CS 107
Lecture 15: Managing the Heap Part II

Friday, March 2, 2018

malloc()
calloc() realloc()
free()
Today's Topics

- Programs from class: /afs/ir/class/cs107/samples/lect15
- Tail recursion elimination (question from sayat.me)
- Review of implicit heap allocator
- Explicit heap allocator
- More examples!
- Extra slides: Casting and structs
In lab, you saw an example of *tail recursion elimination*. Let's take a look at a recursive function to countdown numbers:

```c
void countdown(int n)
{
    if (n < 0) return;
    printf("%d\n",n);
    countdown(n-1);
}
```

Because this function's last instruction is a recursive call (at the *tail*), we can actually *remove the recursion altogether*. 

```
$ ./recursive_countdown 5
5
4
3
2
1
0
```
In C, we could use a `goto` statement (which is **strongly** discouraged!) to perform the same function:

```c
void countdown(int n)
{
    if (n < 0) return;
    printf("%d\n",n);
    countdown(n-1);
}
```

This uses a `label` called `beginning` to show where to jump at the end of the function, and this performs exactly the same as the original. We have **eliminated the tail call recursion**.
Tail Recursion Elimination

Let's look at what gcc does with the original example, with two different optimizations:

- **-Og**: original recursive function

```assembly
0400596 <+0>:  test  %edi,%edi
0400598 <+2>:  js    0x4005bc <countdown+38>
040059a <+4>:  push  %rbx
040059b <+5>:  mov   %edi,%ebx
040059d <+7>:  mov   $0x400694,%esi
040059f <+9>:  mov   %edi,%edx
04005a4 <+14>: mov    $0x1,%edi
04005a9 <+19>: mov    $0x0,%eax
04005ae <+24>: callq  0x400480 <__printf_chk@plt>
04005b3 <+29>: lea   -0x1(%rbx),%edi
04005b6 <+32>: callq  0x400596 <countdown>
04005bb <+37>: pop    %rbx
04005bc <+38>: repz retq
```

- **-O2**: no recursion!

```assembly
0400600 <+0>:  test  %edi,%edi
0400602 <+2>:  js    0x40062c <countdown+44>
0400604 <+4>:  push  %rbx
0400605 <+5>:  mov   %edi,%ebx
0400607 <+7>:  nopw  0x0(%rax,%rax,1)
0400610 <+16>: mov    %ebx,%edx
0400612 <+18>: xor    %eax,%eax
0400614 <+20>: mov    $0x4006b4,%esi
0400619 <+25>: mov    %0x1,%edi
040061e <+30>: sub    $0x1,%ebx
0400621 <+33>: callq  0x400480 <__printf_chk@plt>
0400626 <+38>: cmp    $0xffffffff,%ebx
0400629 <+41>: jne    0x400610 <countdown+16>
040062b <+43>: pop    %rbx
040062c <+44>: repz retq
```

With a high enough optimization, the compiler is smart enough to recognize that if the last C statement is a call to the function, it can perform tail recursion elimination, so it does.
Can we change the countdown function to a countup function, while keeping the same parameters?

```c
void countdown(int n)
{
    if (n < 0) return;
    printf("%d\n",n);
    countdown(n-1);
}

void countup(int n)
{
    if (n < 0) return;
    printf("%d\n",n);
    countup(n-1);
}
```
Can we change the countdown function to a count\textit{up} function, while keeping the same parameters?

```c
void countdown(int n)
{
    if (n < 0) return;
    printf("%d\n",n);
    countdown(n-1);
}
```

```c
void countup(int n)
{
    if (n < 0) return;
    printf("%d\n",n);
    countup(n-1);
}
```

Yup!
Can we change the countdown function to a countup function, while keeping the same parameters?

```c
void countup(int n) {
    if (n < 0) return;
    countup(n-1);
    printf("%d\n",n);
}
```

```bash
$ ./recursive_countup 5
0
1
2
3
4
5
```

Now, we can no longer perform tail recursion elimination, because the stack is doing the work of reversing for us! So, even on higher levels of optimization, the compiler still has to keep the recursion.
• 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);

- 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.
a = malloc(16);
b = malloc(8);

• We changed the header to reflect the fact that 8 bytes are going 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)

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0xffffe820</td>
</tr>
<tr>
<td>d</td>
<td>0xffffe818</td>
</tr>
<tr>
<td>c</td>
<td>0xffffe810</td>
</tr>
<tr>
<td>b</td>
<td>0xffffe808</td>
</tr>
<tr>
<td>a</td>
<td>0xffffe800</td>
</tr>
</tbody>
</table>

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);

• 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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0xfffffe820</td>
</tr>
<tr>
<td>d</td>
<td>0xfffffe818</td>
</tr>
<tr>
<td>c</td>
<td>0xfffffe810 0x130</td>
</tr>
<tr>
<td>b</td>
<td>0xfffffe808 0x120</td>
</tr>
<tr>
<td>a</td>
<td>0xfffffe800 0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>16 U</td>
</tr>
<tr>
<td>0x108</td>
<td>aaaaaaaa U</td>
</tr>
<tr>
<td>0x110</td>
<td>8 U</td>
</tr>
<tr>
<td>0x118</td>
<td>bbbb U</td>
</tr>
<tr>
<td>0x120</td>
<td>24 U</td>
</tr>
<tr>
<td>0x128</td>
<td>cccccccccc U</td>
</tr>
<tr>
<td>0x130</td>
<td>16 F</td>
</tr>
<tr>
<td>0x138</td>
<td></td>
</tr>
<tr>
<td>0x140</td>
<td></td>
</tr>
<tr>
<td>0x148</td>
<td></td>
</tr>
<tr>
<td>0x150</td>
<td></td>
</tr>
<tr>
<td>0x158</td>
<td></td>
</tr>
</tbody>
</table>
Implicit Free List (review from Monday)

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0xffffe820</td>
</tr>
<tr>
<td>d</td>
<td>0xffffe818</td>
</tr>
<tr>
<td>c</td>
<td>0xffffe810 0x130</td>
</tr>
<tr>
<td>b</td>
<td>0xffffe808 0x120</td>
</tr>
<tr>
<td>a</td>
<td>0xffffe800 0x108</td>
</tr>
</tbody>
</table>

The diagram now reflects the free.
Implicit Free List (review from Monday)

```plaintext
a = malloc(16);
b = malloc(8);
c = malloc(24);
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.
Implicit Free List (review from Monday)

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

• 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!)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0xffffe820</td>
</tr>
<tr>
<td>d</td>
<td>0xffffe818</td>
</tr>
<tr>
<td>c</td>
<td>0xffffe810</td>
</tr>
<tr>
<td>b</td>
<td>0xffffe808</td>
</tr>
<tr>
<td>a</td>
<td>0xffffe800</td>
</tr>
</tbody>
</table>

96 bytes
Implicit Free List (review from Monday)

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0xfffffe820</td>
</tr>
<tr>
<td>d</td>
<td>0xfffffe818</td>
</tr>
<tr>
<td>c</td>
<td>0xfffffe810 0x130</td>
</tr>
<tr>
<td>b</td>
<td>0xfffffe808 0x120</td>
</tr>
<tr>
<td>a</td>
<td>0xfffffe800 0x108</td>
</tr>
</tbody>
</table>

- 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.
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.
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.
Coalescing forwards is straightforward:

![Diagram showing memory layout]

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.
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. 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 freed 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

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x108</th>
<th>0x110</th>
<th>0x118</th>
<th>0x120</th>
<th>0x128</th>
<th>0x130</th>
<th>0x138</th>
<th>0x140</th>
<th>0x148</th>
<th>0x150</th>
<th>0x158</th>
<th>0x160</th>
<th>0x168</th>
<th>0x170</th>
<th>0x178</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>bbbb</td>
<td>8</td>
<td>24</td>
<td>F</td>
<td>24</td>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

160 bytes

Freed block
More on Coalescing: coalescing backwards

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>16 F</td>
</tr>
<tr>
<td>0x108</td>
<td>16 F</td>
</tr>
<tr>
<td>0x110</td>
<td>8 F</td>
</tr>
<tr>
<td>0x118</td>
<td>bbb U</td>
</tr>
<tr>
<td>0x120</td>
<td>8 U</td>
</tr>
<tr>
<td>0x128</td>
<td>24 F</td>
</tr>
<tr>
<td>0x130</td>
<td>8 U</td>
</tr>
<tr>
<td>0x138</td>
<td>24 F</td>
</tr>
<tr>
<td>0x140</td>
<td></td>
</tr>
<tr>
<td>0x148</td>
<td></td>
</tr>
<tr>
<td>0x150</td>
<td></td>
</tr>
<tr>
<td>0x158</td>
<td></td>
</tr>
<tr>
<td>0x160</td>
<td>24 F</td>
</tr>
<tr>
<td>0x168</td>
<td>8 F</td>
</tr>
<tr>
<td>0x170</td>
<td>8 F</td>
</tr>
<tr>
<td>0x178</td>
<td>8 F</td>
</tr>
</tbody>
</table>

160 bytes

Freed block header
More on Coalescing: coalescing backwards

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x108</th>
<th>0x110</th>
<th>0x118</th>
<th>0x120</th>
<th>0x128</th>
<th>0x130</th>
<th>0x138</th>
<th>0x140</th>
<th>0x148</th>
<th>0x150</th>
<th>0x158</th>
<th>0x160</th>
<th>0x168</th>
<th>0x170</th>
<th>0x178</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>16 F</td>
<td>16 F</td>
<td>8</td>
<td>bbbb</td>
<td>8</td>
<td>F</td>
<td>24</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td>F</td>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

Footer for previous block (also free)
More on Coalescing: coalescing backwards

Entire free area

After coalescing backwards
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.
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!
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

```c
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.
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?

First Free Block

160 bytes
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 7, 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
```
**Implicit 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
Implicit Heap Allocation

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

Heap size: 256 bytes

- Addresses are in hex
- Byte allocations are in decimal
Implicit Heap Allocation

- Allocation of 24 bytes

Heap size: 256 bytes

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

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

Heap size: 256 bytes

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

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

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

| 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  | 00000000 | 24 | U  | 11111111 | 184 | F  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

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

| 0 | 8 | 10 | 18 | 20 | 28 | 30 | 38 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 | 80 | 88 | 90 | a0 | a8 | b0 | b8 | c0 | c8 | d0 | d8 | e0 | e8 | f0 | f8 |
|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 24 |   |   | 24 |   | 184 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0000000 | 1111111 | F |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

pointer to heap

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

- Allocation of 16 bytes

Heap size: 256 bytes

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

**use this block**
### Implicit Heap Allocation

- Allocation of 16 bytes

**Heap size: 256 bytes**

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |  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 |
|   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|   | 24 |    |    |    |    | 24 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    | 16 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

- **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 for the request? **yes**

**use this block**
Implict Heap Allocation

- Free 1

Heap size: 256 bytes

1. Mark as free
Implicit Heap Allocation

- Free 1

Heap size: 256 bytes

1. Mark as free
Implicit Heap Allocation

- Allocation of 32 bytes

Heap size: 256 bytes

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

- Allocation of 32 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 |
|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U | 00000000 | F | 24 | 24 | U | 16 | 160 | 2222 | F | 16 |

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)
Implicit Heap Allocation

- Allocation of 32 bytes

Heap size: 256 bytes

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

- Allocation of 32 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 | 00000000 | 24 | F | 16 | U | 2222 | 160 | F | | | | | | | | | | | | | | | | | | | | | | | | |

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

**use this block**

- 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`
Implicit Heap Allocation

• Allocation of 32 bytes

Heap size: 256 bytes

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

**use this block**
Implicit Heap Allocation

- Allocation of 12 bytes

<table>
<thead>
<tr>
<th>Heap size: 256 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
</tr>
<tr>
<td>24 U 0000000 24 F 16 U 2222 32 U 3333333333 120 F</td>
</tr>
</tbody>
</table>

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

- Allocation of 12 bytes

<table>
<thead>
<tr>
<th>Heap size: 256 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
</tr>
<tr>
<td>24 U 0000000 24 F</td>
</tr>
<tr>
<td>16 U 2222 32 U 3333333333 120 F</td>
</tr>
</tbody>
</table>

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

**use this block**
Implicit Heap Allocation

- Allocation of 12 bytes

Heap size: 256 bytes

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

Heap size: 256 bytes

1. Set to free
Implicit Heap Allocation

- **Free 3**

<table>
<thead>
<tr>
<th>Heap size: 256 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
</tr>
<tr>
<td>24 U 0000000 24 U 4444444 16 U 2222 32 F 120 F</td>
</tr>
</tbody>
</table>

- **pointer to heap**

  1. **Set to free**

     No coalescing for implicit!
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 24 | 0  | 24 | 0  | 24 | 0  | 44 | 44 | 44 | 44 | 32 | 16 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| 24 | U  | 00 | 00 | 00 | 00 | 24 | U  | 00 | 00 | 00 | 00 | 24 | U  | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

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. 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 the whole list for a spot with enough space (see the next slide)
Implict Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes

1. Is the block free? **no**
   - Check next block

Assume, for a moment, that there had not been space. We would have started searching
Implicit 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 | 00000000 | 24 | U | 44444444 | 16 | U | 2222 | 32 | F |
| 16 | U | 5555 | 96 | F |

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

- Check next block

Assume, for a moment, that there had not been space. We would have started searching
Realloc 2 to 60 bytes

Heap size: 256 bytes

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 | 44444444 | 16 | U | 2222 | 32 | F | 16 | U | 5555 | 96 | F | 44444444 | 16 | U | 2222 | 32 | F | 16 | U | 5555 | 96 | F |

pointer to heap

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

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

Check next block
Implicit Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes

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

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

Assume, for a moment, that there had not been space. We would have started searching.
Implicit 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 | 16 | U | 5555 | 96 | F | 16 | U | 5555 | 96 | F | 16 | U | 5555 |

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 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

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
<th>heap state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
<td>U</td>
</tr>
<tr>
<td>8</td>
<td>4444444</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>32 F</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>16 U</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>64 F</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>222222222222</td>
<td>F</td>
</tr>
</tbody>
</table>

Heap size: 256 bytes

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

- Realoc 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 | 120| F | 2222 | 32| F | 120| F | 2222 | 32| F | 120| F | 2222 | 32| F | 120| F | 2222 | 32| F | 120| F |

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
Implicit Heap Allocation

- Realloc 2 to 60 bytes

Heap size: 256 bytes

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

- 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

- Is there enough space? **yes** use this block
Explicit Heap Allocation

- Allocate 24 bytes

Heap size: 256 bytes

- 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
  - 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
  - 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**

| 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  | 00 | 00 | 00 | 00 | 24 | U  | 11 | 11 | 11 | 11 | 18 | 4 F | next | prev | \0  | \0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- 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

|   | 0  | 8 | 10 | 18 | 20 | 28 | 30 | 38 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 | 80 | 88 | 90 | a0 | a8 | b0 | b8 | c0 | c8 | d0 | d8 | e0 | e8 | f0 | f8 |
|---|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 24 | U | 0000000 | 24 | U | 1111111 | 16 | U | 2222 | 160 | F | next | \0 | prev | \0 | | | | | | | | | | | | | | | | | | | | | |

- 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

| 0 | 8 | 10 | 18 | 20 | 28 | 30 | 38 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 | 80 | 88 | 90 | a0 | a8 | b0 | b8 | c0 | c8 | d0 | d8 | e0 | e8 | f0 | f8 |
| 24 | U | 0000000 | 24 | U | 1111111 | 16 | U | 2222 | 160 | F |

pointer to free block

- Free 1

| 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

• 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

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

**Follow next pointer**
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
```
Explicit Heap Allocation

- Allocate 32 bytes

Heap size: 256 bytes

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

  **add here and update list**
Explicit Heap Allocation

- Allocate 32 bytes

  Heap size: 256 bytes

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

  **add here and update list**
Explicit Heap Allocation

- Allocate 12 bytes

Heap size: 256 bytes

- Check node (green) in list — is there enough room? **yes**
  
  add here and update list
Explicit Heap Allocation

- Allocate 12 bytes

Heap size: 256 bytes

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

```plaintext
24 U 0000000
24 U 4444444
16 U 2222
32 U 3333333333
120 F

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
```
Explicit Heap Allocation

- Free 3

Heap size: 256 bytes

• 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

• Free 3, **and coalesce**
• Make sure to update linked list
Explicit Heap Allocation

- Reallocate 2 to 60 bytes

Heap size: 256 bytes

```plaintext
<table>
<thead>
<tr>
<th>a0</th>
<th>a1</th>
<th>a2</th>
<th>f1</th>
<th>a3</th>
<th>a4</th>
<th>f3</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>32</td>
<td>12</td>
<td>3</td>
<td>60</td>
</tr>
</tbody>
</table>
```

- 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

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

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`):

```c
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:

```c
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.
We can do the same thing for a print function:

```c
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:

```c
#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)
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:

```plaintext
$ 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.
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.
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: 0x405d99999999999a 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 reverse: `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: 0x4054d99999999999a 0x4057800000000000
0x602050: 0x0000000000000000 0x0000000000000000
(gdb) n
```

Address \texttt{0x60211} is the \texttt{0x67} in the first block, which is a "g".

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

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

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

```
(gdb) n
(gdb) x/10gx student_data
0x602010: 0x73406767657263 0x2e64726f666e6174
0x602020: 0x0000000000756465 0x0000000000000000
0x602030: 0x0000000000000007 0x4057000000000000
0x602040: 0x4054d9999999999a 0x4057800000000000
0x602050: 0x0000000000000000 0x0000000000000000
(gdb) n
```

Address \texttt{0x60217} is the \texttt{0x73} in the first block, which is a "s" (in stanford.edu)

After we update the student, the info is located in \texttt{student_data}, but we have to be careful reading it -- it presumes little-endian format, but prints in normal format...
Let's look at a gdb trace of the program:

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: 0x40549999999999a 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.
Let's look at a gdb trace of the program:

(gdb) n
(gdb) x/10gx student_data

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.

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...