CS 107
Lecture 14: Managing the Heap Part II

Friday, March 3, 2023

Computer Systems
Winter 2023
Stanford University
Computer Science Department

Lecturer: Chris Gregg

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

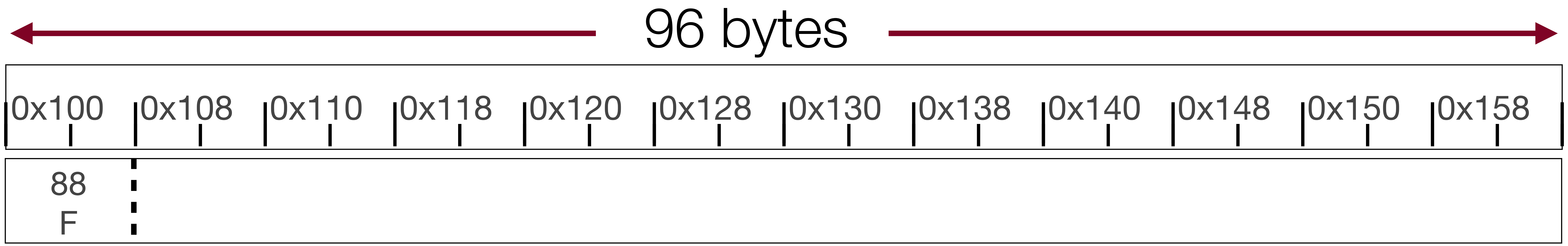
Today's Topics

- Programs from class: `/afs/ir/class/cs107/samples/lect14`
- Review of implicit heap allocator
- Explicit heap allocator
- More examples!
- Extra slides: Casting and `structs`

Implicit Free List (review from Friday)

- On Monday, we discussed the *implicit free list*, which is one common (though slow...) way to create a heap allocator. It uses what is called a **block header** to hold the information.

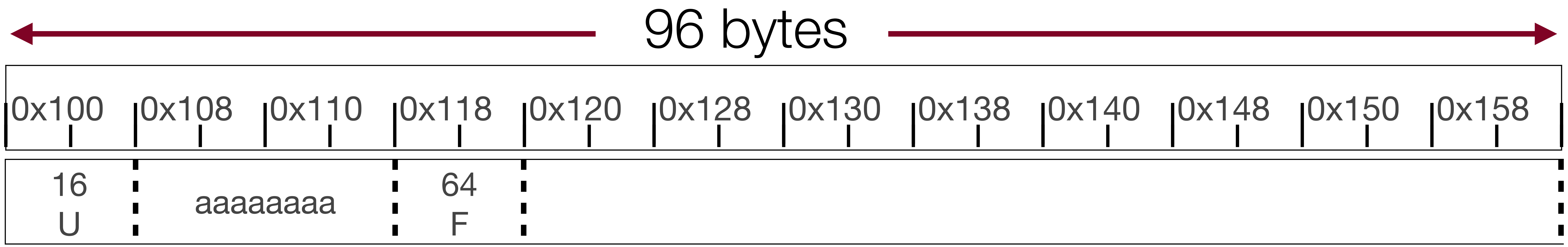
- The block header is actually *stored in the same memory area as the payload, and it generally precedes the payload.*



Implicit Free List (review from Monday)

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

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

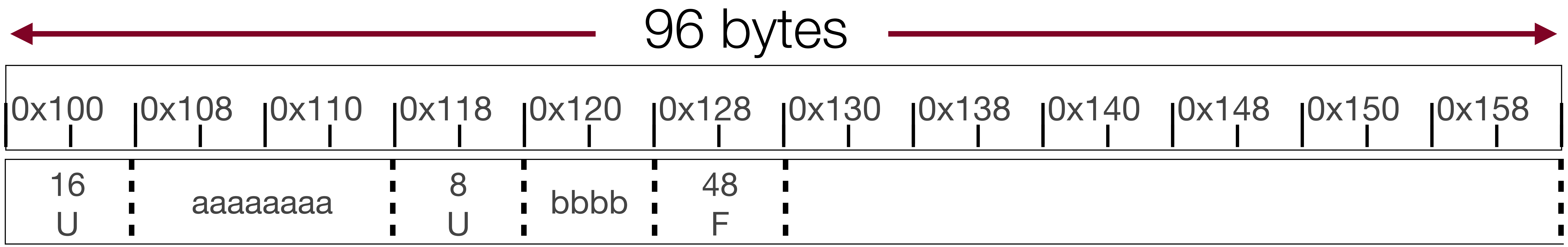


- This is where things start to get a bit tricky. The heap allocator has 96 bytes, and it needs to keep the free block information *in those 96 bytes* (I N C E P T I O N)
- In other words, the heap allocator is using part of the 96 bytes as housekeeping.
- Note here that there are now 16 bytes of overhead, 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.

Implicit Free List (review from Monday)

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

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

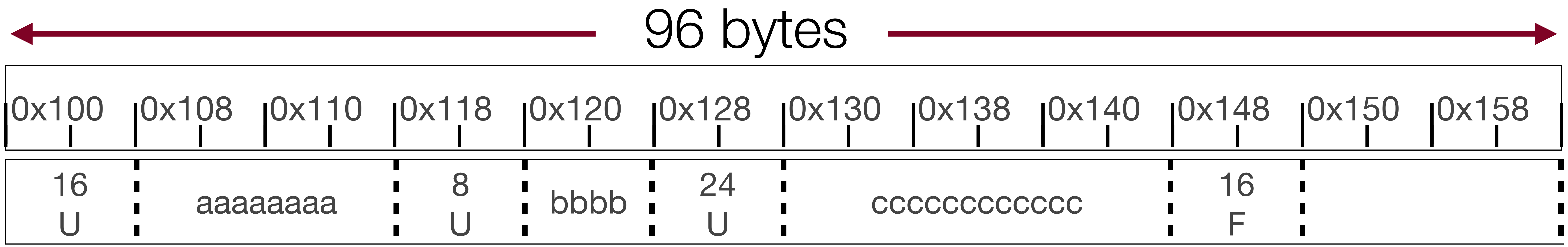


- We changed the header to reflect the fact that 8 bytes are going to to `b`, and we added a header for the remaining 48 bytes.
- Also, note that the pointer returned for `a` is 0x108, and the pointer returned for `b` is 0x120.

Implicit Free List (review from Monday)

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

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

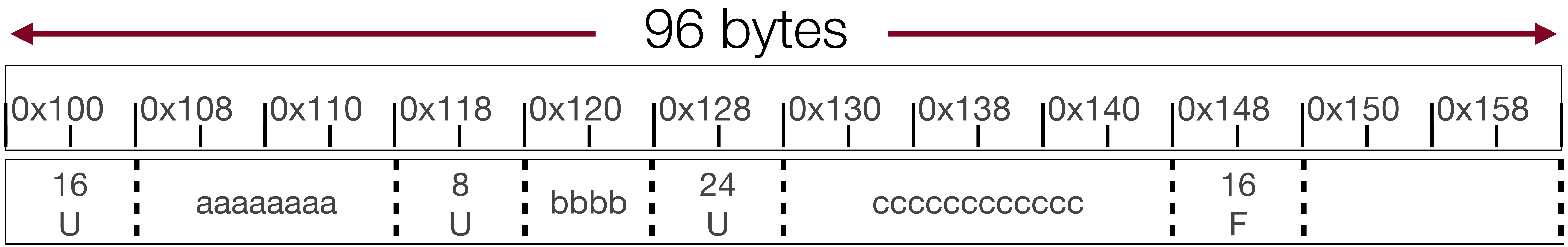


- Now we only have 16 bytes left for payloads…let's free some memory.

Implicit Free List (review from Monday)

```
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

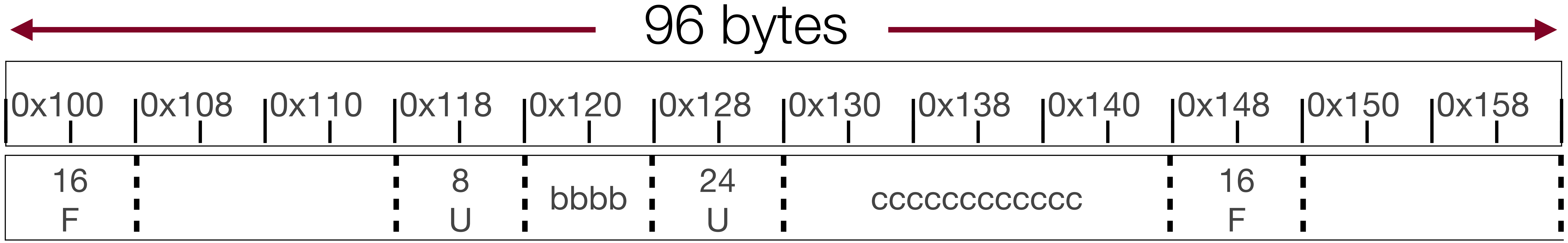


- Notice that 0x108 will be passed to free. How do we know how much to free?
  - We have to do some pointer arithmetic, so we can grab the 16 from address 0x100 (this diagram does not reflect the `free` yet).
- As you'll find out when writing your heap allocator: the arithmetic is super important.

Implicit Free List (review from Monday)

```
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**

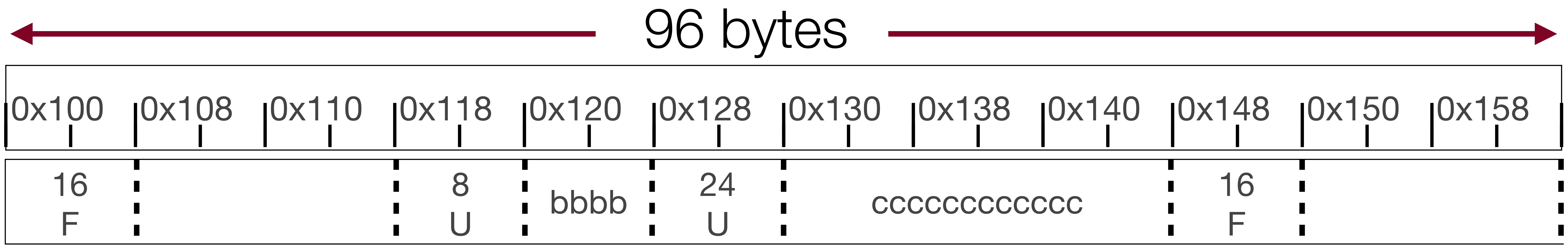


- The diagram now reflects the free.

Implicit Free List (review from Monday)

```
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

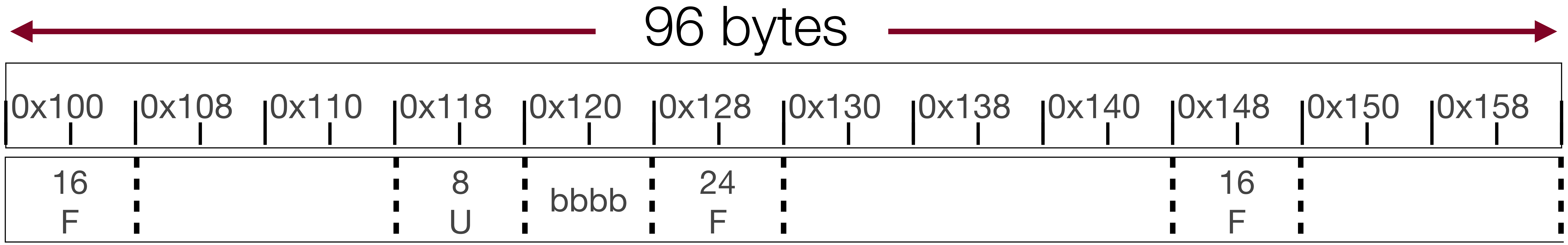


- Again, 0x130 is passed in to this free, so we need to figure out that we need to look at address 0x128 for the amount of bytes to free.

Implicit Free List (review from Monday)

```
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**

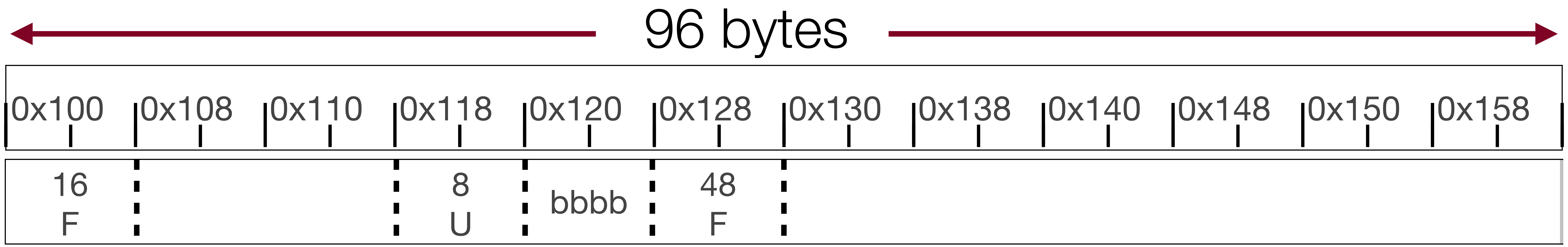


- One choice for the **`free`** is this diagram. Note that we have actually fragmented our free space! It looks like we only have a block of 24 bytes and then a block of 16 bytes to allocate, yet we should have a block of 48 bytes (we can save a header, too!)

Implicit Free List (review from Monday)

```
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.

Explicit Free List

One critical issue with the implicit list is the problem with the linear search to find free blocks (by the way: the implicit list just keeps a pointer to the first block for first-fit).

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 (by the way: the implicit list just keeps a pointer to the first block for first-fit).

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 (by the way: the implicit list just keeps a pointer to the first block for first-fit).

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. Notice something important: the two pointers we had just got eaten up by the payload!

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 freed memory, and update the pointers

Explicit Free List

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

Now when we free `b`, we point to the newly freed memory, and update the pointers

The newly freed block becomes the first free block (it is added to the beginning of the list)

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 Implicit and Explicit Heap Allocation

- For Assignment 6, you will be writing two heap allocators: implicit and explicit.
- Let's perform the following allocation and free on both allocators, to practice more. *Note:* the method outlined here doesn't have to be exactly how you implement the allocators (and in fact, they can be improved conceptually to be faster, etc.), but this should give you an overview of the differences and some details.

`a 0 24`	allocation 0 for 24 bytes
`a 1 20`	allocation 1 for 20 bytes
`a 2 16`	allocation 2 for 16 bytes
`f 1`	free 1
`a 3 32`	allocation 3 for 32 bytes
`a 4 12`	allocation 4 for 12 bytes
`f 3`	free 3
`r 2 60`	reallocate 2 to 60 bytes

Implict Heap Allocation

- We will start with the following free heap

Heap size: 256 bytes

0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	f0	f8

- Headers will take up 8 bytes.
- Addresses are in hex

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- After initialization, header holds the value of the whole free area, and designates it as (F)ree

Heap size: 256 bytes

0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	f0	f8
248 F																															

pointer to heap

- Addresses are in hex
- Byte allocations are in decimal

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- Allocation of 24 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Allocation of 24 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Allocation of 20 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**
2. Go to next block (use pointer arithmetic to get there)

Implict Heap Allocation

- Allocation of 20 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Allocation of 20 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Why 24 bytes? **8-byte alignment**

Is this strictly necessary here? No — the header blocks don't have to be aligned to 8-byte boundaries. But it may make it easier! Remember, all **allocations** must be on an 8-byte alignment.

Implict Heap Allocation

- Allocation of 16 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**
2. Go to next block (use pointer arithmetic to get there)

Implict Heap Allocation

- Allocation of 16 bytes

Heap size: 256 bytes

1. Is the block free? **no**
2. Go to next block (use pointer arithmetic to get there)

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- Allocation of 16 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Allocation of 16 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Free 1

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- Free 1

Heap size: 256 bytes

1. Mark as free

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- Allocation of 32 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**
2. Go to next block (use pointer arithmetic to get there)

Implict Heap Allocation

- Allocation of 32 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **no**
3. Go to next block (use pointer arithmetic to get there)

Implict Heap Allocation

- Allocation of 32 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**
2. Go to next block (use pointer arithmetic to get there)

Implict Heap Allocation

- Allocation of 32 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Allocation of 32 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for the request? **yes**

use this block

Implict Heap Allocation

- Allocation of 12 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**
2. Go to next block (use pointer arithmetic to get there)

Implict Heap Allocation

- Allocation of 12 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space for request? **yes**

use this block

Implict Heap Allocation

- Allocation of 12 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

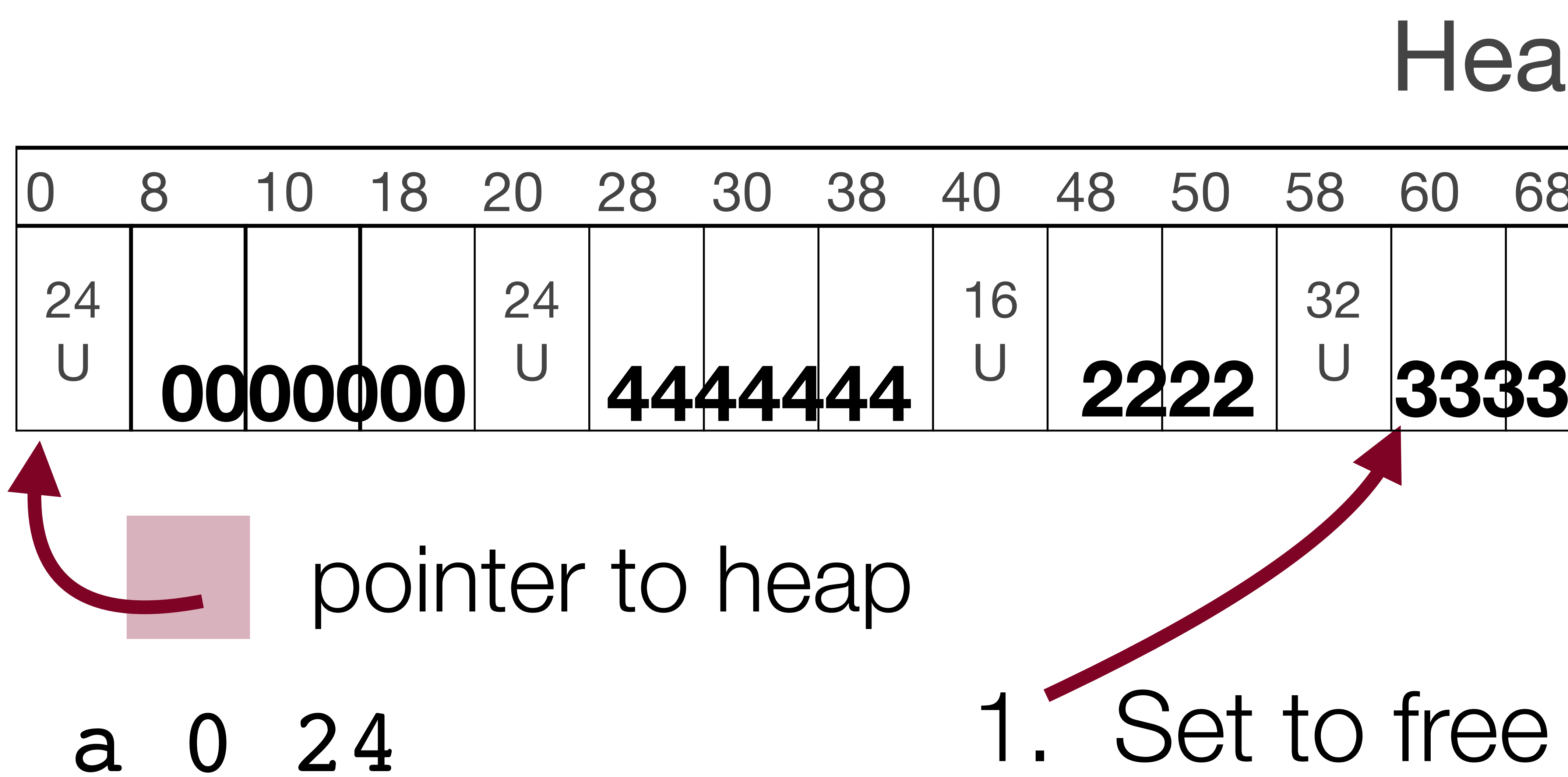
1. Is the block free? **yes**
2. Is there enough space for request? **yes**

use this block

Why not 12 or 16? **alignment and not enough space after**

Implict Heap Allocation

- Free 3




```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- Free 3

Heap size: 256 bytes

1. Set to free

No coalescing for implicit!

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Enough space after?

yes (take as much space to the right as possible, but only for realloc, not malloc)

If no, we would have had to move the block by searching through he whole list for a spot with enough space (see the next slide)

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**

   Check next block

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**

Check next block

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**

   Check next block

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes

0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	f0	f8
24 U	**0000000**			24 U	**4444444**			16 U	**2222**		32 F					16 U	**5555**		96 F												

pointer to heap

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough space? **no**

Check next block

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **no**

   Check next block

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough room? **yes**

   Move 2, free, and update

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough room? **yes**

   Move 2, free, and update

Assume, for a moment, that there had not been space. We would have started searching

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough room? **yes (take as much as possible to the right)**

Back to the version where we *do* have space, if we count all the possible free space to the right

Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

1. Is the block free? **yes**
2. Is there enough room? **yes (take as much as possible to the right)**
3. Expand block

Back to the version where we *do* have space, if we count all the possible free space to the right

Explicit Heap Allocation

- After initialization, header holds the value of the whole free area, and designates it as (F)ree

Heap size: 256 bytes

pointer to free block

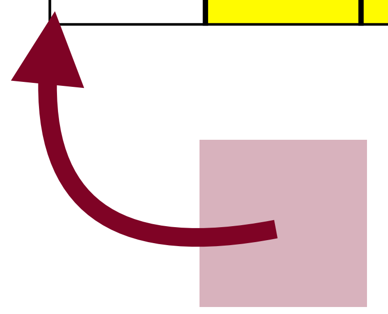
```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Addresses are in hex
- Byte allocations are in decimal
- The pointer to the heap is always to a free block!
- For explicit, we will have two pointers that *live in the potential payload area* (in yellow).
- The pointers are the previous and next pointers for our linked list, although in this case they will both be NULL because there aren't any other nodes.

Explicit Heap Allocation

- Allocate 24 bytes

Heap size: 256 bytes



pointer to free block

- Is there enough space? **yes**

use this block

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Explicit Heap Allocation

- Allocate 24 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Is there enough space? **yes**

  use this block

- The pointers become part of the allocated space, because we don't need them now!
- We update the heap pointer to point to a free block.

Explicit Heap Allocation

- Allocate 20 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Is there enough space? **yes**

  use this block

- The pointers become part of the allocated space, because we don't need them now!
- We update the heap pointer to point to a free block.

Explicit Heap Allocation

- Allocate 20 bytes

Heap size: 256 bytes

- Is there enough space? **yes**

use this block

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- The pointers become part of the allocated space, because we don't need them now!
- We update the heap pointer to point to a free block.

Explicit Heap Allocation

- Allocate 16 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Is there enough space? **yes**

  use this block

- The pointers become part of the allocated space, because we don't need them now!
- We update the heap pointer to point to a free block.

Explicit Heap Allocation

- Allocate 16 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Is there enough space? **yes**

  use this block

- The pointers become part of the allocated space, because we don't need them now!
- We update the heap pointer to point to a free block.

Explicit Heap Allocation

- Free 1

Heap size: 256 bytes

- Free, **and add to linked list**

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Explicit Heap Allocation

- Free 1

Heap size: 256 bytes

pointer to free block

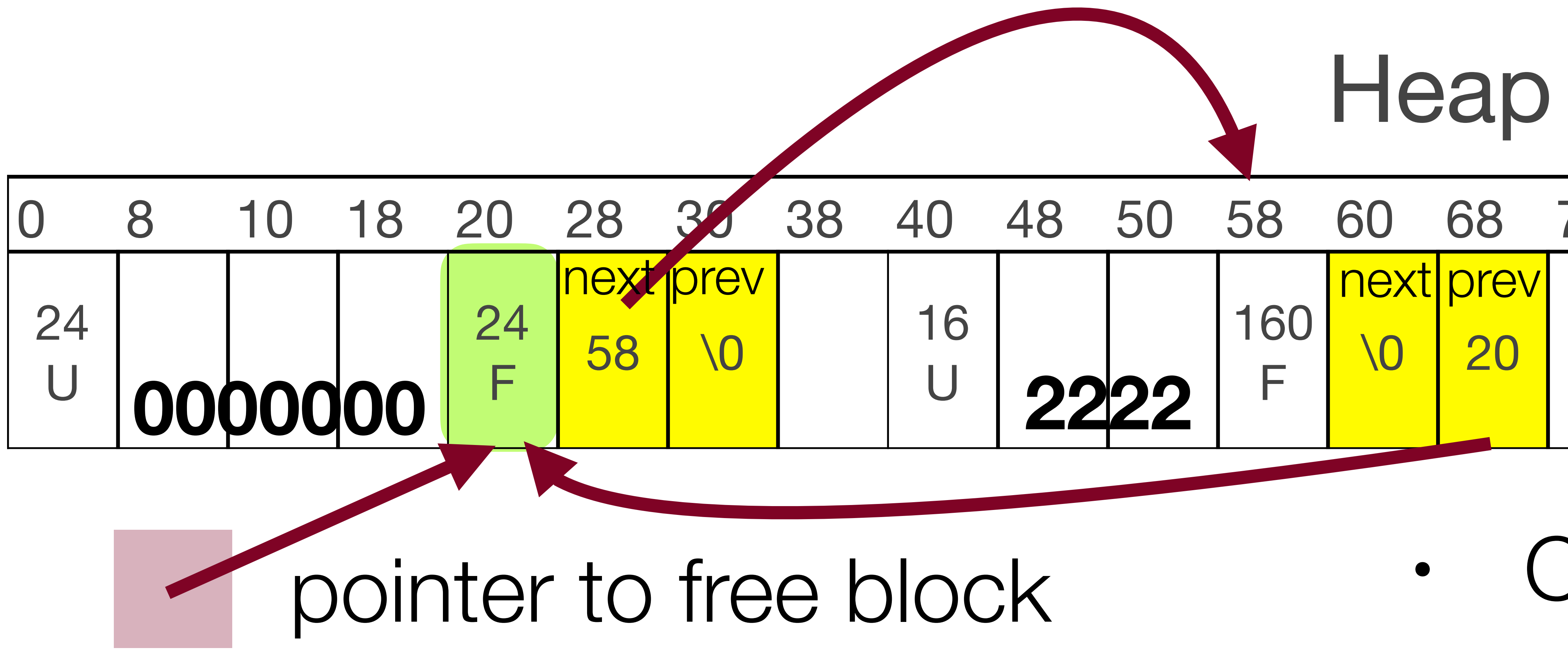
```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Free, **and add to linked list**
- (remember to update all necessary doubly-linked list pointers!)

Explicit Heap Allocation

- Allocate 32 bytes

Heap size: 256 bytes




```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Check head of list (green) — is there enough room? **no**

Follow next pointer

Explicit Heap Allocation

- Allocate 32 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Explicit Heap Allocation

- Allocate 32 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Check node (green) in list — is there enough room? **yes**

add here and update list

Explicit Heap Allocation

- Allocate 32 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Check node in list — is there enough room? **yes**

add here and update list

Explicit Heap Allocation

- Allocate 12 bytes

- Check node (green) in list — is there enough room? **yes**

add here and update list

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Explicit Heap Allocation

- Allocate 12 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Check node (green) in list — is there enough room? **yes**

add here and update list

Explicit Heap Allocation

- Free 3

Heap size: 256 bytes

pointer to free block

- Free 3, **and coalesce**

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Explicit Heap Allocation

- Free 3

Heap size: 256 bytes

pointer to free block

- Free 3, **and coalesce**
- Make sure to update linked list

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

Explicit Heap Allocation

- Reallocate 2 to 60 bytes

Heap size: 256 bytes


```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Enough space after? **yes**
- No move necessary, but we do have to do our updates.

Explicit Heap Allocation

- Reallocate 2 to 60 bytes

Heap size: 256 bytes

pointer to free block

```
a 0 24
a 1 20
a 2 16
f 1
a 3 32
a 4 12
f 3
r 2 60
```

- Enough space after? **yes**
- No move necessary, but we do have to do our updates.

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

Extra Slides

Struct Casting

For your heap allocator assignment, you might want to consider casting all of your `void *` pointers to `structs`, as this will make it easier to debug. It will also make the code a lot cleaner. Let's see an example of casting structs that helps see how it is done.

First, here is a struct for some student information (`struct_ex.c` in `samples/lect15`):

```
typedef struct student_info {
    char email[32];
    int labs_attended;
    double assignment_avg;
    double midterm;
    double final;
} student_info;
```

We can typedef the struct to make it easier to work with. Now we can refer to it as, simply, `student_info`.

Struct Casting

Next, let's write an update_info function but without the benefit of any type information for the struct:

```
void update_info(void *student, char *email, int labs_attended,
                 double assignment_avg, double midterm, double final)
{
    // we have a void *, but we can simply cast it
    student_info *si = (student_info *)student;
    strcpy(si->email, email);
    si->labs_attended = labs_attended;
    si->assignment_avg = assignment_avg;
    si->midterm = midterm;
    si->final = final;
}
```

We simply cast the `void *` pointer to our data type, and the `struct` just works.

Struct Casting

We can do the same thing for a print function:

```
void print_student(void *student)
{
    // again, we have a void *, but we can just cast
    student_info *si = (student_info *)student;
    printf("Email: %s\n",si->email);
    printf("Labs attended: %d out of %d\n",si->labs_attended,NUM_LABS);
    printf("Assignment Average: %g\n",si->assignment_avg);
    printf("Midterm: %g\n",si->midterm);
    printf("Final: %g\n",si->final);
    double overall_avg = (si->labs_attended / (double)NUM_LABS * 10 +
                          si->assignment_avg * 0.4 +
                          (si->midterm * 0.33 + si->final * 0.67) * 0.5);
    printf("Overall average: %g\n\n",overall_avg);
}
```

Struct Casting

Here is where it gets interesting. In **`main`**, we're just going to grab a typeless block of data:

```
#define BIG_BLOCK 10000

int main(int argc, char **argv)
{
    // big block of data with no type information :(
    void *student_data = malloc(BIG_BLOCK);

    // let's add some students
    // let's add one at the beginning of the block of data
    update_info(student_data, "cgregg@stanford.edu", 7, 92.0, 83.4, 94.0);

  ...
```

We can add a student to *anywhere we want* in that block of data (though this might not be ideal for alignment purposes)

Struct Casting

Again, we can add a student anywhere in the data!

```
    // it's a big block of data, so we can add a student anywhere :O
    // let's add one 6543 bytes down the line. Remember, we do have to
    // cast if we want pointer arithmetic
    void *random_location = (char *)student_data + 6543;

    update_info(random_location, "super_student@stanford.edu", 8, 100.0, 100.0, 100.0);

    // let's print the two students
    print_student(student_data);
    print_student(random_location);

    free(student_data);
    return 0;
}
```

We just added a student at location 6543 in our block of data -- we didn't have to put the data on what would be a struct boundary in an array of structs.

Struct Casting

Let's look at a gdb trace of the program:

```
$ gdb struct_ex
(gdb) break main
Breakpoint 1 at 0x400756: file struct_ex.c, line 45.
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/
cgregg/tmp/lect15/struct_ex

Breakpoint 1, main (argc=1, argv=0x7fffffffea78) at
struct_ex.c:45
(gdb) n
47      void *student_data = malloc(BIG_BLOCK);
(gdb)
52      update_info(student_data, "cgregg@stanford.edu",
7, 92.0, 83.4, 94.0);
(gdb) x/10gx student_data
0x602010: 0x0000000000000000  0x0000000000000000
0x602020: 0x0000000000000000  0x0000000000000000
0x602030: 0x0000000000000000  0x0000000000000000
0x602040: 0x0000000000000000  0x0000000000000000
0x602050: 0x0000000000000000  0x0000000000000000
(gdb) n
```

We can look at the bytes in the **`student_data`** block. One compact way to do this is with the "**`x/gx`**" command.

Before we update the student, the data happens to be all 0s.

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763  0x2e64726f666e6174
0x602020: 0x0000000000756465  0x0000000000000000
0x602030: 0x0000000000000007  0x4057000000000000
0x602040: 0x4054d9999999999a  0x4057800000000000
0x602050: 0x0000000000000000  0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

We put in the following information: email: cgregg@stanford.edu, number of labs: 7.

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763   0x2e64726f666e6174
0x602020: 0x0000000000756465   0x0000000000000000
0x602030: 0x0000000000000007   0x4057000000000000
0x602040: 0x4054d9999999999a   0x4057800000000000
0x602050: 0x0000000000000000   0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

We put in the following information: email: cgregg@stanford.edu, number of labs: 7.

The long values are read in reverese: `0x60210` is the `0x63` in the first block, which is a "c".

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763   0x2e64726f666e6174
0x602020: 0x0000000000756465   0x0000000000000000
0x602030: 0x0000000000000007   0x4057000000000000
0x602040: 0x4054d9999999999a   0x4057800000000000
0x602050: 0x0000000000000000   0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

Address **`0x60211`** is the **`0x67`** in the first block, which is a "g".

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763  0x2e64726f666e6174
0x602020: 0x0000000000756465  0x0000000000000000
0x602030: 0x0000000000000007  0x4057000000000000
0x602040: 0x4054d9999999999a  0x4057800000000000
0x602050: 0x0000000000000000  0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

Address **`0x60212`** is the **`0x72`** in the first block, which is a "r".

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763   0x2e64726f666e6174
0x602020: 0x0000000000756465   0x0000000000000000
0x602030: 0x0000000000000007   0x4057000000000000
0x602040: 0x4054d9999999999a   0x4057800000000000
0x602050: 0x0000000000000000   0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

Address **`0x60217`** is the **`0x73`** in the first block, which is a "s" (in stanford.edu)

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763   0x2e64726f666e6174
0x602020: 0x0000000000756465   0x0000000000000000
0x602030: 0x0000000000000007   0x4057000000000000
0x602040: 0x4054d9999999999a   0x4057800000000000
0x602050: 0x0000000000000000   0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

We set aside 32 bytes for the email address, which spans from **`0x602010`** to **`0x60202f`**.

Struct Casting

Let's look at a gdb trace of the program:

```
(gdb) n

(gdb) x/10gx student_data
0x602010: 0x7340676765726763   0x2e64726f666e6174
0x602020: 0x0000000000756465   0x0000000000000000
0x602030: 0x0000000000000007   0x4057000000000000
0x602040: 0x4054d9999999999a   0x4057800000000000
0x602050: 0x0000000000000000   0x0000000000000000
(gdb) n
```

After we update the student, the info is located in **`student_data`**, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...

Our next data is the **`int`** for the number of labs. It turns out that the alignment for other fields is going to be on an 8-byte boundary, so the struct actually takes 8 bytes for the int. Again, the addresses are backwards, because the number itself is in little-endian format. So, `0x602030` is the `0x07` byte, `0x0602031` is the `0x00` byte to the left of the `0x07`, etc.