```
uint32_t nums[] = {77, 99, 45, 11, 13};
uint32_t* ptr = nums + 2;

printf("%p\n", nums);            // prints 0x1000
printf("%p %d", ptr, ptr[0]);    // what does this print?
```

ptr = nums + 4
how would answer change?
0x1010 13

0x1000 77	0x1008 45	0x1002 45	0x1000 45
would be true for ptr = nums	reflects	would be true for char*	this doesn't reflect that ptr is a different address than nums

pointer arithmetic rules. ptr has type uint32_t, which is 4 bytes so adding 2 means add 2 uint32_t-sized chunks

$$p + n = p + (\text{sizeof}(T) * n)$$

T*

byte address result

```c
// DOUBLE the capacity of the list and copy over the old contents into the new
// space.
void expandCapacity(List* l) {
        uint32_t new_capacity = l->capacity * 2;
        Str* new_contents = calloc(sizeof(Str), new_capacity);
        Str* old_contents = l->contents;
        l->contents = new_contents;
        for(int i = 0; i < l->size; i += 1) {
                l->contents[i] = old_contents[i];
        }
        free(old_contents); // Important to avoid a memory leak
        l->capacity = new_capacity;
}
```

How to use realloc() to simplify expandCapacity?

```c
size_t size = new_cap * sizeof(Str);
l->contents = realloc(l->contents, size );
```

void* realloc(void* ptr, $\boxed{\text{size\_t}}$ size)

size_t → essentially uint64_t

could we find out by just trying sizeof(size_t)?

Try to expand memory at ptr to size.

If not possible, allocate new memory and copy the contents of ptr there, and return new address. And free ptr

What happens to the old address of l->contents if realloc returns a new address? How would you free the memory?

what does void* mean?

void* is a pointer to data of unknown type

sizeof(void) == 1 sizeof(void*) == 8

Implementing malloc() and free()

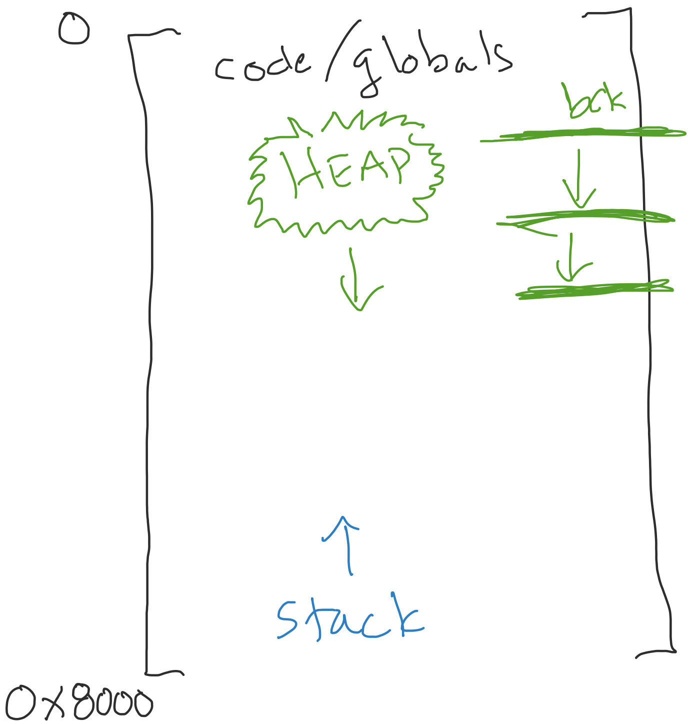
#include <stdlib.h>

```
void* malloc(size_t size) { ... }

void free(void* ptr) { ... }
```

What's involved?

What do we need?



We need the start of the heap!

```
void *sbrk(intptr_t increment);
```

sbrk() increments the program's data space by increment bytes. Calling **sbrk**() with an increment of 0 can be used to find the current location of the program break.

"program's data space" is the heap

```
void* malloc(size_t size) {
    return sbrk(size);
}
void free(void* ptr) { return; }
```

How does the return; statement in free prevent that address from being used again?

malloc() never repeats an address

- Good throughput (mem ops / time)
- Utilization BAD

If the break is moved, how do you keep track of where one memory block is supposed to end in case one piece of data tries to write into the next block?

Then what function does stop you from overwriting blocks in the heap?

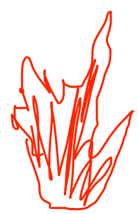
Nothing in C can prevent this

malloc doesn't do this – it does not prevent users of malloc from making mistakes

Consider your web browser
(Safari Chrome Edge Ffox)

These programs run for weeks

is this why chrome is so slow?

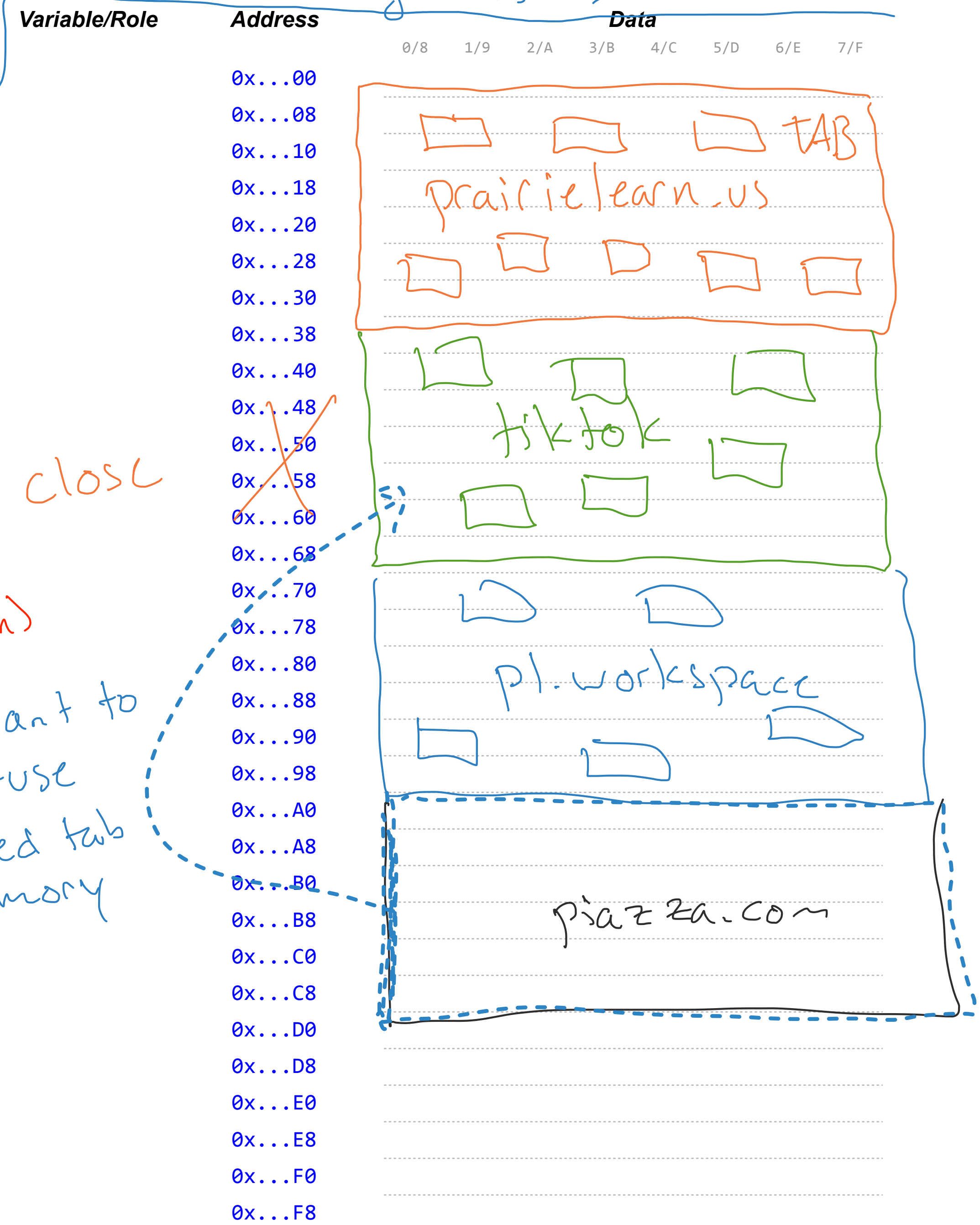


(example of a sick burn)

The memory leak team on Chrome is a big team

close

want to re-use closed tab memory



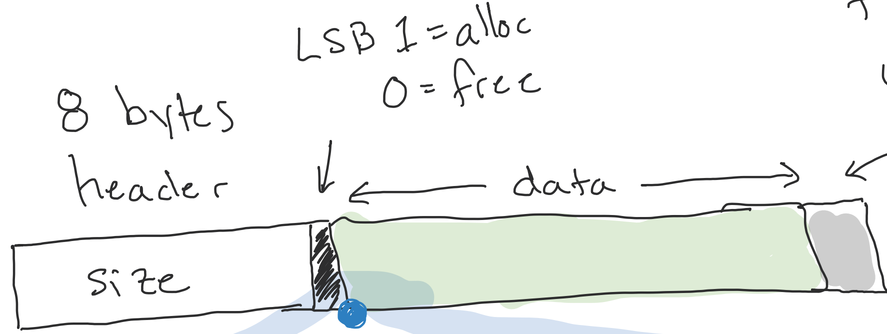
void* malloc(size_t size)

void free(void* ptr)

2 necessary kinds of information:

- how long each "block" is
- whether each block is alloc'ed or free

Store this directly in the heap next to the allocated/free blocks for user



padding up to next mult of 8

- perf + simplicity of addresses
- ensures the last 3 bits of the byte count are 000

return from malloc

or expect as argument to free()

Why?

By analogy – multiple of 1000 why are the last 3 digits always 000

0b 1100 0001

header value of 0xC1

what kind/size of block?

its allocated and it is size 12

i think it's allocated

and 192 bytes?

hmm, the size of block would be 12 * 16 + 1 = 193?

not part of size!

h & 1 (free/busy)

h & (~1) (count)

h & 0xFFFF---FE

free, size = 8+(13*16) bytes

$\boxed{0xD8}$

size/kind?

The LSB is free/busy metadata. The count/size always uses a 0 for LSB

a = malloc(13)

b = malloc(19)

c = malloc(8)

free(b)

If the next multiple of 8 is 24, then why 25?

need to set 1/0 for busy/free to 1

Isn't it supposed to be ==

For conditional

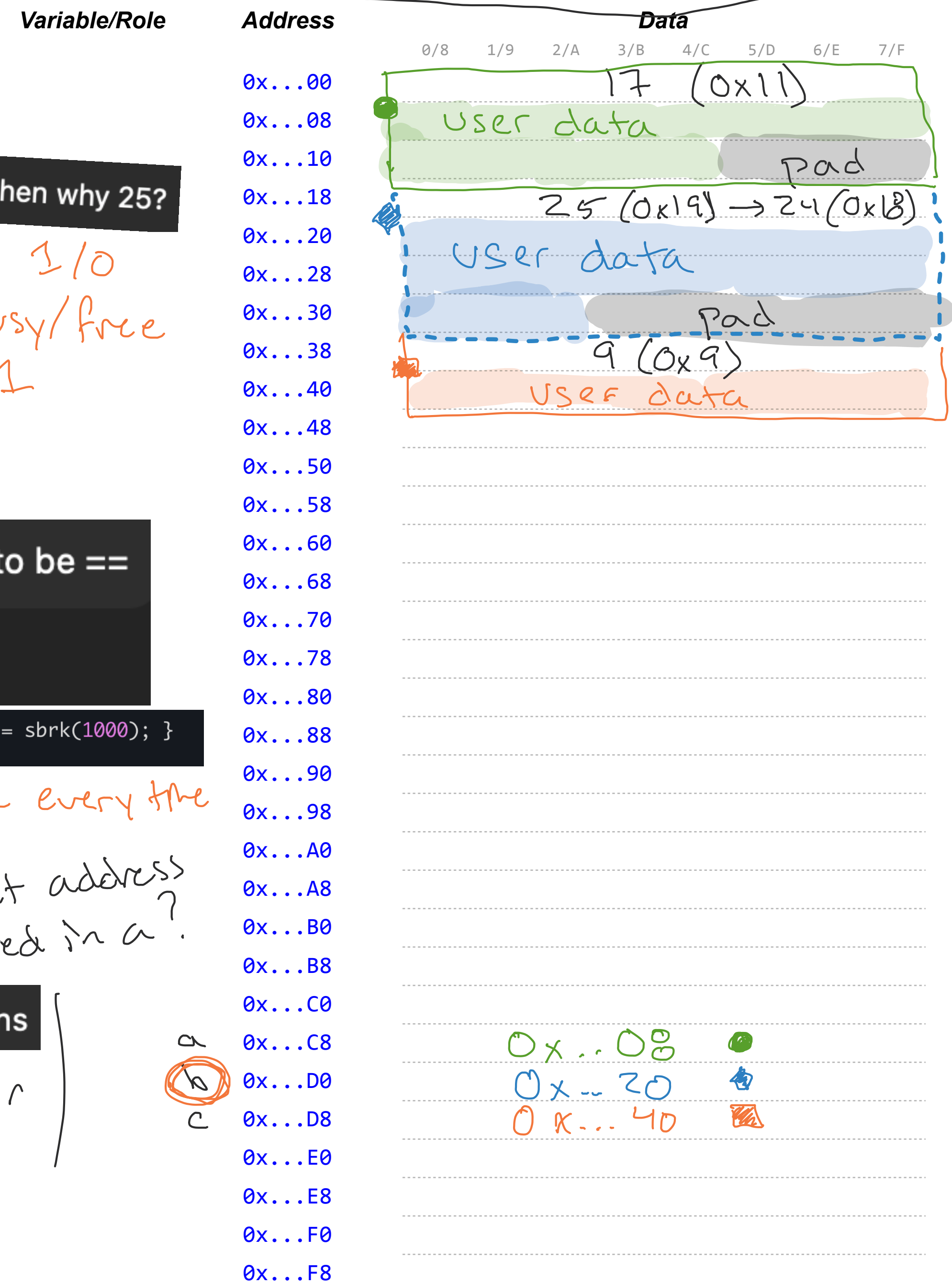
```
if(HEAP_START = NULL) { HEAP_START = sbrk(1000); }
```

assigned to NULL every time

what address stored in a?

The address where the user data part of a begins

main



```c
int main() {

        char* a = mymalloc(13);
        printf("HEAP_START vs a: %p %p\n", HEAP_START, a);
        printf("first heap loc: 0x%lx\n", *(uint64_t*)HEAP_START);
        char* b = mymalloc(19);
        printf("HEAP_START vs b: %p %p\n", HEAP_START, b);
        char* c = mymalloc(8);
        printf("HEAP_START vs c: %p %p\n", HEAP_START, c);
        uint64_t* b_header = (uint64_t*)(b - 8);
        printf("b's header: %ld 0x%lx\n", b_header[0], b_header[0]);
        myfree(b);
        printf("b's header: %ld 0x%lx\n", b_header[0], b_header[0]);

        char* d = mymalloc(20); // should use b's space!
                                // But it just goes forward!
                                // What to do?

        print_heap();

```

```
0 (free)
0 (free)
0 (free)
0 (free)
$ gcc -g mymalloc.c -o mymal
$ ./mymalloc
Rounded block size: 13 16
HEAP_START vs a: 0x5582b87fa
first heap loc: 0x11
Rounded block size: 19 24
HEAP_START vs b: 0x5582b87fa
Rounded block size: 8 8
HEAP_START vs c: 0x5582b87fa
b's header: 25 0x19
b's header: 24 0x18
Rounded block size: 20 24
16 (allocated)
24 (free)
8 (allocated)
24 (allocated)
```

go through blocks in heap to find unallocated that is large enough

Variable/Role	Address	Data							
		0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
	0x...40								
	0x...48								
	0x...50								
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								
	0x...88								
	0x...90								
	0x...98								
	0x...A0								
	0x...A8								
	0x...B0								
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								

Variable/Role	Address	Data							
		0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
	0x...40								
	0x...48								
	0x...50								
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								
	0x...88								
	0x...90								
	0x...98								
	0x...A0								
	0x...A8								
	0x...B0								
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								

Variable/Role	Address	Data 0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...88								
	0x...90								
	0x...98								
	0x...A0								
	0x...A8								
	0x...B0								
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
	0x...40								
	0x...48								
	0x...50								
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								

Variable/Role	Address	Data							
		0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...88								
	0x...90								
	0x...98								
	0x...A0								
	0x...A8								
	0x...B0								
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
	0x...40								
	0x...48								
	0x...50								
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								