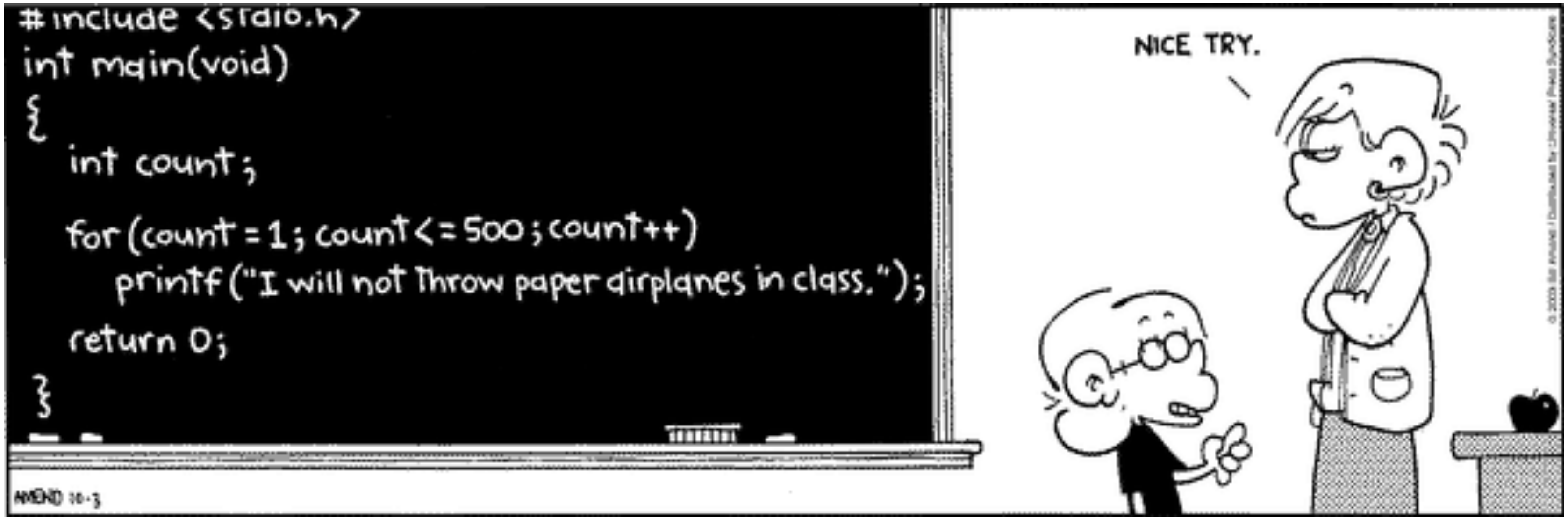
Admin

- Assign 2 grading results out soon, revise and resubmit on open issues
- printf perseverance and pride!!



Today: Thanks for the memory!

Runtime stack, stack frame layout

Linker memory map, address space layout

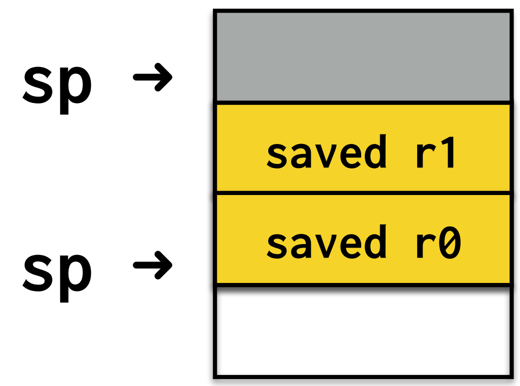
Loading, how an executable file becomes a running program

Heap allocation, malloc and free

Stack operations

```
// push: add/store val to stack
// *--sp = val
// decrement sp before store

push {r0, r1}


// pop: remove/load val from stack
// val = *sp++
// increment sp after load

pop {r0, r1}
```

“Full Descending” stack



APCS "full frame"

APCS = ARM Procedure Call Standard

Conventions for frame pointer and frame layout

Enable reliable stack introspection

CFLAGS to enable: `-mapcs-frame`

`r11` used as `fp`

Adds prolog/epilog to each function that sets up/tears down the standard frame and manages `fp`

Trace APCS full frame

Prolog

`push fp, sp*, lr, pc`

set `fp` to first word of stack frame

Body

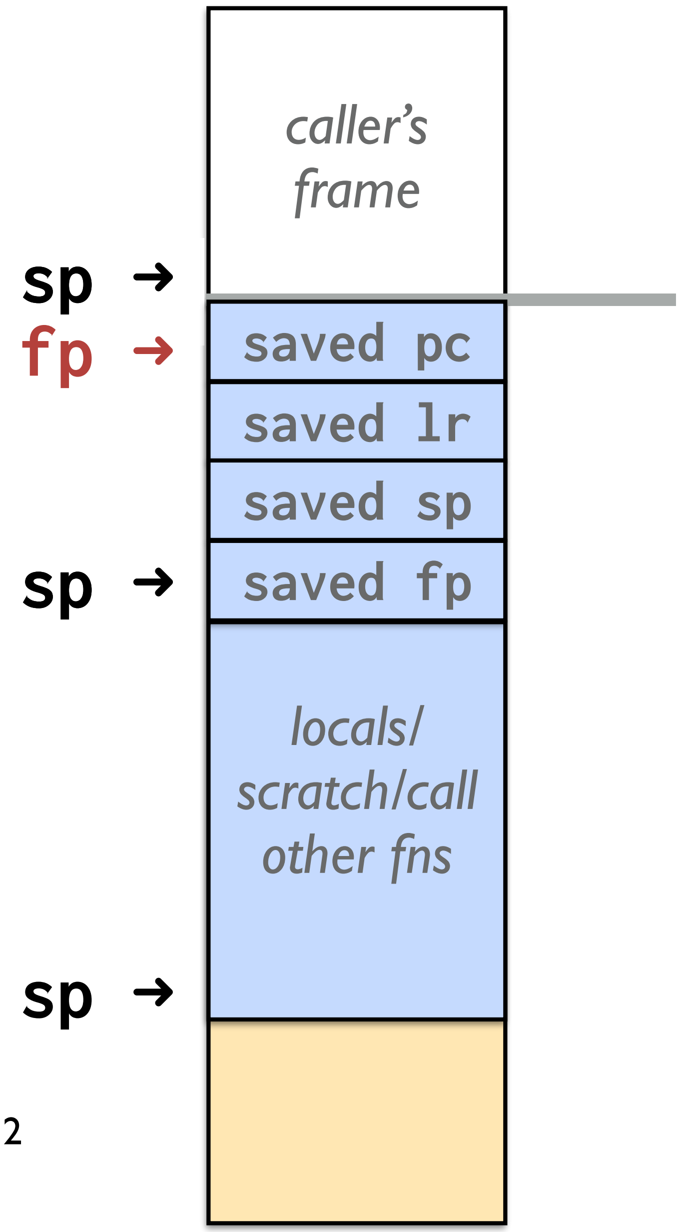
`fp` stays anchored

access data on stack `fp`-relative

offsets won't vary even if **sp** changing

Epilog

`pop fp, sp*, lr, pc*`

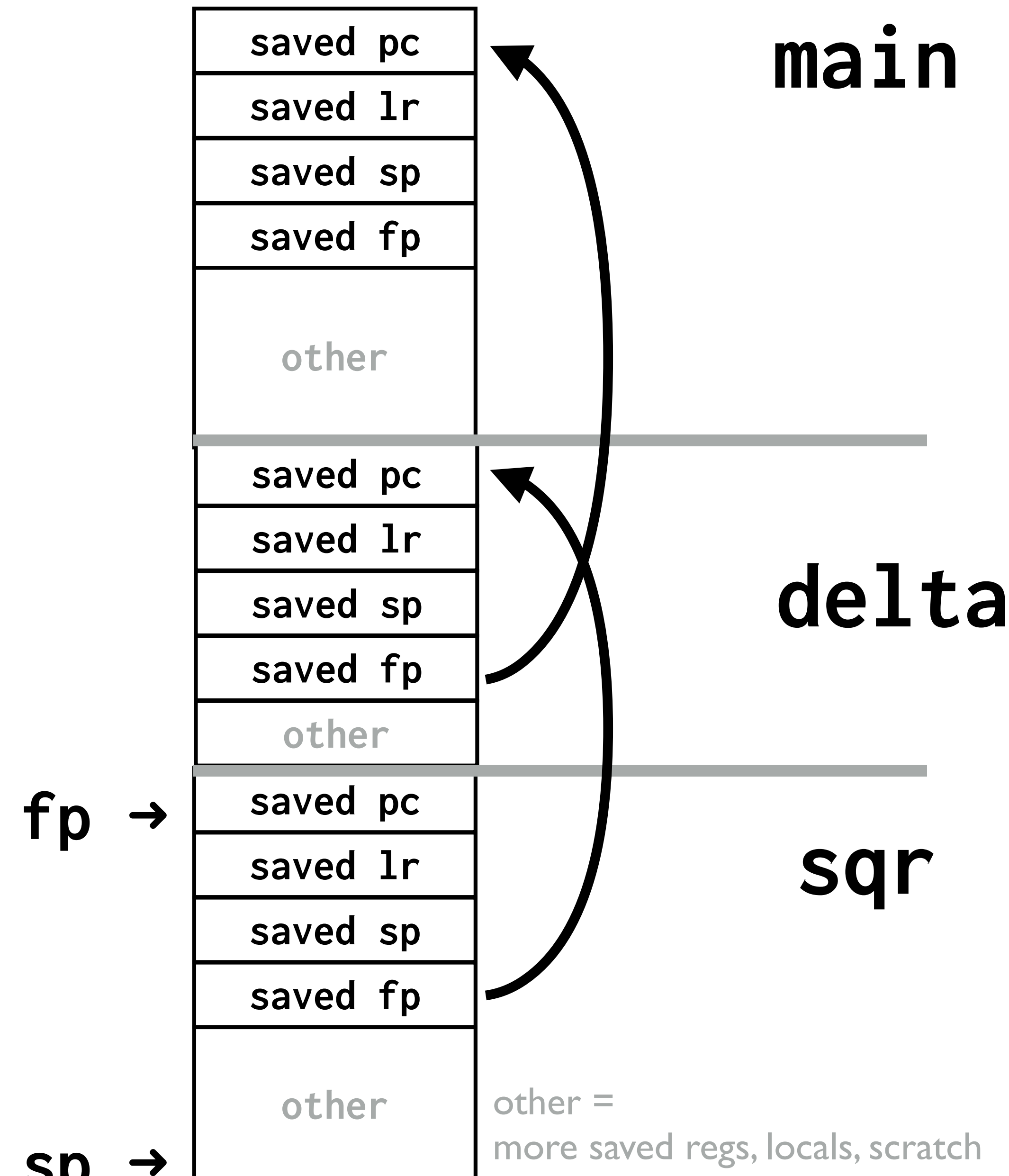


* I am fudging a bit about use of push and pop
The **sp** register cannot be directly pushed/popped, instead moved through r12
pc cannot be popped at end, is manually removed from stack

Frame pointers form linked chain

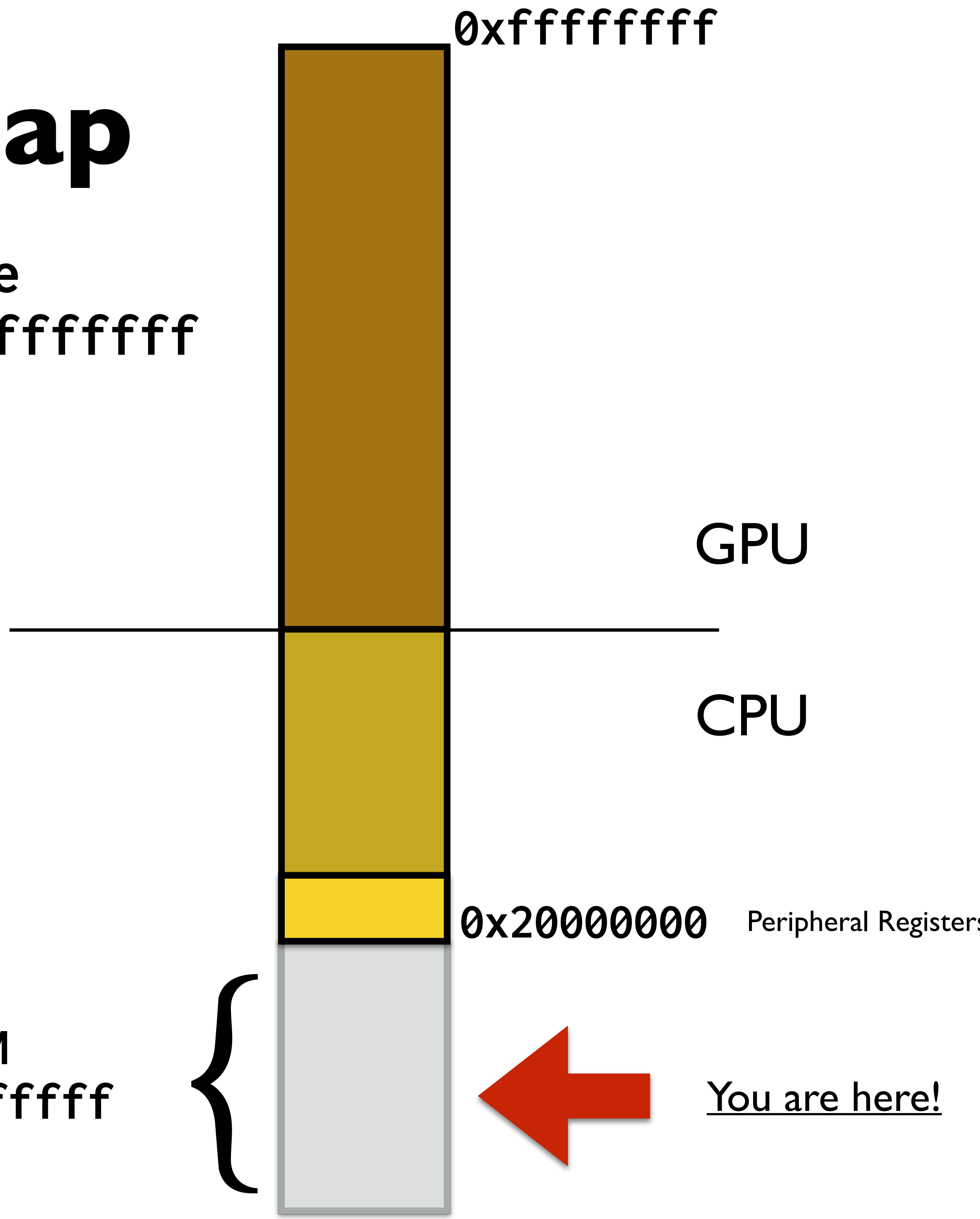
Can start at currently executing call (**sqr**) and back up to caller (**delta**), from there to its caller (**main**), who ends the chain

Deep dive into full frame coming up in this week's lab!



Memory Map

32-bit address space
Addresses 0 to 0xffffffff



512 MB of physical RAM
Addresses 0 - 0x1fffffff

Ref: BCM2835-ARM-Peripherals.pdf

```
SECTIONS
{
   .text 0x8000 : { *(.text.start)
                    *(.text*)}
   .rodata :      { *(.rodata*) }
   .data :        { *(.data*) }

   __bss_start__ = .;
   .bss :         { *(.bss*)
                    *(COMMON) }
   __bss_end__ = ALIGN(8);
}
```

```
_start:
    mov sp, #0x8000000
    mov fp, #0
    bl _cstart
```

```
void _cstart(void) {
  char *bss = &__bss_start__;
  while (bss < &__bss_end__)
     *bss++ = 0;
  }
  main();
}
```

Use this memory for heap☞

(zeroed data) .bss

(initialized data) .data

(read-only data) .rodata

.text

We have global storage ...

\+ **Convenient**

Fixed location, shared across entire program

No explicit allocate/deallocate

\+ **Fairly efficient, plentiful**

(But cost to send over serial line to bootloader)

+/- **Scope and lifetime is global**

No encapsulation, hard to track use/dependencies

One shared namespace, possibility of conflicts

Frowned upon stylistically

... and stack storage ...

\+ **Convenient**

Automatic alloc/dealloc on function entry/exit

\+ **Efficient, fairly plentiful**

(But finite size limit on total stack usage)

+/- **Scope/lifetime dictated by control flow**

Private to stack frame

Does not persist after function exits

Why do we also need a heap?

An example:

`code/heap/names.c`

Dynamic storage

+ Programmer controls scope/lifetime
Versatile, precise
Works for situations where global/stack do not

- Needs software runtime support
Library routines manage the heap memory and
Process allocation/deallocation requests

- C version is low on safety
No type safety (raw void *, number of bytes)
Much opportunity for error
(allocate wrong size, use after free, double free)

Heap interface

```
void *malloc (size_t nbytes);
void  free (void *ptr);
```

void* pointer

"Generic" pointer, a memory address

Type of pointee is not specified, could be any data

What you can do with a void*

Pass to/from function, pointer assignment

What you cannot do with a void*

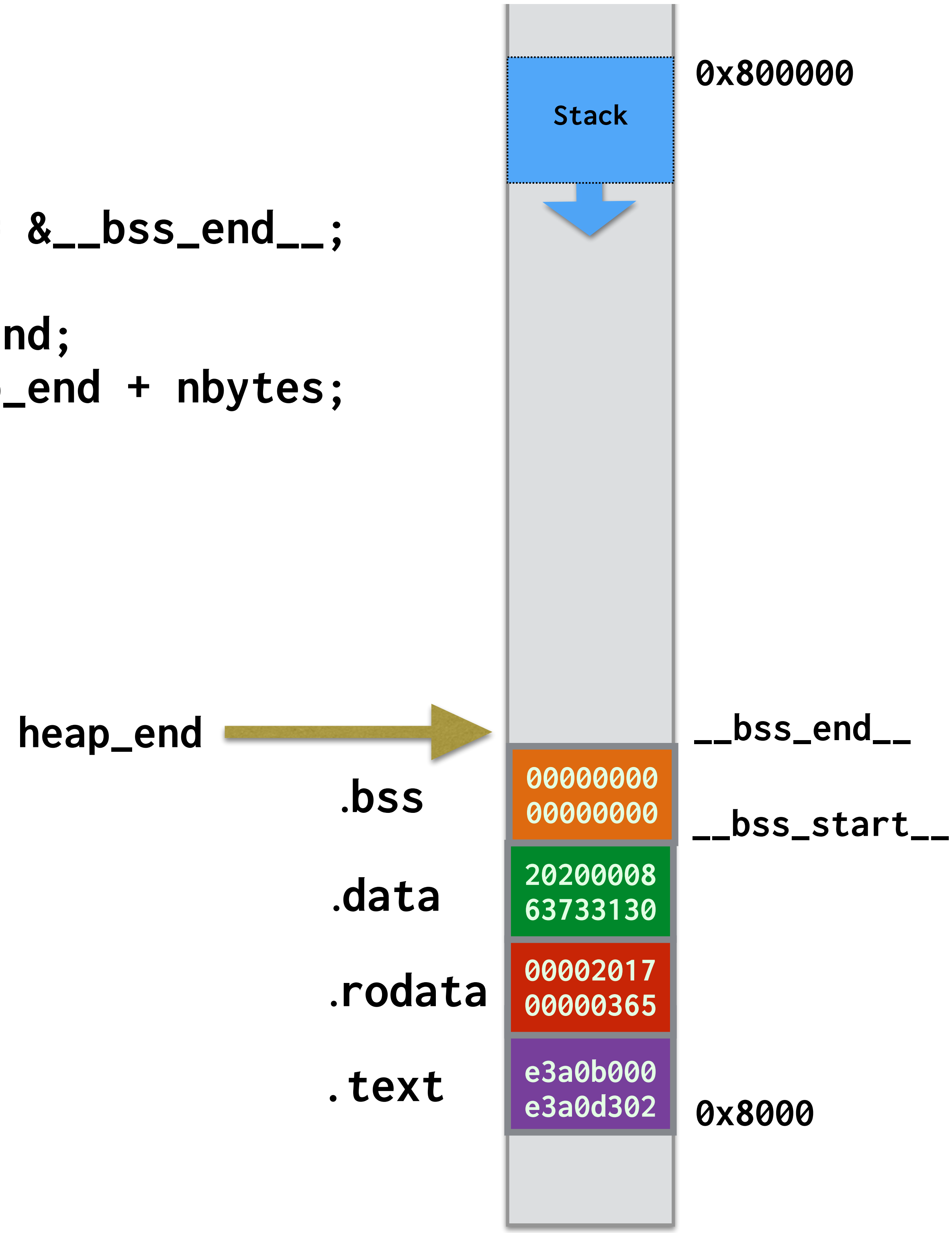
Cannot dereference (must cast first)

Cannot do pointer arithmetic (cast to char * to manually control scaling)

Cannot use array indexing (size of pointee not known!)

How to implement a heap

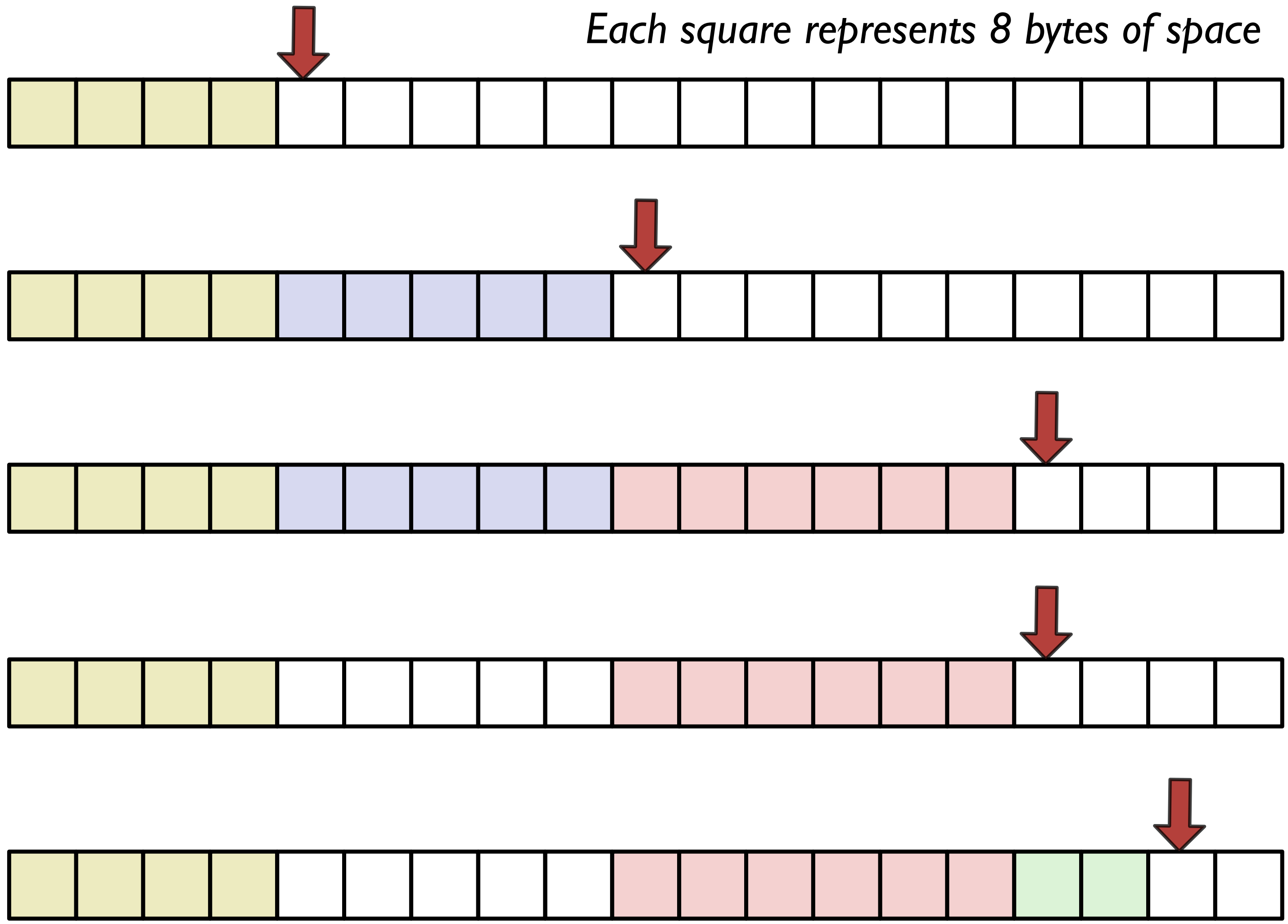
```
void *sbrk(int nbytes)
{
    static void *heap_end = &__bss_end__;

    void *prev_end = heap_end;
    heap_end = (char *)heap_end + nbytes;
    return prev_end;
}
```



Tracing the bump allocator

Each square represents 8 bytes of space

```
p1 = malloc(32)
p2 = malloc(40)
p3 = malloc(48)
free(p2)
p4 = malloc(16)
```



Bump Memory Allocator

`code/heap/malloc.c`

Evaluate bump allocator

+ Operations super-fast
+ Very simple code, easy to verify, test, debug

- No recycling/re-use

    (in what situations will this be problematic?)

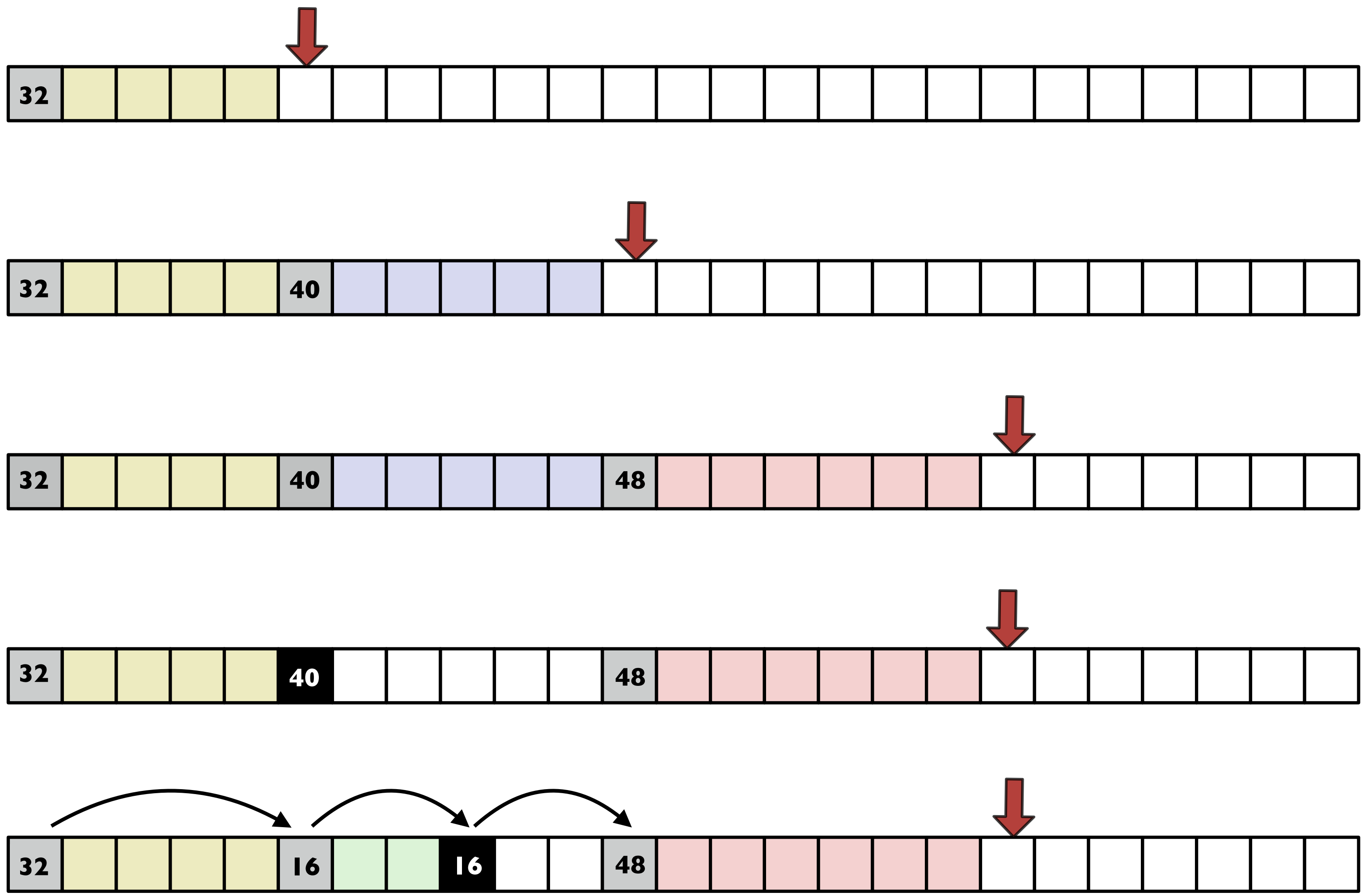
- Sad consequences when `sbrk()` advances into stack

    (what can we do about that?)

Pre-block header, implicit list

Each square represents 8 bytes, header records size of payload in bytes

```
p1 = malloc(32)
p2 = malloc(40)
p3 = malloc(48)
free(p2)
p4 = malloc(16)
```



Header struct on each block

```
struct header {
    unsigned int size;
    unsigned int status;
};                              // sizeof(struct header) = 8 bytes

enum { IN_USE = 0, FREE = 1};


void *malloc(size_t nbytes)
{
    nbytes = roundup(nbytes, 8);
    size_t total_bytes = nbytes + sizeof(struct header);

    struct header *hdr = sbrk(total_bytes); // extend end of heap
    hdr->size = nbytes;
    hdr->status = IN_USE;
    return hdr + 1;     // return address at start of payload
}
```

Challenges for malloc client

- **Correct allocation (size in bytes)**
- **Correct access to block (within bounds, not freed)**
- **Correct free (once and only once, at correct time)**

What happens if you…

— forget to free a block after you are done using it?
— access a memory block after you freed it?
— free a block twice?
— free a pointer you didn't malloc?
— access outside the bounds of a heap-allocated block?

Challenges for malloc implementor

just `malloc` is easy 😎
`malloc` with `free` is hard 🤔
Efficient `malloc` with `free` ….Yikes! 🥵

Complex code (pointer math, typecasts)
Thorough testing is challenge (more so than usual)
Critical system component

correctness is non-negotiable!

Survival strategies:

draw pictures
printf (you've earned it!!)
early tests use examples small enough to trace by hand if need be
build up to bigger, more complex tests