## 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()





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

### Today's Topics



- way to create a heap allocator. It uses what is called a **block header** to hold the information.
- generally precedes the payload.

					96
0x100	0x108	0x110	0x118	0x120	0x1
88 F					

•On Monday, we discussed the *implicit free list*, which is one common (though slow...)

•The block header is actually stored in the same memory area as the payload, and it





#### malloc(16); a



- needs to keep the free block information in those 96 bytes (INCEPTION)
- blocks.

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	
b	0xfffe808	
a	0xfffe800	<b>0x108</b>

•This is where things start to get a bit tricky. The heap allocator has 96 bytes, and it •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

•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.



- malloc(16);  $\blacksquare$ a
- malloc(8); b



- added a header for the remaining 48 bytes.
- is 0x120.

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	
b	0xfffe808	<b>0x120</b>
a	0xfffe800	<b>0x108</b>

96 bytes 0x150 10x128 10x130 0x138 0x140 0x148 J0x158 48

•We changed the header to reflect the fact that 8 bytes are going to to b, and we

•Also, note that the pointer returned for a is 0x108, and the pointer returned for b



- malloc(16); Ξ a
- malloc(8); b
- malloc(24);= C



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

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	<b>0x130</b>
b	0xfffe808	<b>0x120</b>
a	0xfffe800	<b>0x108</b>

96 bytes

28	0x130	0x138	0x140	0x148	0x150	0x158
4 J	CCCCCCCCCC		16 F			



- malloc(16); a
- malloc(8); b =
- malloc(24);C

free(a);



 Notice that 0x108 will be passed to free. How do we know how much to free? 0x100 (this diagram does not reflect the free yet). •As you'll find out when writing your heap allocator: the arithmetic is super important.

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	<b>0x130</b>
b	0xfffe808	<b>0x120</b>
а	0xfffe800	<b>0x108</b>

96 bytes 0x130 0x138 0x158 0x148 0x150 0x140 24 16 CCCCCCCCCCC

•We have to do some pointer arithmetic, so we can grab the 16 from address



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

free(a);



•The diagram now reflects the free.

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	<b>0x130</b>
b	0xfffe808	<b>0x120</b>
a	0xfffe800	<b>0x108</b>

96 bytes |0x128 |0x130 |0x138 |0x140 |0x148 |0x150 |0x158 24 cccccccccc F = 16 U = 16





- malloc(8); b
- malloc(24);C

free(a);

free(c);



•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.

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	<b>0x130</b>
b	0xfffe808	<b>0x120</b>
a	0xfffe800	<b>0x108</b>

96 bytes 0x130 0x138 0x140 0x148 0x150 J0x158 16 **CCCCCCCCCC** F







16 bytes to allocate, yet we should have a block of 48 bytes (we can save a header, too!)

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	<b>0x130</b>
b	0xfffe808	<b>0x120</b>
a	0xfffe800	<b>0x108</b>

96 bytes 0x138 0x130 0x140 0x148 0x150 J0x158 24 16

•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







- that the heap allocator uses to keep memory as unfragmented as possible.
- move that block until the program we gave it to frees it.

	Address	Value
е	0xfffe820	
d	0xfffe818	
С	0xfffe810	<b>0x130</b>
b	0xfffe808	<b>0x120</b>
a	0xfffe800	<b>0x108</b>

96 bytes 0x130 0x150 0x138 10x140 10x148 J0x158

•When we combine free blocks, this is called *coalescing*, and it is an important tool •We can't coalesce any more because b is in the middle, and we absolutely cannot



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.

#### 160 bytes 0x138 0x140 0x148 0x150 0x158 0x160 0x168 0x170







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.

#### 160 bytes 0x150 0x138 0x140 0x148 0x158 0x160 0x168 0x170







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.

> 0x100 First Free Block

$\checkmark$							160
0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x13
152	P:	• N:					
F	0x0	0x0					

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.

#### ) bytes 0x158 0x160 0x170 38 <sub>1</sub>0x140 <sub>1</sub>0x148 <sub>1</sub>0x150 0x168









#### malloc(16); a =

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!

#### 160 bytes 0x138 0x140 0x148 0x150 0x158 <sub>1</sub>0x160 0x168 <sub>1</sub>0x170







- malloc(16); a =
- malloc(8); b
- malloc(24); C

room for pointers if we eventually free (e.g., b has more space than it requested).



## We continue the process. Note that we must leave at least 16 bytes in a block to save





#### malloc(16); a =

- malloc(8); b
- malloc(24); C

#### free(b);

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







- a = malloc(16);
- = malloc(8); b
- = malloc(24); С
- 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)

) bytes
---------

38  0x140  0x148	0x150	0x158	0x160	0x168	0x170	
CCCCCCCCCCC	48 F	P: 0x118	N: 0x0			







#### Why is this better than the implicit free list?



#### Explicit Free List

#### 160 bytes

38 0x140 0x148	0x150	0x158	0x160	0x168	0x170	
000000000000000000000000000000000000000	48 F	P: 0x118	N: 0x0			





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.
- through two blocks (0x118 and then 0x150) to find enough space.



# •For instance, if we now tried to malloc 24 bytes, we would only need to look

$h_{\rm trans}$						
J Dytes						
38 <sub>1</sub> 0x140 <sub>1</sub> 0x148	0x150	0x158	0x160	0x168	0x170	
000000000000000000000000000000000000000	48 F	P: 0x118	N: 0x0			





## More on Implicit and Explicit Heap Allocation

For Assignment 6, you will be writing two heap allocators: implicit and explicit. should give you an overview of the differences and some details.

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

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

ytes

ytes

ytes

ytes

ytes

/tes



We will start with the following free heap •

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	fC

- Headers will take up 8 bytes. • •
- a 0 24
- a 1 20
- a 2 16
- f 1
- 3 32 a
- 12 4 a
- f 3

r 2 60

#### Heap size: 256 bytes

Addresses are in hex





ullet

#### After initialization, header holds the value of the whole free area, and designates it as (F)ree Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fO	f8

Addresses are in hex Byte allocations are in decimal





Allocation of 24 bytes ullet

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

2. Is there enough space for the request? yes

use this block





Allocation of 24 bytes ullet

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

2. Is there enough space for the request? yes

use this block





• Allocation of 20 bytes

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

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





Allocation of 20 bytes ullet

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

2. Is there enough space for the request? yes

use this block





Allocation of 20 bytes  $\bullet$ 

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

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.





• Allocation of 16 bytes

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

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





• Allocation of 16 bytes

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

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





Allocation of 16 bytes ullet

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

2. Is there enough space for the request? yes

use this block





Allocation of 16 bytes ullet

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

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

use this block





• Free 1

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC





• Free 1

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC





Allocation of 32 bytes  $\bullet$ 

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

2. Go to next block (use pointer arithmetic to get there)





Allocation of 32 bytes  $\bullet$ 

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

3. Go to next block (use pointer arithmetic to get there)




Allocation of 32 bytes ullet

### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	h8	<u>_</u>	68	0b	8h	e0	68	fC
10															

2. Go to next block (use pointer arithmetic to get there)





Allocation of 32 bytes ullet

### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

2. Is there enough space for the request? yes

use this block





Allocation of 32 bytes ullet



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
33	120 F														

2. Is there enough space for the request? yes

use this block





Allocation of 12 bytes  $\bullet$ 

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
33	120 F														

2. Go to next block (use pointer arithmetic to get there)





Allocation of 12 bytes ullet



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
33	120 F														

# 2. Is there enough space for request? yes

use this block





Allocation of 12 bytes ullet



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
33	120 F														

2. Is there enough space for request? **yes** 

## use this block

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





• Free 3

0	8	3 -	10 -	8	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	fC
2 เ	24 J	000	000	00	24 U	44	44	444	16 U	22	222	32 U	33	333	333:	333	120 F														
			р	oir	nter	r to	he	eap																							
	a	0	24	2					1		Set	to	fre	e																	
	a	1	20	)																											
	a	2	16	- )																											
	f	1																													
	a	3	32																												
	a	4	12																												
	f	3																													
	r	2	60																												TELANY





Free 3 •

0	8	10	18	20	28	30	38	40	48 5	) 58	60	68	70	78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	f(
24 U	0	000	000	24 U	44	44	444	16 U	222	32 <b>2</b> F	2				120 F														
		Ķ	ooir	nter	r to	he	eap																						
ć	a (	) 2	4					1.	Se	et to	o fre	e																	
ð	a [	L 2	0						Nc		ale	SC	ina	f∩r	r im	noli	cit												
ð	a 2	2 1	6													'P''													
t		L																											
6	a S	33	2																										
ć	a 4	ł 1	2																										
ſ		3																											li
1		2 6	0																										TELAND





Realloc 2 to 60 bytes  $\bullet$ 



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
	120 F														

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)







Realloc 2 to 60 bytes ullet



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
	16 U	55	555	96 F											

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





Realloc 2 to 60 bytes ullet

#### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
	16 U	55	555	96 F											

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





Realloc 2 to 60 bytes ullet



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
	16 U	55	555	96 F											

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





Realloc 2 to 60 bytes •

0	8	10	) 18	20	28	30	38	40	48	50 క	58	60	68	70	78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
24 U	0	000	0000	<b>)</b> 24	44	444	144	16 U	22	22	32 F					16 U	55	555	96 F											
			poi	nte	r tc	he	eap																							
6	a (	0	24					1.		s th	е	blo	ck	fre	e?	, <b>λ</b> €	<b>S</b>	_												
õ	<b>£</b>	1	20					2.		s th	er	e e	nc	ug	h s	spa	ceʻ	?	no											
6	£ 2	2	16								C	Che	ck	i ne	ext	blo	ock													
f		1						٨	~~			fa					<b>n</b> t	+h		+b		- h			-+		• •			
6	A .	3	32					A	55	um	ie,	, 10				ne	nı,		้อเ			; n	au		<b>J</b> L I		en			
6	£ 4	4	12							spa		<b>e</b> . \	We	e v	/01	JIC	ha	IVE	9 <b>S</b> 1	tar	tec		eal	rcr	nn	g				
f	-	3																												
۲	~ '	2	60																											ELAND





Realloc 2 to 60 bytes  $\bullet$ 



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
	16 U	55	555	96 F											

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





Realloc 2 to 60 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	fC
24 U	00	000	<b>000</b>	24 U	44	444	444	16 U	22	22	32 F					16 U	5	555	96 F											
		p	ooir	ntei	r tc	he	eap																							
a	0	2	4					1	15	s th	e	blo	ck	fre	e?	γe	<b>?</b> S													
a	1	2	0					2	.	s th	ner	re e	enc	DUC	gh r	00	m	<b>) y</b>	es											
a	2	1	6							Mc	VE	2	, fr	ee,	ar	ndı	JDO	dat	e											
f	1											•		,											_ #					
a	3	3	2					A	SS	um	<b>ie</b> ,	, TO	or a	an		me	nτ,	, τr	ιατ	τη	ere	e n	ad			De	en			
a	4	1	2							spa	ac	<b>e</b> .	We	e N	/01	JID	ha	IVE	e st	tar	tec	S	ea	rch	nin	g				
f	3																													
r	2	6	$\mathbf{\cap}$																											TAND





Realloc 2 to 60 bytes •

 $\mathbf{U}$ 

0	8	; 1	10 18	3 20	28	30	38	40	48	50 5	3 60	D 68	70	78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
2 l	4 J	00(	<b>)00C</b>	24 U	44	444	444	16 F			82 F				16 U	5	555	64 F	22	2222	222	222	222	222	222	222	24 F		
			ро	inte	er to	he	eap																						
	a	0	24					1.	Is	s the	bl	ock	c fre	e?	, <b>λ</b> ε	<b>?</b> S													
	a	1	20					2	. 15	s the	ere	enc	DUC	gh r	00	m	<b>y</b>	es											
	a	2	16							Mov	/e	2, fr	ee.	ar	nd I	UD	dat	е											
	f	1										,	,																
	a	3	32					A	SS	um	Э, Т	or	a n		me	nτ,	, τη	ιατ	τη	ere	e n	ad	no	οτ I	oee	en			
	a	4	12						S	spa	ce.	. We	e N	/01	JID	ha	ave	st	ar	tec	S	eal	rch	nin	g				
	f	3																											
	r	2	60																										ELAND





Realloc 2 to 60 bytes •

0 8	-	10 18	3 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	fC
24 U	000	0000	0 24 U	44	144	444	16 U	22	222	32 F					120 F														
		ро	inte	r tc	o he	eap	_						<b>_</b>																
a	0	24					1	. 19	s tr	ne	DIC	)Ck	tre	e:	ye	<b>S</b>													
a	1	20					2	.	s th	ner	e e	enc	DUC	yh r	00	m?	)	уе	es	(ta	ke	as	m	nuc	;h	as			
a	2	16																р	OS	sib	le	to	th	e r	igł	nt)			
f	1								_									•	_						•				
a	3	32					Ba	ck	to	th	le v	ve	rsi	on	wł	<b>1er</b>	<b>e</b> 1	We		o h	av	es	spa	<b>ICE</b>	), Ī	fw	<b>e</b>		
a	4	12						СС	bun	nt a	all	th	e p	OS	sik	ble	fr	ee	sp	ac	et	ot	he	ri	gh	t			
f	3																												ß
r	2	60																											TELAND





Realloc 2 to 60 bytes  $\bullet$ 



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
222	22	112 F													

## 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





ullet

## After initialization, header holds the value of the whole free area, and designates it as (F)ree Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

- 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.











• Allocate 24 bytes

### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

## Is there enough space? **yes** use this block





Allocate 24 bytes  $\bullet$ 

### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

## 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







Allocate 20 bytes  $\bullet$ 

### Heap size: 256 bytes



78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC

## 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







Allocate 20 bytes ullet

### 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	fC
24 U	00	0000	000	24 U	11	111	11	184 F	next	t prev \0	/																			
		C	ooir	nter	r to	fre	e k		ck					ls t	her	re e	eno	ugl	n s	pac	ce?	ye	es							

- a 0 24 a 1 20
- 2 16 a
- f
- 3 a 32
- 12 4 a
- f 3

r 2 60

- - block.

# 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







Allocate 16 bytes ●

-																														
C	) (8	3 -	10 18	8 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	fO
	24 U	000	0000	<b>0</b>	1 11	<b>1 1</b>	111	184 F	next \0	prev \0																				
			ро	inte	er to	o fre	ee k		ck				•	ls t	her	e e	eno	ug	h s	pa	ce?	, <b>λ</b>	es							
	a	0	24													U	se	thi	s b	loc	k									
	a	1	20										•	Th	o r	noir	oto	ro	ho	oor	no	na	ort d	∽f t	·ho	പി		oto	d	
	a	2	16										-		ςŀ			13				μc				all		ลเธ	U S	she
	f	1												De	ca	USE	e w	'e (	JOL	זר	nee	ea	the	em	no	W!				
	a	3	32										•	We	e u	pd	ate	e th	ne h	nea	ip	oci	nte	er to	ор	oir	nt te	0 9	tre	Эе
	a	4	12											blo	ock															
	f	3																												
	r	2	60																											LELAN







Allocate 16 bytes  $\bullet$ 

### Heap size: 256 bytes

8 0	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80	88	90	98	a0	a8	b0	b8	c0	<b>c8</b>	d0	d8	e0	e8	fC
24 U <b>0</b>	0000	00	24 U	11	111	11	16 U	22	22	160 F	next \0	t prev \0	r																



- a 0 24
- 20 a a 2 16
- f
- 32 a 3
- 12 4 a
- f 3

r 2 60

- block.

## 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







Free 1 •

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	fC
24 U	00	000	000	24 U	11	11	111	16 U	22	222	160 F	next \0	pre√ ∖0	/																
		F	oir	nte	r tc	o fre	ee k	oloc	ck			•		Fre	е, а	ano	d a	dd	to	lir	ike	dl	ist							
a	. 0	2	4																											
a	. 1	2	0																											
a	. 2	1	6																											
f	1																													
a	. 3	3	2																											
a	. 4	1	2																											
f	3																													
r	2	6	0																											TELAND





• Free 1



#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	<b>b</b> 8	c0	c8	d0	d8	e0	e8	fC

## Free, and add to linked list

(remember to update all necessary doubly-linked list pointers!)







78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fO

- Check head of list (green) is there enough room? **no**
- Follow next pointer







78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC







#### Heap size: 256 bytes

78	80	88	90	98	a0	a8	b0	<b>b</b> 8	c0	c8	d0	d8	e0	e8	fC

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

### add here and update list







78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
333	120 F	next \0	prev 20												

- Check node in list is there enough







78	80	88	90	98	a0	a8	b0	b8	c0	c8	d0	d8	e0	e8	fC
333	120 F	next \0	prev 20												
													-		

- Check node (green) in list is there





Allocate 12 bytes  $\bullet$ 

### 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	fC
24 U	00	000	00	24 U	44	444	44	16 U	22	222	32 U	33	333	33:	333	120 F	nex \0	t prev \0	/											



- a 0 24
- 1 20 a
- a 2 16
- f 1
- 3 a 32
- 4 12 a
- f 3

r 2 60

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

### add here and update list





• Free 3

### Heap size: 256 bytes

0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80	88 nex	90 tlprev	98 /	a0	a8	b0	b8	<u>c0</u>	c8	d0	d8	e0	e8	fC
24 U	00	000	00	24 U	44	444	44	16 U	22	222	32 U	33	333	33	333	120 F	\0	\0												

•



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

Free 3, and coalesce





Free 3 •

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	fC
24 U	000	000	00	24 U	44	444	44	16 U	22	222	160 F	next \0	prev \0																	
		р	oin	ter	to	fre	ee k		ck			•		Fre Ma	e 3 ke	8, <b>a</b> Sur	nd e te	<b>CO</b> 0 U	<b>ale</b> Ipd	<b>esc</b> ate	<b>:e</b> : lin	ke	d lis	st						
a	0	24	1																1											
a	1	20	)																											
a	2	16	5																											
f	1																													
a	3	32	2																											
a	4	12	2																											
f	3																													
r	2	6(	)																											LELAND





Reallocate 2 to 60 bytes •

0	0	10	10	00	00	00	00	10	10	FO	50	<u> </u>	<u> </u>	70	70	00	00	00	00	-0	- 0	<b>b</b> 0	<b>b</b> 0	-0	- 0			- 0	- 0	<u>ــــــــــــــــــــــــــــــــــــ</u>
24	8	10	18	20	28	30	38	16	48	50	58 160	next	b8 prev	70	78	80	88	90	98		<u>að</u>		80	CU	<u>C8</u>	au	08	eu	68	
U	000	000	00	U	44	444	44	U	22	22	F																			
	-	р	OIr	nter	to	fre	e k		ck			•	• [	Enc	DUÇ	gh s	spa	Ce	aft	er?	y ye	es								
a	0	2	4										•		m do			ece	SSa	ary,	bu	t w	e c	ok	ha∖	/e				
a	1	2	0												JO	Our	Uμ	JUS		5.										
a	2	1	6																											
f	1																													
a	3	3	2																											
a	4	1	2																											
f	3																													
r	2	6	0																											TELAN




## Explicit Heap Allocation

Reallocate 2 to 60 bytes •

0	8	10 18	3 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	fC
24 U	000	0000	0 24	44	444	144	64 U	22	222	222	2222	222	222	222	222	112 F	next \0	i prev V0	/										
		ро	inte	r tc	fre	ee k		ck					Eno	SUC	gh s	spa	ICE	aft	er?	, <b>у</b>	es		40	hou					
a	0	24										-	to (	do	our		oda	ssa tes	ar y, S.	DU	IL V\		JO	1 av	/e				
a	1	20														-1-			-										
a	2	16																											
f	1																												
a	3	32																											
a	4	12																											
f	3																												
r	2	60																											ELAND

### Heap size: 256 bytes





## References and Advanced Reading

### References:

- •The textbook is the best reference for this material.
- <u>cs241/sp2014/lecture/06-HeapMemory\_sol.pdf</u>

Advanced Reading:

2946604/c-implementation-tactics-for-heap-allocators

•Here are more slides from a similar course: <u>https://courses.engr.illinois.edu/</u>

Implementation tactics for a heap allocator: <u>https://stackoverflow.com/questions/</u>







## Extra Slides

# Extra Slides



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.





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,

// 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.

```
double assignment avg, double midterm, double final)
```





### 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("Assignment Average: %g\n",si->assignment avg); printf("Midterm: %g\n",si->midterm); printf("Final: %g\n",si->final);

printf("Overall average: %g\n\n",overall avg);

```
printf("Labs attended: %d out of %d\n",si->labs attended,NUM LABS);
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);
```



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
```

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

update info(student data, "cgregg@stanford.edu", 7, 92.0, 83.4, 94.0);



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

// it's a big block of data, so we can add a student anywhere :0 // 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.







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

```
$ gdb struct ex
(gdb) break main
Breakpoint 1 at 0x400756: file struct ex.c, 1
(gdb) run
Starting program: /afs/.ir.stanford.edu/users
cgregg/tmp/lect15/struct ex
Breakpoint 1, main (argc=1, argv=0x7fffffffea
struct ex.c:45
(gdb) n
47
      void *student data = malloc(BIG BLOCK)
(gdb)
52
       update info(student data, "cgregg@stan
7, 92.0, 83.4, 94.0);
(gdb) x/10gx student data
0x602010: 0x00000000000000000
                               0x00000000000000
0x602020: 0x00000000000000000
                               0x0000000000000
0x602030:
                               0x00000000000000
          0x000000000000000000
0x602040:
          0x000000000000000000
                               0x0000000000000
                               0x0000000000000
0x602050:
          0x000000000000000000
(gdb) n
```

ine 45.
/c/g/
78) at
;
ford.edu",
0000
0000
0000

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 Os.





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

(gdb) n

(gdb) x/10gx student data 0x602010: 0x7340676765726763 0x602020: 0x000000000756465 0x602030: 0x00000000000000007 0x602040: 0x4054d9999999999a 0x602050: 0x000000000000000000 (gdb) n

0x2e64726f666e6174 0x000000000000000000 0x4057000000000000 0x4057800000000000 0x000000000000000000

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



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

(gdb) n

(gdb) x/10gx student data 0x602010: 0x7340676765726763 0x000000000756465 0x602020: 0x602030: 0x00000000000000007 0x602040: 0x4054d9999999999a 0x602050: 0x00000000000000000000 (gdb) n

0x2e64726f666e6174 0x000000000000000000 0x4057000000000000 0x4057800000000000 0x000000000000000000

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

The long values are read in reverese:  $0 \times 60210$  is the  $0 \times 63$  in the first block, which is a "c".



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

(gdb) n

(gdb) x/10gx student data 0x602010: 0x7340676765726763 0x602020: 0x000000000756465 0x602030: 0x00000000000000007 0x602040: 0x4054d9999999999a 0x602050: 0x000000000000000000 (gdb) n

0x2e64726f666e6174 0x000000000000000000 0x4057000000000000 0x4057800000000000 0x000000000000000000

### Address $0 \times 60211$ is the $0 \times 67$ in the first block, which is a "g".



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

(gdb) n

(gdb) x/10gx student data 0x602010: 0x7340676765726763 0x000000000756465 0x602020: 0x602030: 0x00000000000000007 0x602040: 0x4054d9999999999a 0x602050: (gdb) n

0x2e64726f666e6174 0x000000000000000000 0x4057000000000000 0x4057800000000000 0x000000000000000000

### Address $0 \times 60212$ is the $0 \times 72$ in the first block, which is a "r".



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

(gdb) n

(gdb) n

(gdb) x/10gx student data 0x602010: 0x7340676765726763 0x602020: 0x000000000756465 0x602030: 0x00000000000000007 0x602040: 0x4054d9999999999a 0x602050: 

0x2e64726f666e6174 0x000000000000000000 0x4057000000000000 0x4057800000000000 0x000000000000000000

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  $0 \times 60217$  is the  $0 \times 73$  in the first block, which is a "s" (in stanford.edu)



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

(gdb) n

(gdb) x/10	)gx student data	
0x602010:	0x7340676765726763	0x2e64726f666e6
0x602020:	0x000000000756465	0x00000000000000
0x602030:	$0 \times 000000000000000000007$	0x405700000000
0x602040:	0x4054d99999999999a	0x405780000000
0x602050:	$0 \ge 0 \ge$	$0 \times 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0$
(gdb) n		

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





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

(gdb) n

(gdb) x/10gx student data 0x602010: 0x7340676765726763 0x602020: 0x000000000756465 0x602030: 0x0000000000000007 0x602040: 0x4054d9999999999a 0x602050: 0x0000000000000000000 (gdb) n

0x2e64726f666e6174 0x000000000000000000 0x4057000000000000 0x4057800000000000 0x000000000000000000

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,  $0 \times 602030$  is the  $0 \times 07$  byte,  $0 \times 0602031$  is the  $0 \ge 000$  byte to the left of the  $0 \ge 07$ , etc.

