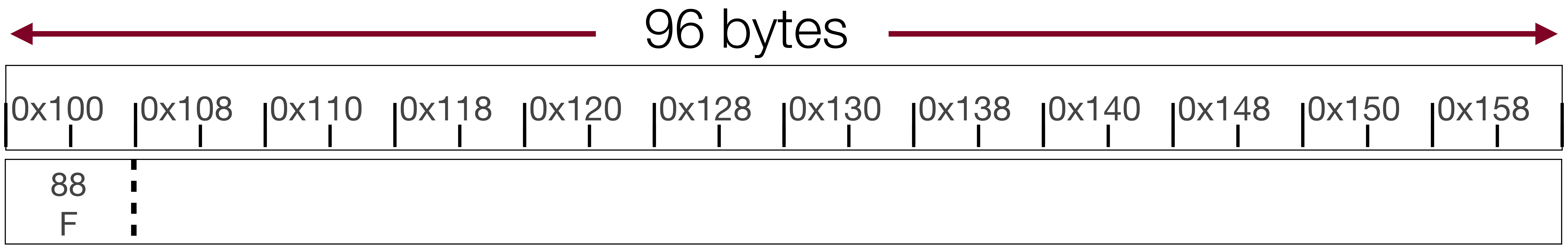
- A second possibility, and the one that is actually common and used in practice, uses what is called a **block header** to hold the information.

- The block header is actually *stored in the same memory area as the payload, and it generally precedes the payload.*

Another Possibility

- A second possibility, and the one that is actually common and used in practice, uses what is called a **block header** to hold the information.

- The block header is actually *stored in the same memory area as the payload, and it generally precedes the payload.*



96 bytes

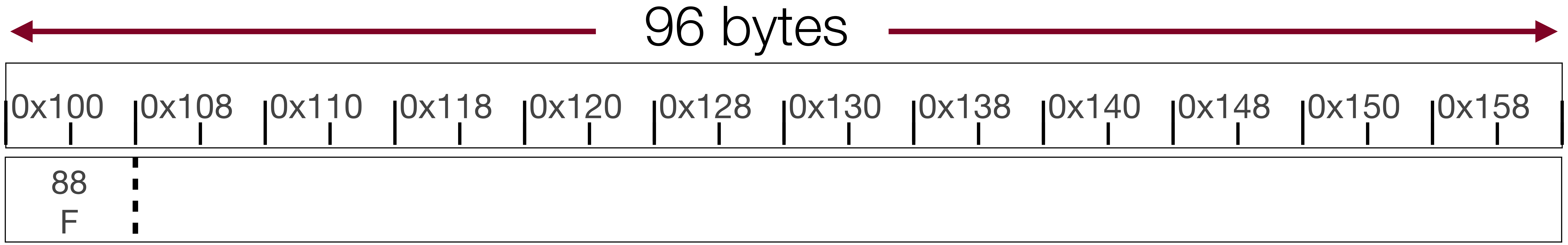
0x100 0x108 0x110 0x118 0x120 0x128 0x130 0x138 0x140 0x148 0x150 0x158

88
F

Another Possibility

- A second possibility, and the one that is actually common and used in practice, uses what is called a **block header** to hold the information.

- The block header is actually *stored in the same memory area as the payload, and it generally precedes the payload.*

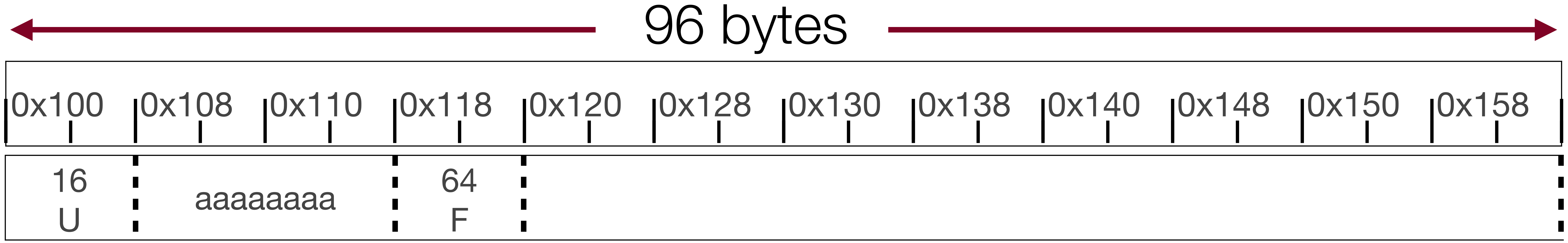


  - This is where things start to get a bit tricky. The heap allocator has 96 bytes, and it needs to keep the free block information *in those 96 bytes* (I N C E P T I O N)
  - In other words, the heap allocator is using part of the 96 bytes as housekeeping.
  - In this case, 8 bytes are taken up with the information that there are 88 Free (F) bytes ahead in the block.

Another Possibility

```
a = malloc(16);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	
b	`0xffffe808`	
a	`0xffffe800`	**0x108**



96 bytes

0x100 | 0x108 | 0x110 | 0x118 | 0x120 | 0x128 | 0x130 | 0x138 | 0x140 | 0x148 | 0x150 | 0x158

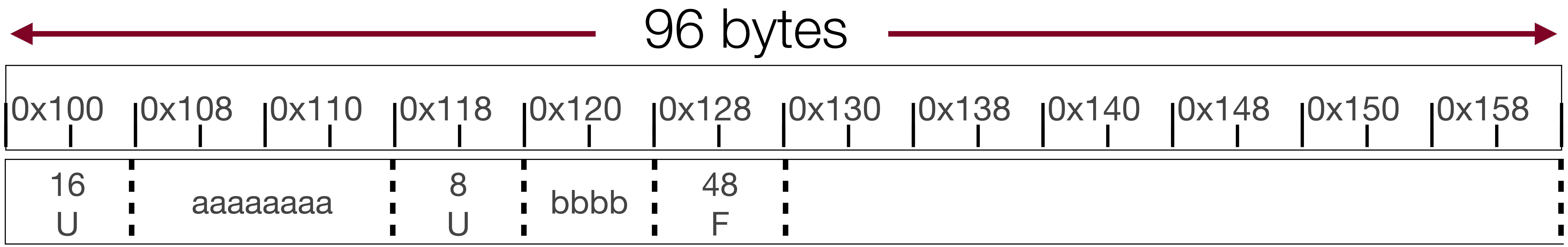
16 U | aaaaaaaa | 64 F

- This is where things start to get a bit tricky. The heap allocator has 96 bytes, and it needs to keep the free block information *in those 96 bytes* (I N C E P T I O N)
- In other words, the heap allocator is using part of the 96 bytes as housekeeping.
- Note here that there are now 16 bytes of overhead, because there are two *header blocks*.
- Here, the first 8-byte header block denotes 16 Used bytes, then there is a 16 byte payload, and then there is another 8-byte header to denote the 64 free bytes after.

Another Possibility

```
a = malloc(16);
b = malloc(8);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**

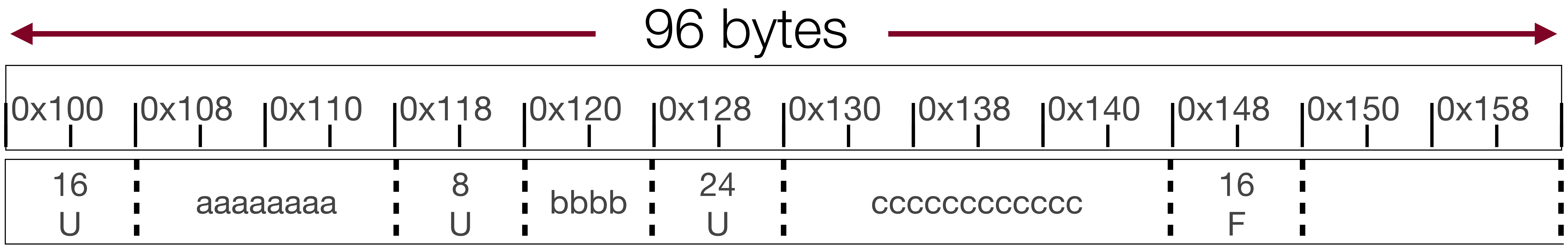


- We changed the header to reflect the fact that 8 bytes are going to to `b`, and we added a header for the remaining 48 bytes.
- Also, note that the pointer returned for `a` is 0x108, and the pointer returned for `b` is 0x120.

Another Possibility

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**

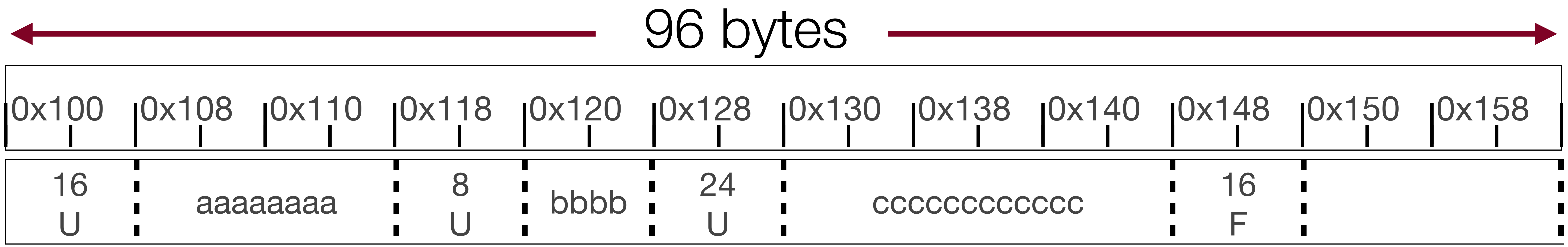


- Now we only have 16 bytes left for payloads…let's free some memory.

Another Possibility

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
free(a);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**

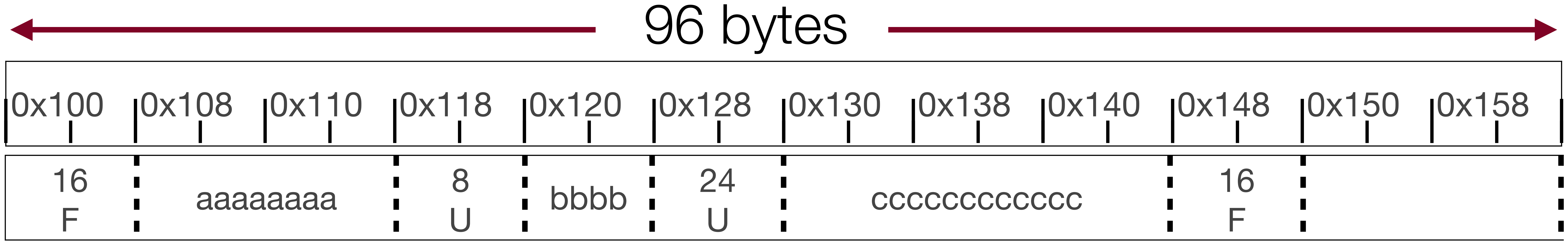


- Notice that 0x108 will be passed to free. How do we know how much to free?
  - We have to do some pointer arithmetic, so we can grab the 16 from address 0x100 (this diagram does not reflect the `free` yet).
- As you'll find out when writing your heap allocator: the arithmetic is super important.

Another Possibility

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
free(a);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**

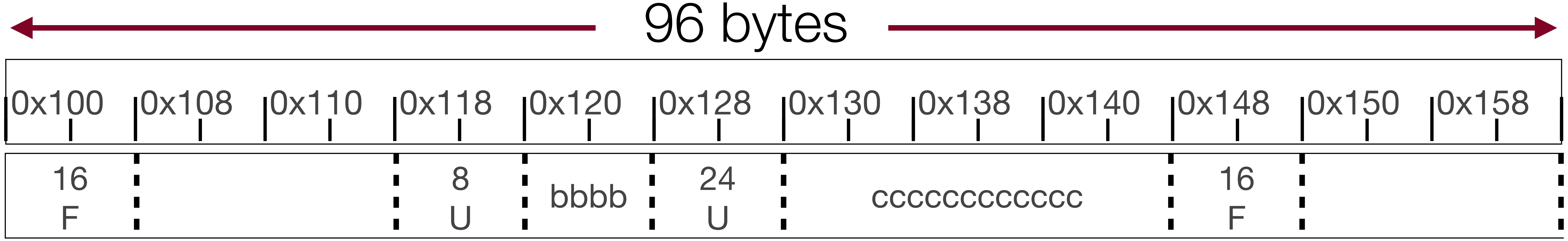


- The diagram now reflects the free.
- The change to the diagram was subtle — the *only* thing that changed was that the block header now says "F" (free) instead of "U" (used). This is because the data remains, but it can be written over any time after we reassign that block — this can cause bugs! For clarity sake, on the next page, we'll remove the `aaaaaaaa`, but know that the heap allocator doesn't wipe it clean (this another reason that `free` can be fast!)

Another Possibility

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
free(a);
free(c);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**

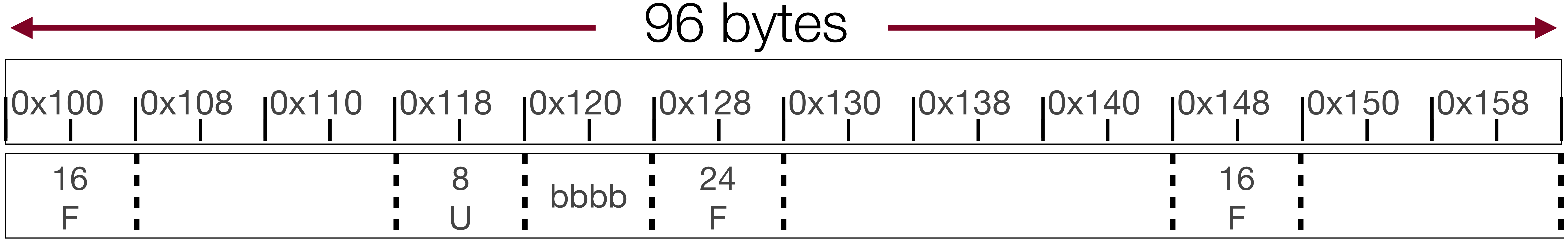


- Again, 0x130 is passed in to this free, so we need to figure out that we need to look at address 0x128 for the amount of bytes to free.
- On the next slide, we will remove the \`cccccccccccc\`, but again: it is *not* cleared out, and we're just doing this for the sake of clarity on the diagram.

Another Possibility

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
free(a);
free(c);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**

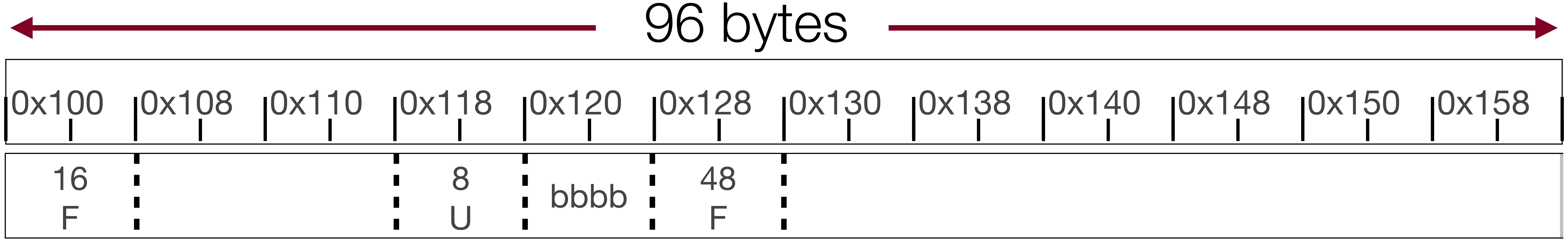


- This diagram shows one possible result of the `free`. Note that we have actually fragmented our free space! It looks like we only have a block of 24 bytes and then a block of 16 bytes to allocate, yet we should have a block of 48 bytes (we can save a header, too!)

Another Possibility

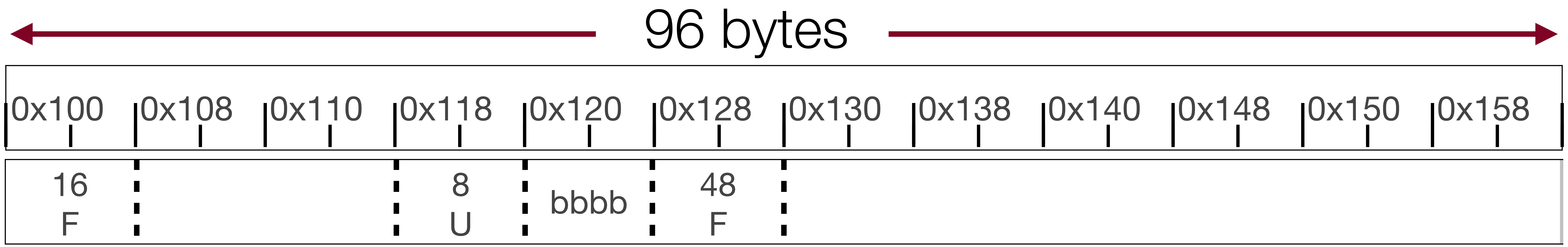
```
a = malloc(16);
b = malloc(8);
c = malloc(24);
free(a);
free(c);
```

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**



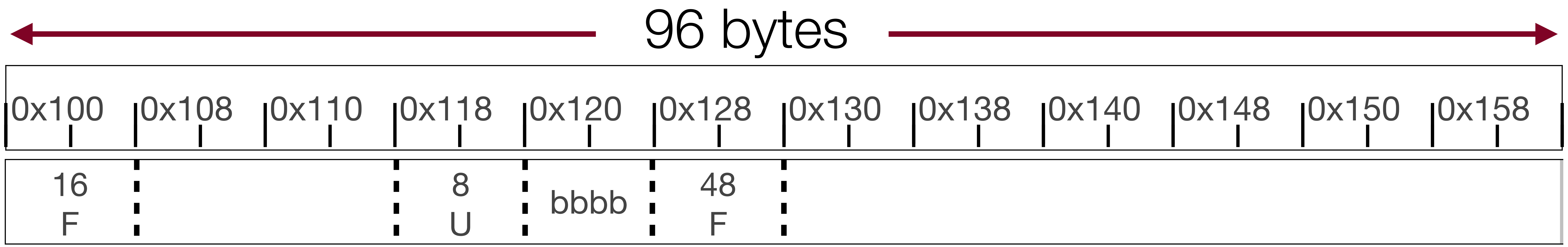
- When we combine free blocks, this is called *coalescing*, and it is an important tool that the heap allocator uses to keep memory as unfragmented as possible.
- We can't coalesce any more because `b` is in the middle, and we absolutely cannot move that block until the program we gave it to frees it.

Implicit Free List



- The method just demonstrated is called an "*implicit free list*," meaning that we have a list of free blocks that we can traverse to find an appropriate fit. The header holds the size of the block and whether it is free (F) or used (U) (note: the free and used information can be stored in 1 bit). To find the next available free block, we must look from the beginning and traverse the list in order.
- As blocks fill up, implicit free lists can cause **`malloc`** to be slow as the heap fills up — the linear search isn't a terrific method. (We will see another type next lecture!)

Implicit Free List



- Let's answer the questions we posed before:
  - How do we track the information in a block?
    - The header block that holds the bytes in the block and the state (free or used)
  - How do we organize/find free blocks?
    - Linear search, starting from the first block.
  - How do we pick which free block from available options?
    - If the block is free and has enough space we can choose it, though there are other options (covered in the next few slides).
  - What do we do with excess space when allocating a block?
    - If we can fit another header and still have at least a block's worth of space, we can do that. If we can't, it should just become part of the block we are allocating.
  - How do we recycle a freed block?
    - Mark it free, and coalesce if we can.

Placement: first-fit, next-fit, best-fit

The method we have described simply finds the first available block that is free and fits the request, and then starts from the beginning again on a future allocation. This is called a ***first-fit*** placement policy. One drawback is that you always have to start from the beginning of the heap, and it can be slow. Another drawback is that it can leave "splinters" (small free blocks) towards the beginning of the list. One advantage is that it leaves large blocks towards the end of the list, which allows for larger allocations if necessary.

A second method is called ***next-fit***, and was first proposed by Donald Knuth. With next-fit, you start looking for follow-on blocks after the location of the last allocation. If you found a suitable block before, you have a good chance to find another one in the same location. It is still not clear whether next-fit leads to better (or comparable) memory utilization.

The final method is called ***best-fit***, and relies on searching the entire heap to find a block that matches the requested allocation the best. The obvious drawback of best-fit is that it requires an exhaustive search of the list.

Splitting and Coalescing

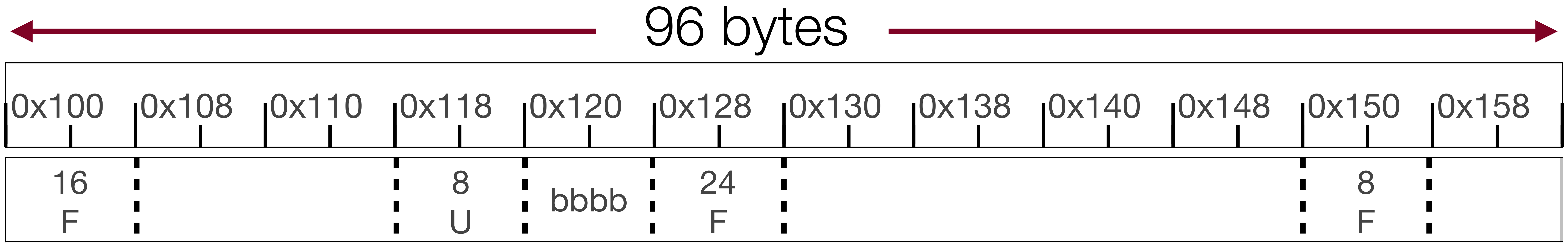
We have already described both splitting and coalescing as used in the implicit free list implementation.

Splitting the memory block is necessary when you have one large block to work with (which is what you will have for the heap allocator assignment). However, the heap allocator can request an increase in the size of the block of memory (using the `sbrk` *system call*), meaning that you could have a policy to use the entire block and just request more. But, we aren't going to cover that low level in this course.

Coalescing does not have to happen when you `free` — you can postpone coalescing until future `mallocs` or `reallocs`, and while it makes malloc a bit slower, frees are lighting fast.

More on Coalescing: coalescing backwards

Coalescing forwards is straightforward:



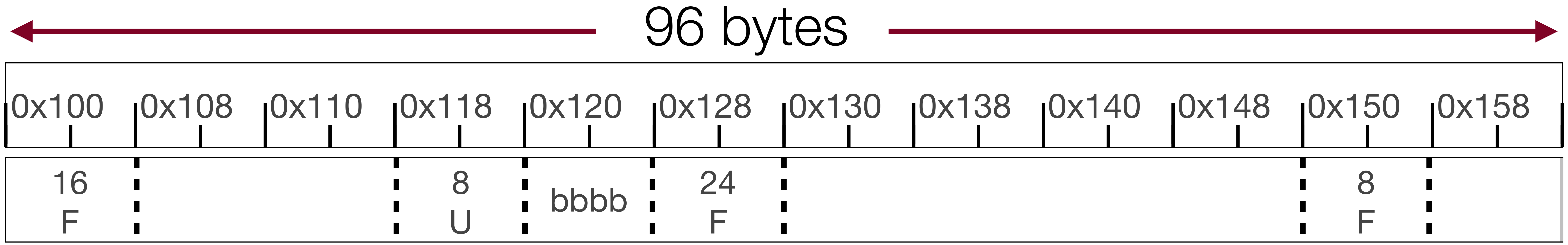
If we just freed the 24-byte block, we know exactly where the next block is in order to see if it (and subsequent blocks) are free.

However, what if we had just freed the 8 byte block? How could we coalesce the two blocks?

One way would be to look through the whole list from the beginning, keeping track of where the just-freed block is. But…this is slow.

More on Coalescing: coalescing backwards

Coalescing forwards is straightforward:



Another method (described by Knuth) is to keep a footer on each block, as well. The footer is identical to the header, but it refers to the prior bytes. The above list would look like this with headers and footers (assume we were using them the whole time, and we have to add more space because of the extra overhead):

Now, let's say we just free'd the 8 byte block at 0x168. We can look eight bytes back (to 0x160) at the footer for the 24-byte block, and we can see that it is also free, and we can coalesce.

More on Coalescing: coalescing backwards

Free'd block

More on Coalescing: coalescing backwards

Free'd block header

More on Coalescing: coalescing backwards

Footer for previous block (also free)

More on Coalescing: coalescing backwards

Entire free area

After coalescing backwards

Explicit Free List

One critical issue with the implicit list is the problem with the linear search to find free blocks.

The *explicit* free list solves this problem by keeping a linked list of free blocks embedded in the memory. This is best shown with an example. As before, let's start with an empty block of memory. With an explicit list, we keep a pointer to the first free block.

160 bytes

0x100 0x108 0x110 0x118 0x120 0x128 0x130 0x138 0x140 0x148 0x150 0x158 0x160 0x168 0x170 0x178

152
F

We use two blocks *in the payload* of the free block to point to the *next* and *previous* free blocks.

Explicit Free List

One critical issue with the implicit list is the problem with the linear search to find free blocks.

The *explicit* free list solves this problem by keeping a linked list of free blocks embedded in the memory. This is best shown with an example. As before, let's start with an empty block of memory. With an explicit list, we keep a pointer to the first free block.

We use two blocks *in the payload* of the free block to point to the *next* and *previous* free blocks.

Explicit Free List

One critical issue with the implicit list is the problem with the linear search to find free blocks.

The *explicit* free list solves this problem by keeping a linked list of free blocks embedded in the memory. This is best shown with an example. As before, let's start with an empty block of memory. With an explicit list, we keep a pointer to the first free block.

We use two blocks *in the payload* of the free block to point to the *next* and *previous* free blocks. In this case, there aren't any more free blocks, so they are `NULL` pointers.

Explicit Free List

```
a = malloc(16);
```

If we malloc 16, then we allocate as we would in the implicit list, but now we have a pointer to the next free block, and that block still has no previous or next free block.

Explicit Free List

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
```

We continue the process. Note that we must leave at least 16 bytes in a block to save room for pointers if we eventually free (e.g., `b` has more space than it requested).

Explicit Free List

```
a = malloc(16);
b = malloc(8);
c = malloc(24);
free(b);
```

Now when we free **`b`**, we point to the newly free'd memory, and update the pointers

Explicit Free List

Why is this better than the implicit free list?

Explicit Free List

Why is this better than the implicit free list?

- We can now traverse only the free blocks!
- This is much faster than traversing the whole list.
- For instance, if we now tried to malloc 24 bytes, we would only need to look through two blocks (0x118 and then 0x150) to find enough space.

- More on explicit free lists next lecture!

References and Advanced Reading

References:

- The textbook is the best reference for this material.
- Here are more slides from a similar course: https://courses.engr.illinois.edu/cs241/sp2014/lecture/06-HeapMemory_sol.pdf

Advanced Reading:

- Implementation tactics for a heap allocator: https://stackoverflow.com/questions/2946604/c-implementation-tactics-for-heap-allocators