Project 2 Details

> ***No one writes a program in a single try.*** Do **NOT** try to write the entire allocator without compiling, and then compile it and get a million errors and fix them and run it and it crashes and aaaaahhhh oh my godddddd

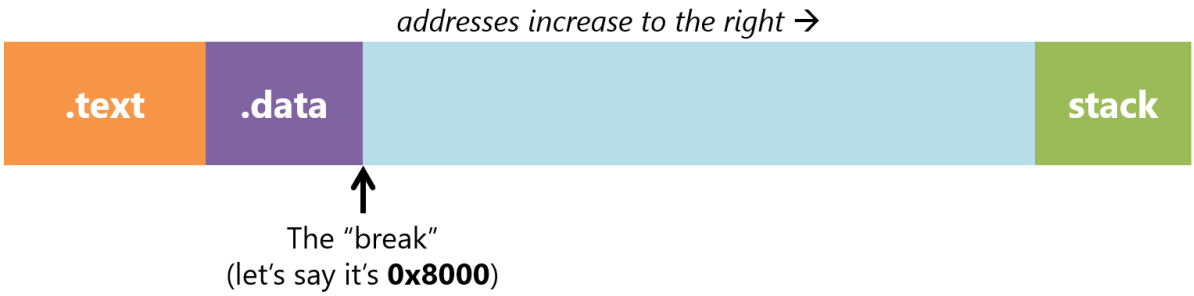
Little functions are your friends

Split your code up into small functions with descriptive names. Seriously. It sounds silly but it makes things so much easier to read and fix.

In particular, **put any complex pointer calculations** or **annoying linked list management** into separate functions. Doubly linked lists are tricky to manage. Don't copy and paste the code to manage them over and over. Write it once, and call it anywhere you need it.

The Break

When any process first starts running, its (virtual!) address space will look something like this:



The "break" represents the "frontier" between the memory that your process *does* have access to, and the empty, unallocated space that it *doesn't* have access to. (Accessing addresses in the large middle region would result in a segfault.)

We can ask the OS for more space dynamically by **moving the break up.** We can do this with the **sbrk** function.

> You can't compile this project on your Mac.

There is no `sbrk()` or `brk()` on macOS.

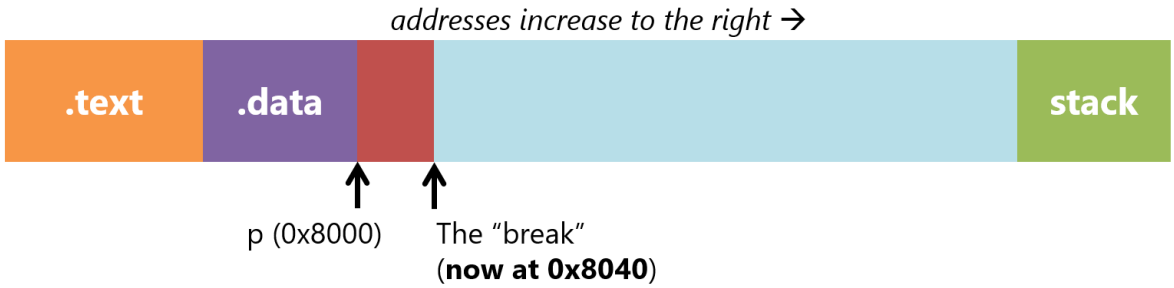
Please do not ask questions about why you get linker errors or deprecation warnings.

This is because you're trying to compile it on your Mac.

You can still write the code on your Mac, but you have to upload and test on thoth.

When you call `sbrk(n)`, it will move the break up by `n` bytes and return the *old* location of the break. This new memory space is where we'll keep our heap!

```
void* p = sbrk(0x40); // 64 bytes
```

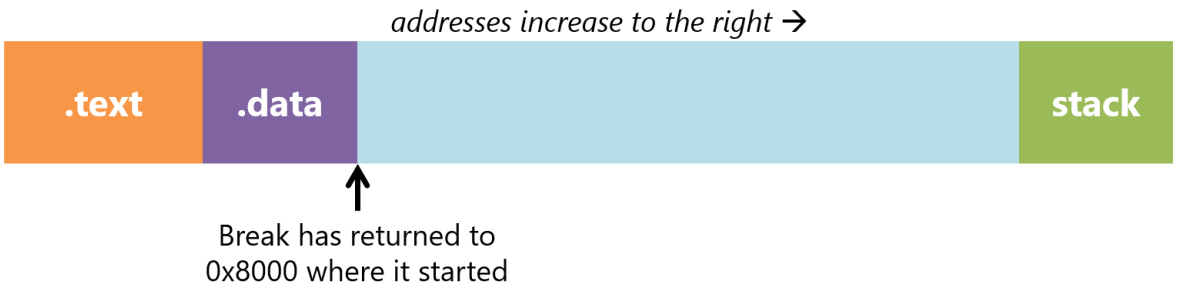


Now, `p` points to a block of 64 bytes.

But moving the break around is more like moving the stack pointer: **you can only deallocate this heap memory by moving the break back down.** We call this **"contracting" the heap.**

There are two ways to contract the heap: either use the **brk** system call to *set* the break to a certain address, or use **sbrk** with a negative offset to move it backwards:

```
brk(p);
// or
sbrk(-0x40);
```



Heap Blocks

Your heap will be divided into **blocks.** The blocks will be in a **doubly-linked list.** Each block can be either **free** (ready for reuse) or **used** (someone `malloc`'ed it and is still using it).

A block has two parts: the **header** and the **data.** They are right next to each other in memory. Each **header** contains the following information:

- The size **of the data part**
  - NOT the size of the data + header!
- Whether this block is in use (allocated) or not (free)
- The pointer to the next block
- The pointer to the previous block

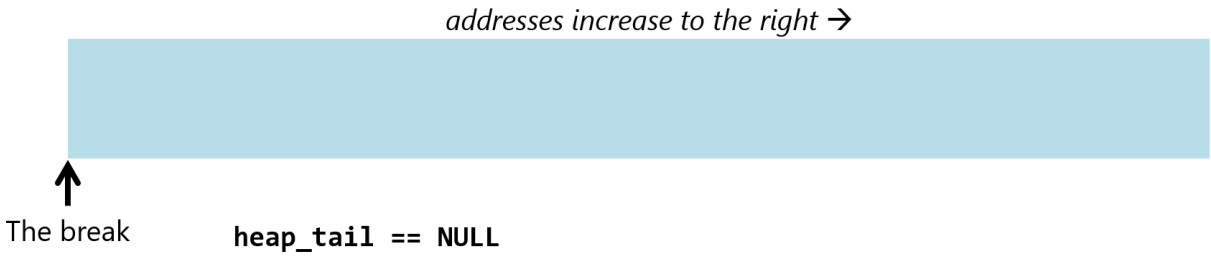
What do you think you should use to represent this information? 😘

Now, think about this: if the user asks for 64 bytes, how big will the block be, *including the header?*

The doubly-linked list

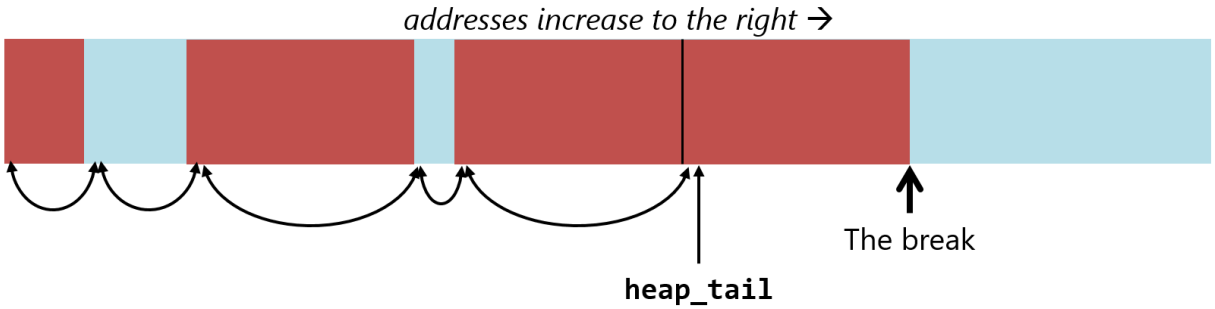
> A doubly-linked list has pointers to the head *and* the tail. These diagrams only show the tail, but you need to keep track of the head too. And since you're using next-fit, you need to keep track of the last-allocated block, too!
>
> You can use global variables for the head, tail, and last-allocated block. This is one of the few places where a global variable makes sense, since there's only one heap in your program.

When the program first starts and the heap is empty, your linked list will also be empty:



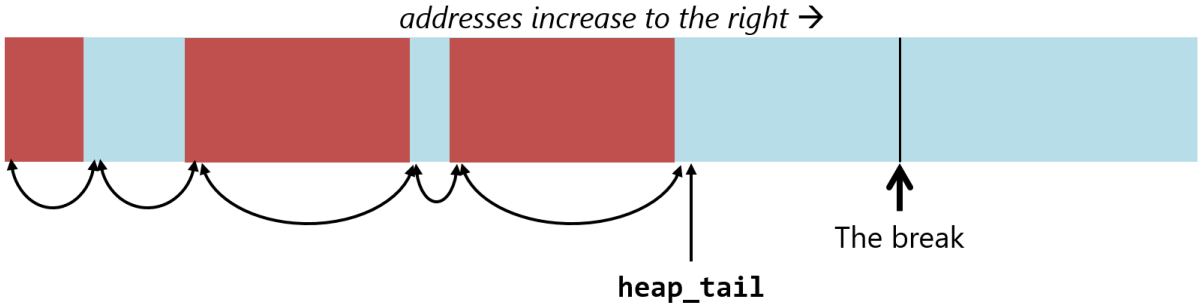
Here is a visual example of how the physical linked list might look with five blocks. Red blocks are allocated, and blue blocks are free. The arrows are the

links between the blocks:

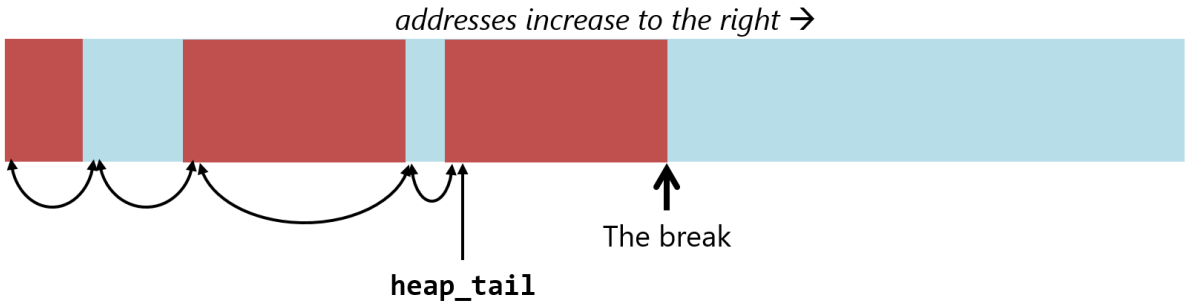


To know when you can **contract the heap,** when you free a block, see if it's the last block on the heap (the tail of the linked list).

Suppose the last block were freed, leaving us with this situation:



Now we need to change where the heap tail is, and move the break down with either `brk()` or `sbrk()`:



GO MAKE A DANG ALLOCATOR

See if you can get the stuff discussed so far working. After each step, test it with your driver and `bigdriver`. As you add more features, more tests in `bigdriver` will pass. Just keep uncommenting tests in `bigdriver`'s `main`.

> The failure of an earlier test in `bigdriver` can cause later ones to fail, not because your allocator is broken, but because there are junk blocks left on the heap from the failed test. Only focus on fixing the problems with the first failed test.

1. Make the dumbest allocator possible by just moving the break up when `malloc`ing and moving it back down when `free`ing.
2. Make it smarter by putting headers on those blocks and linking them together into a list.
    - Make functions to print out the linked list.
    - Feel free to use the coloring macros from `bigdriver.c`.
3. Make it even smarter by allowing the user to free blocks other than the last one.
    - But when they *do* free the last one, remember to move the break down.
4. Then work on *reusing free blocks*.
    - You are implementing **next-fit.** That means you keep track of the most-recently-allocated block and start searching for free blocks after it.
        - If you get to the end of the heap, start back at the beginning.
        - If you get all the way back around to the most-recently-allocated block, then you need to move the break up like before.
    - Even if you reuse a block, that becomes "most-recently-allocated."
    - When you free a block, you must check if it's the most-recently-allocated block before getting rid of it. If so, reset the most-recently-allocated block to the heap head (if there is one).

Hint: C `for` loops fit with linked lists very well...

```
for(current = head; current != NULL; current = current->next)
{
        //...
}
```

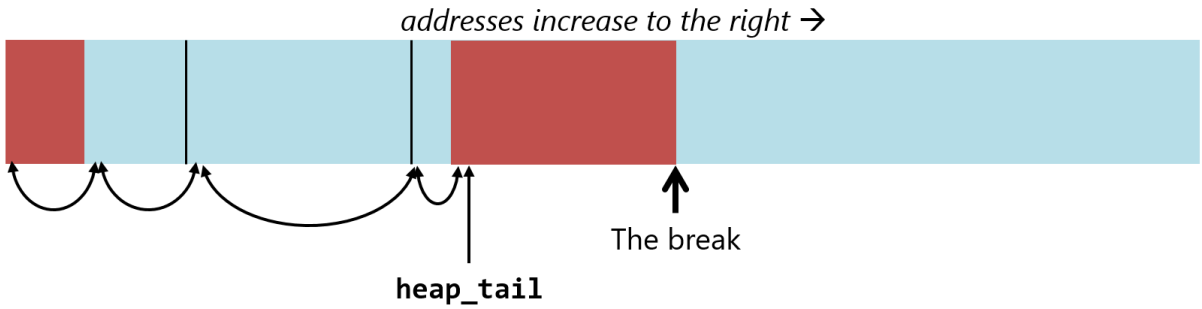
Please, please try to get the above stuff working **before** trying to do block splitting and coalescing. It's better to get all that stuff working and get an 80% than to try to get *everything* working and get a 60%.

Coalescing blocks (in `my_free`)

You don't *technically* have to implement block splitting/coalescing to make a functional allocator. But they are really important for getting anything close to

decent memory usage.

When the user frees a block, you might end up with multiple free blocks in a row, like these three here:

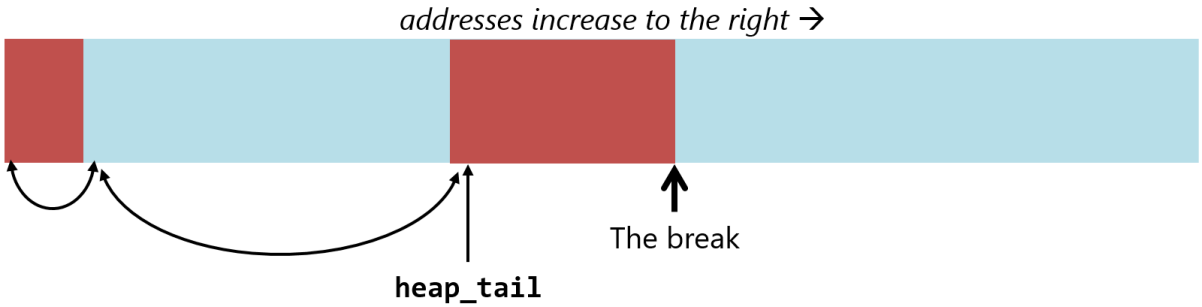


This is a waste, because what if the user wanted an allocation the size of those three put together? Then our algorithm would skip them and move the break.

So there is a simple rule to follow: whenever you free a block, **look at the previous and next blocks, and if either one or both are free, combine the free blocks into one larger free block.**

> *Note: you don't have to look further than one block away! If you follow this rule on every deallocation, it will be impossible to have e.g. 4 or 5 free blocks in a row.*

Once coalesced, the three free blocks in the above diagram would become a single large block, like so:



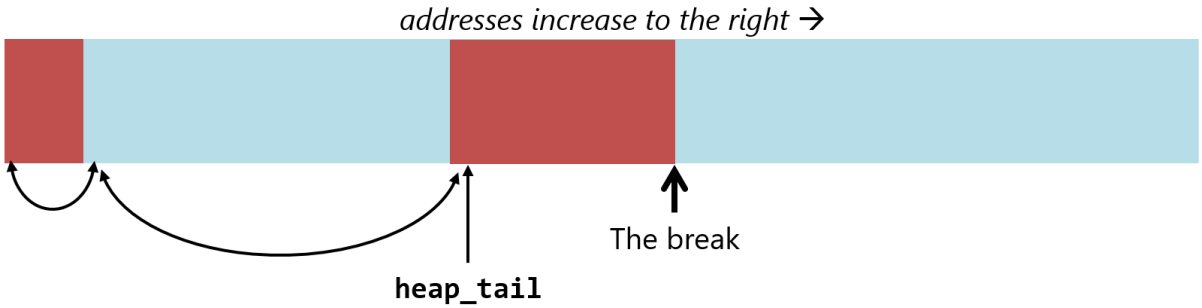
Notes

- We talked about this in lecture 09.
- Think about how the linked list links will have to change.
  - Doubly linked lists have so many special cases… it's annoying. MAKE FUNCTIONS FOR IT.
- You might free the **first or last block** on the heap, so there might not be a previous/next physical block at all.
- You might have **zero free neighbors, one free neighbor, or two free neighbors.** You have to handle all possible cases.

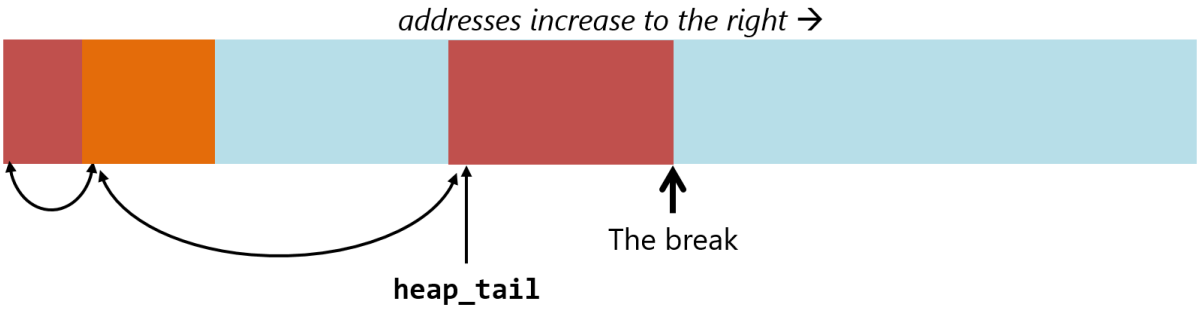
    - But remember, like I said in class, that "two free neighbors" can be handled as "one free neighbor," twice.
- The coalescing should be done **before** deciding whether or not to contract the heap.

Splitting blocks (in `my_malloc`)

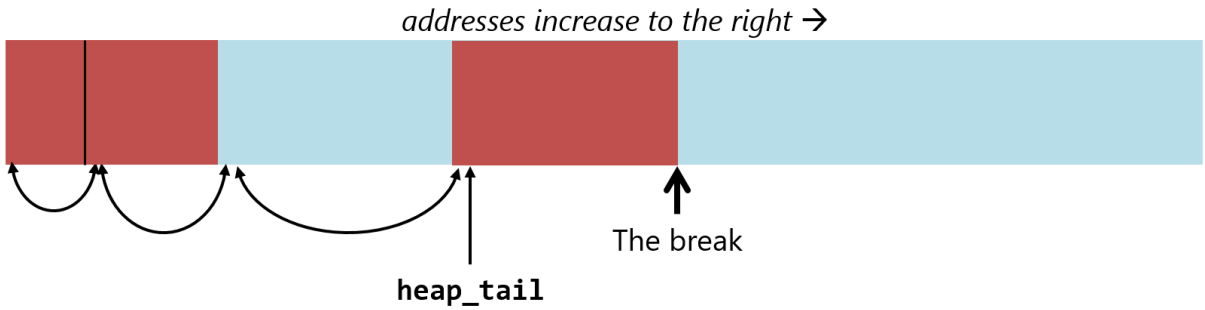
Next-fit just finds the next block that is big enough for the allocation, and sometimes that block is way too big. Let's say your heap looks like this:



but the user only wants a little block:



Now we have to split the original large block into two smaller blocks: one **big enough to hold the requested block,** and one **smaller empty block,** like so:



Undersized blocks

You can't split every free block. Each block has to have space for the header and the data, and the data has to be a minimum size of **16 bytes.** If splitting the block would create one of these undersized blocks:

- Don't split
- But also **don't change the size of the block in the header.**
    - Remember, you're allowed to give the user *more* space than they asked for.
    - You'll take the internal fragmentation hit, but it's only a couple dozen bytes... right? ;)

Notes

- We *also* talked about this in lecture 09.
- The steps are kind of the reverse of coalescing: instead of summing sizes and removing links, you're subtracting sizes and adding links.
- The `PTR_ADD_BYTES` macro in `bigdriver.c` would be useful for this.
    - It adds a certain number of bytes to a pointer to give a new pointer.
    - This avoids the undesired offset scaling.

Extra credit???

Using a linked list to store the blocks is linear time in the worst case. You can do better by **using a more advanced data structure to store the free blocks.** It's up to you to choose what data structure to use - just as long as it results in **better than linear time in the worst case** for allocation.

> *Implementing complex data structures in C is..... "fun" :^)*