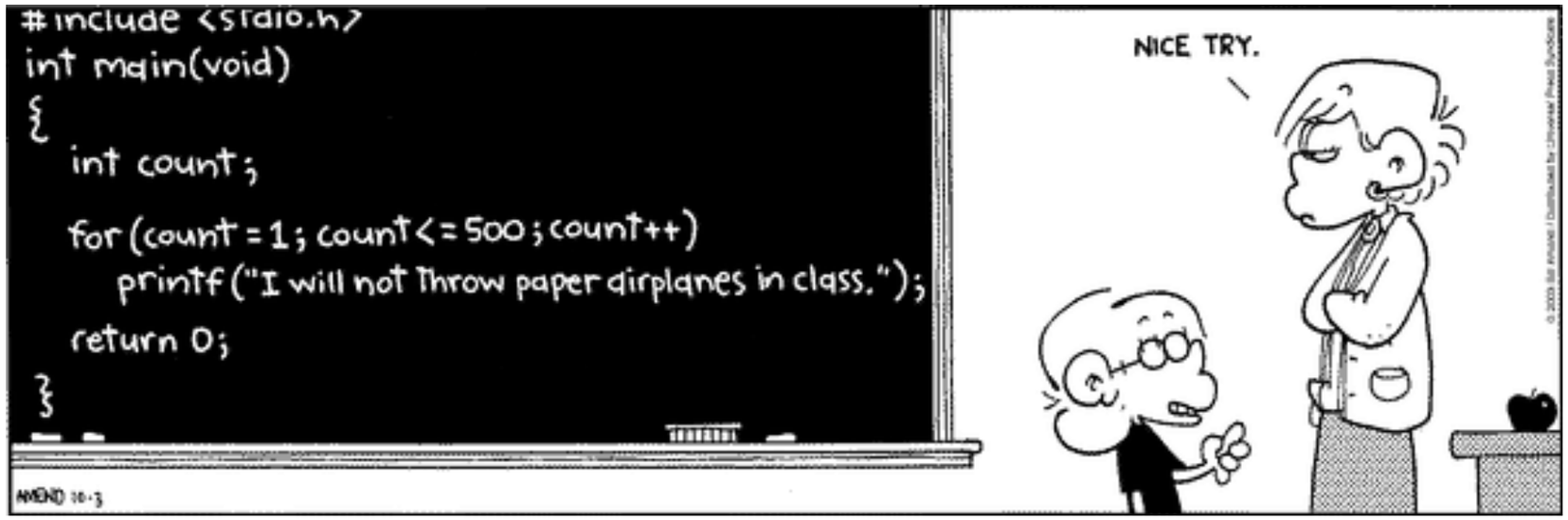
Admin

- 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

```
// start.s
lui   sp,0x50000
call  main
```

```
void main(void)
{
  delta(3, 7);
}

int delta(int a, int b)
{
  int diff = sqr(a) - sqr(b);
  return diff;
}
int sqr(int v)
{
  return v * v;
}
```

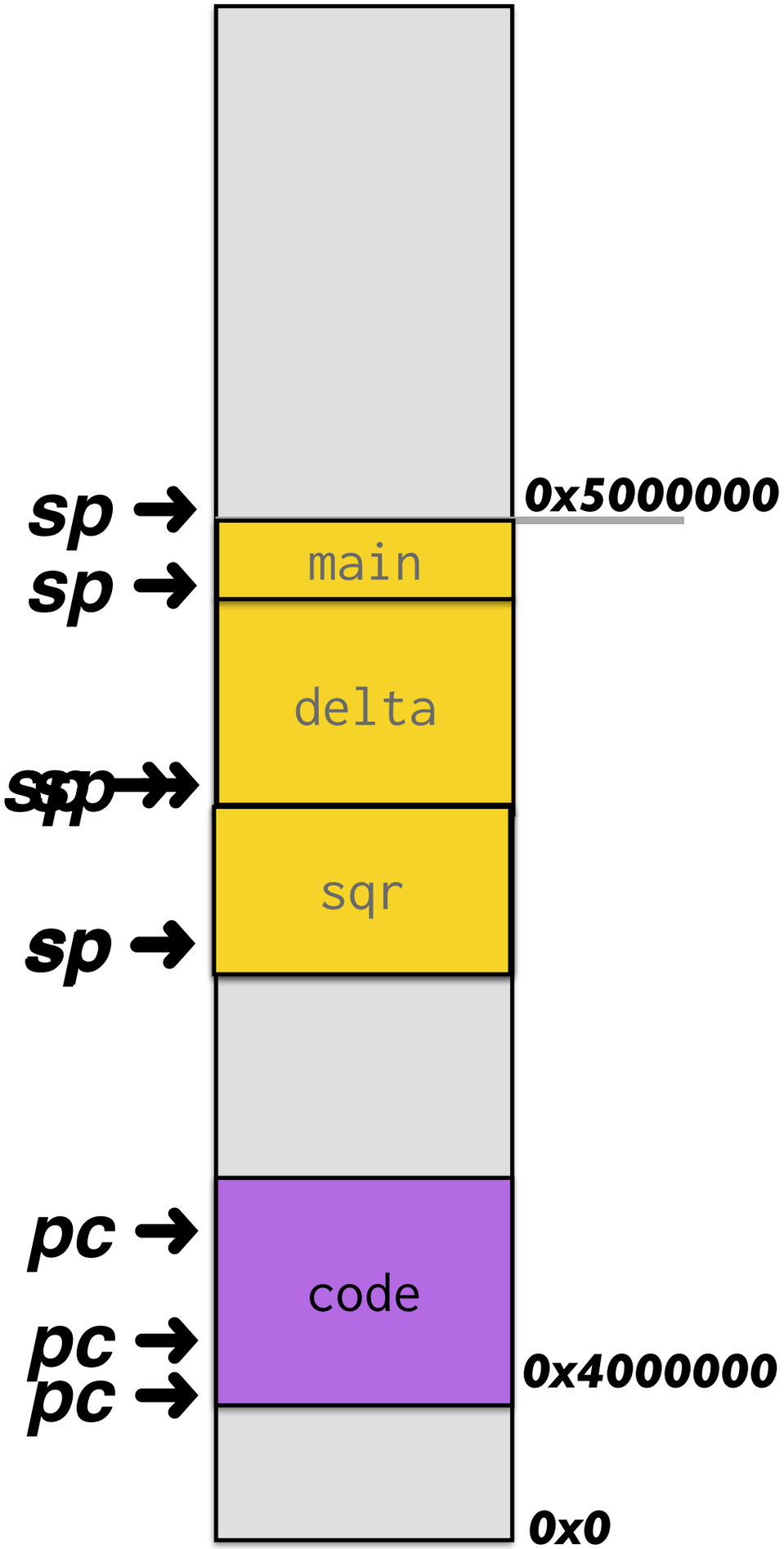
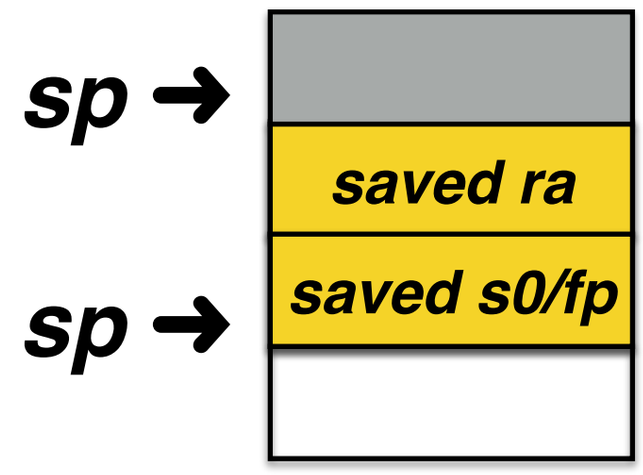


Diagram not to scale

Stack operation

```
addi  sp,sp,-16
sd    ra,8(sp)
sd    s0,0(sp)
addi  s0,sp,16
mv    a1,a0
call  sum
ld    ra,8(sp)
ld    s0,0(sp)
add   sp,sp,16
ret
```



Frame pointer

Conventions for frame pointer and stack frame layout

Enable reliable stack introspection

CFLAGS to enable: **-f-no-omit-frame-pointer**

s0 used as **fp**

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

Trace stack frames

Prolog

Adjust stack pointer to make space (16 + **rest)**

Store ra and fp to stack

Set **fp** to where sp was (end of prev stack frame)

Body

fp stays anchored

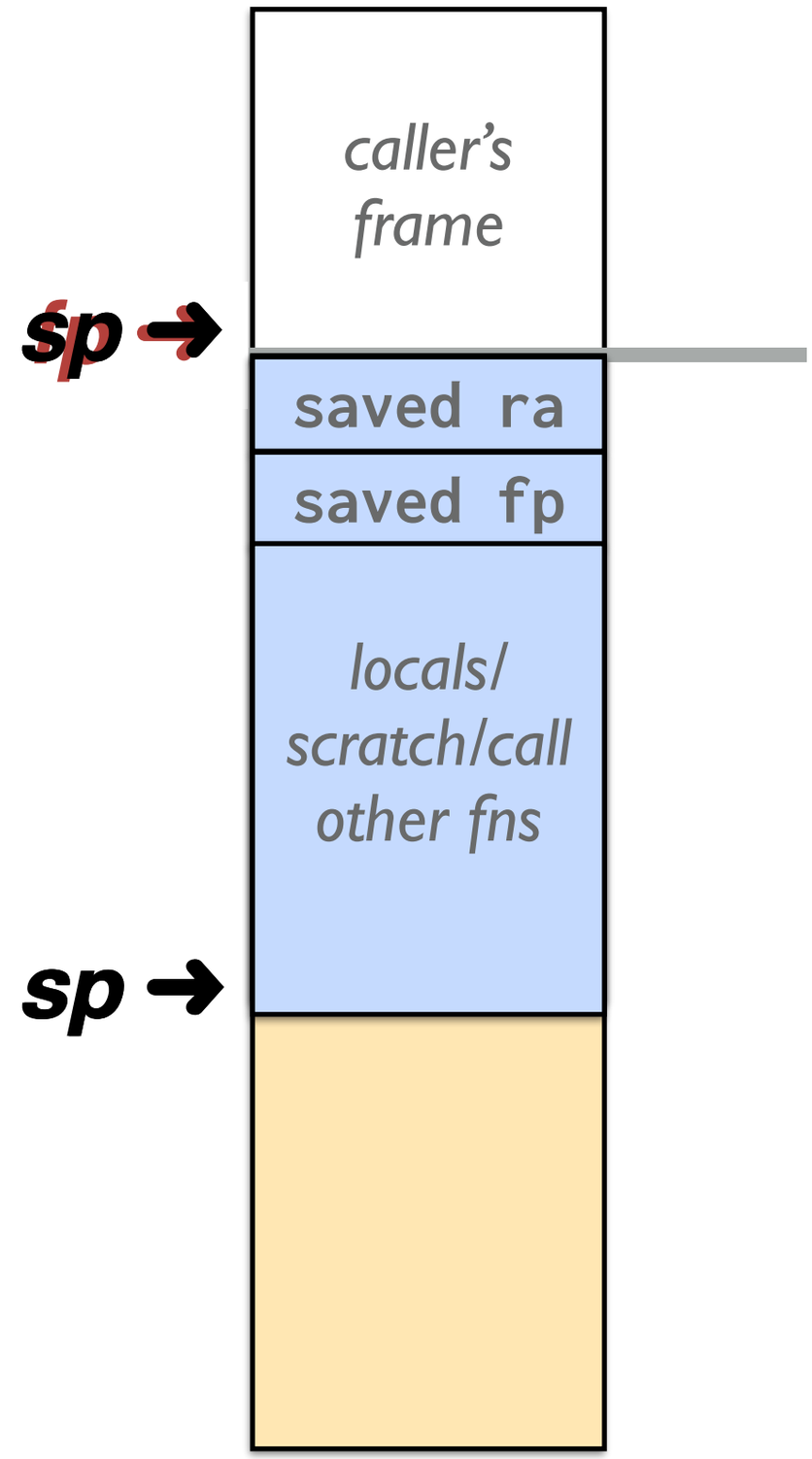
access data on stack **fp**-relative

fp offsets remain stable (even if **sp** changes)

Epilog

Restore ra, fp to saved values

Adjust sp to remove frame

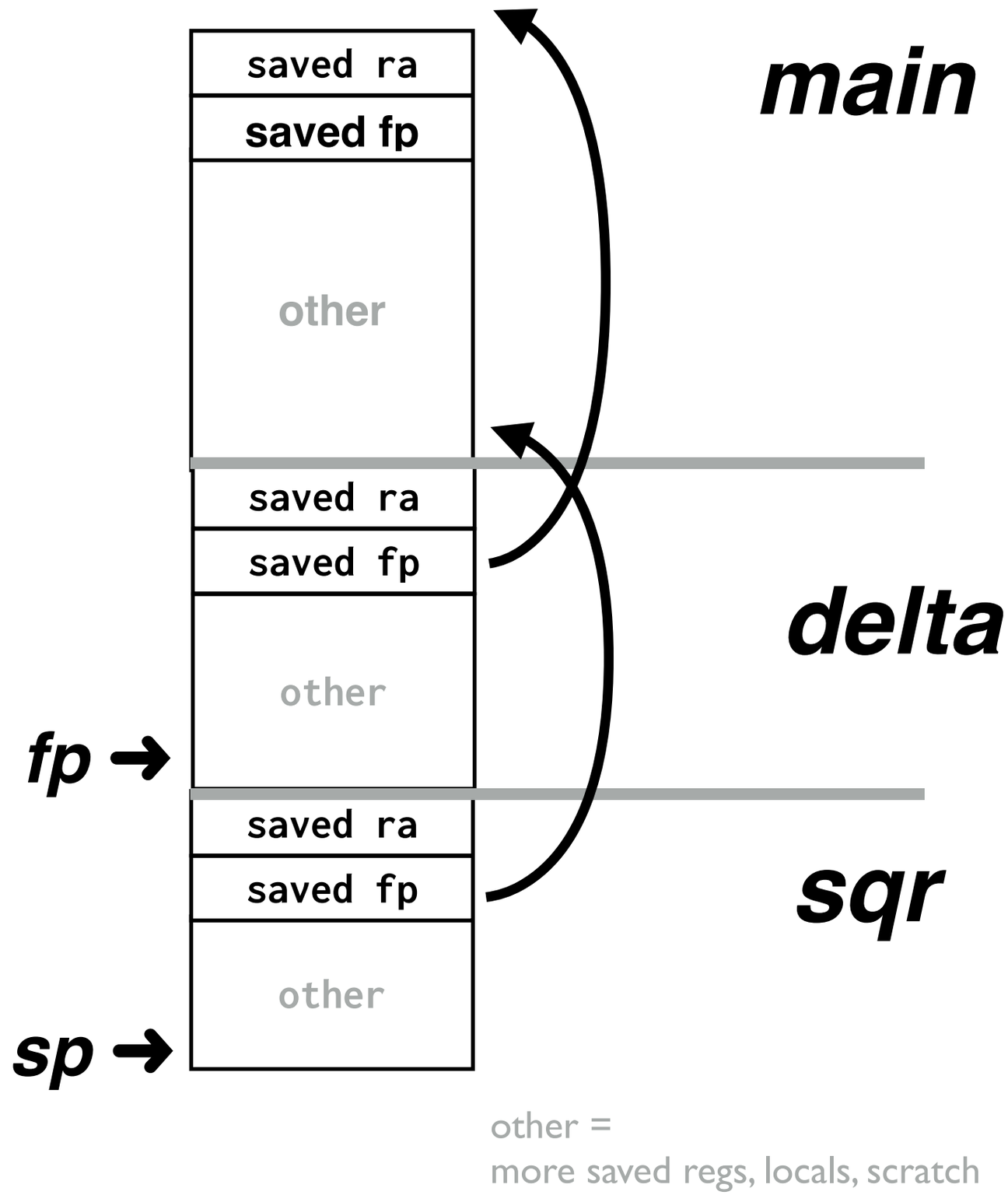


Frame pointers form linked chain

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

```
// start.s

// init fp = 0 as termination
li fp,0
lui sp,0x50000
jal main
```



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

Frame pointer Pros/Cons

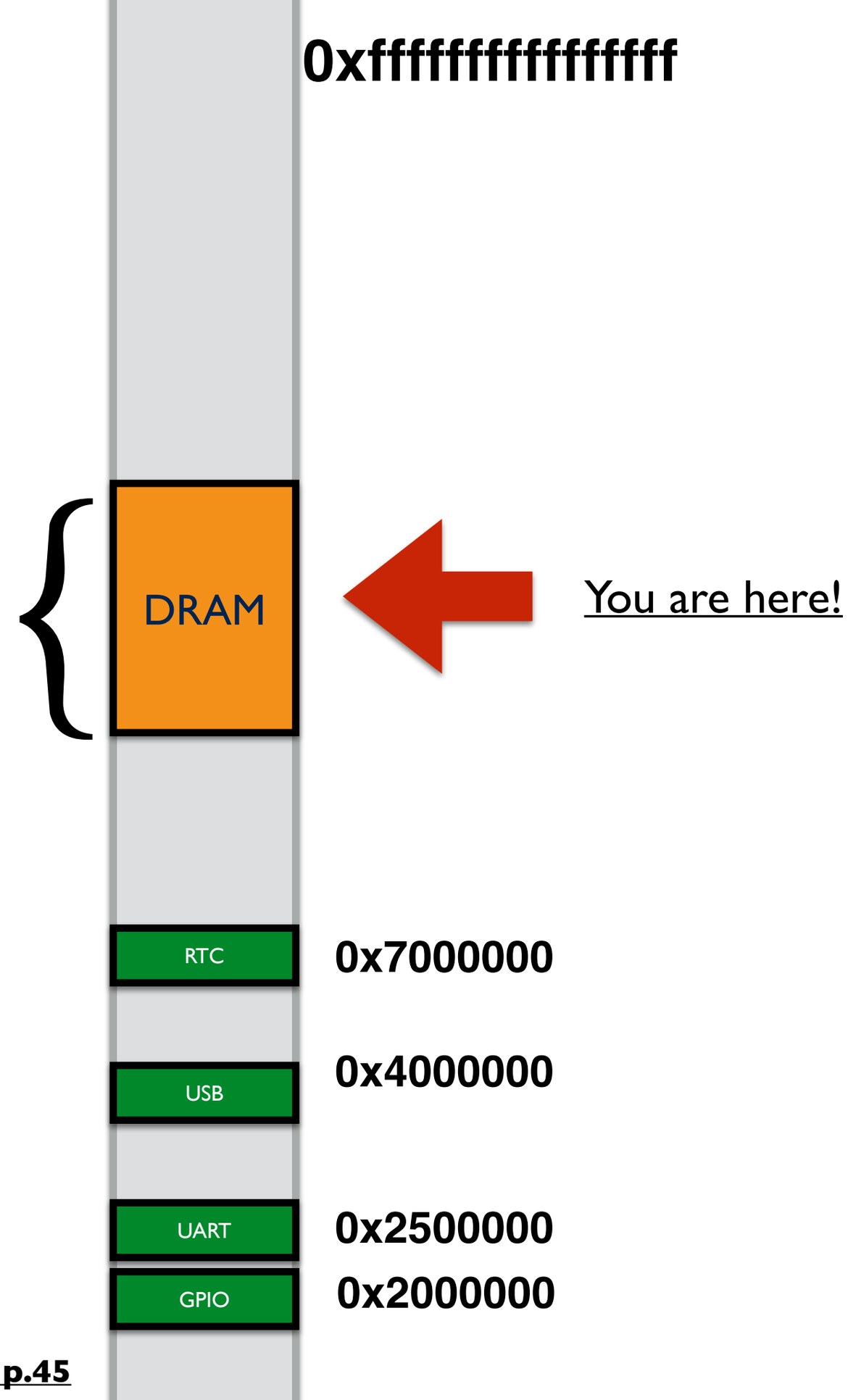
+ Anchored **fp**, offsets are constant
+ Standard frame layout enables runtime introspection
+ Backtrace for debugging
+ Unwind stack on exception

- Fixed overhead cost, every function call affected
- Adds 2-4 instructions to setup/tear down frame
- Adds 8-16 bytes to frame size

Memory Map

64-bit address space
Addresses 0 to 0xffffffff

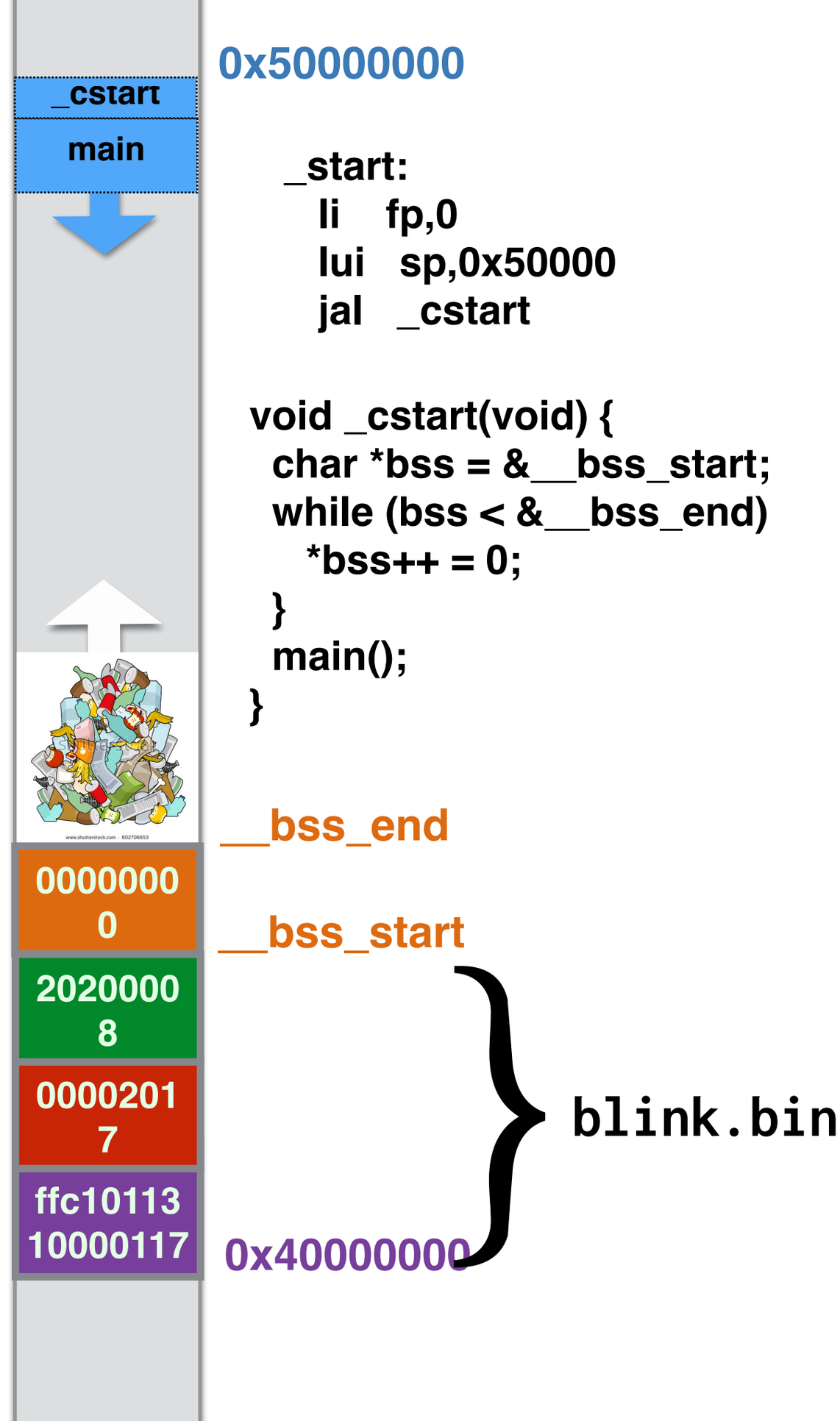
1 GB of physical RAM
0x40000000-0x7fffffff



Ref: D1-H User Manual p.45

```
SECTIONS
{
  .text 0x40000000 :{ *(.text.start)
                      *(.text*)}
  .rodata :          { *(.rodata*) }
  .data :            { *(.data*) }
  __bss_start        = . ;
  .bss :             { *(.bss*) }
  __bss_end          = . ;
}
```

Use this memory for heap☞




```
$ xfel write 0x40000000 blink.bin
$ xfel exec 0x40000000
```

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)

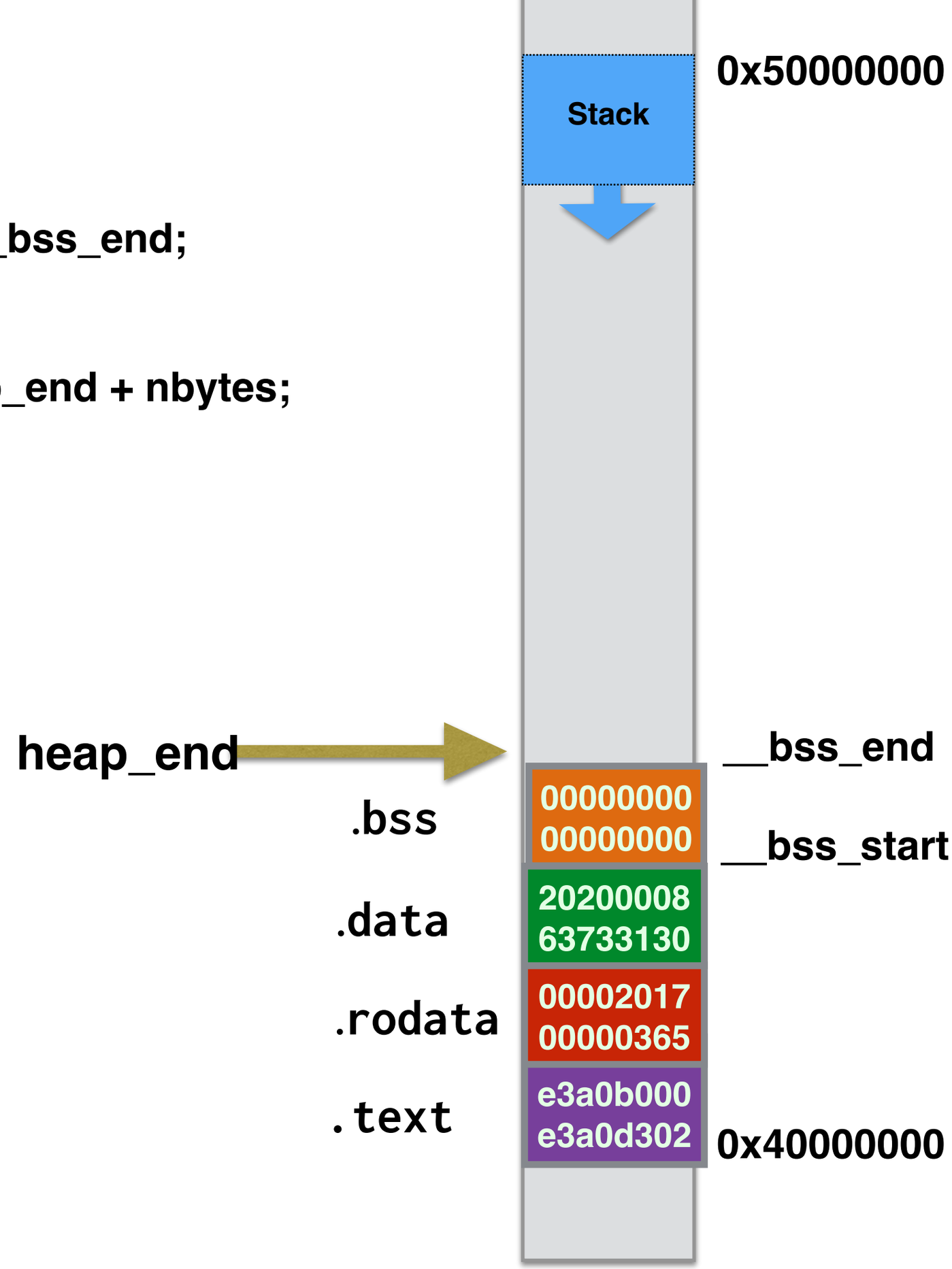
Cannot do pointer arithmetic (cast to char * to manually control scaling)

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

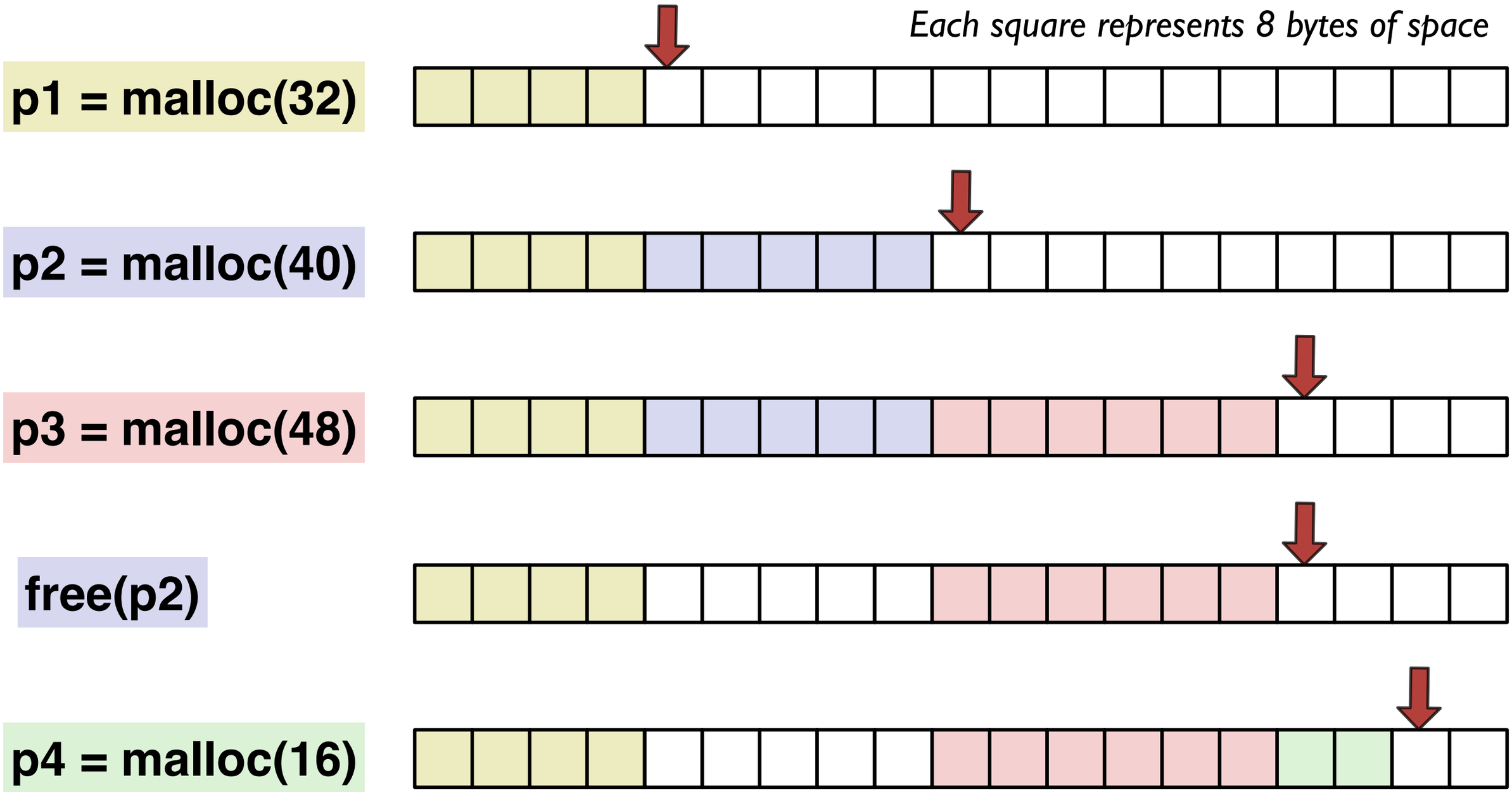
How to implement a heap

```
void *sbrk(int nbytes)
{
    static void *_cur_heap_end = &__bss_end;

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



Tracing the bump allocator



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?)

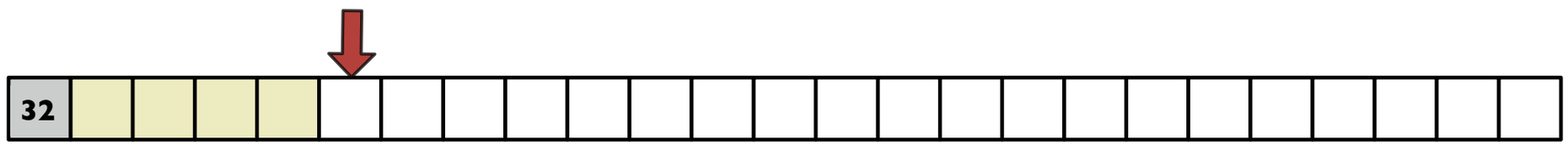
- Sad consequences when `sbrk()` advances into stack

  (what can we do about that?)

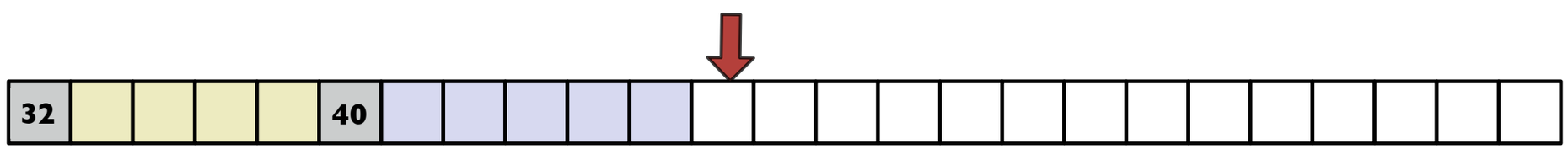
Pre-block header, implicit list

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

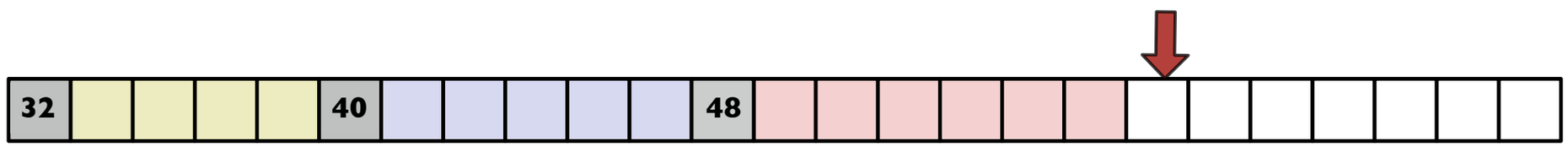
p1 = malloc(32)



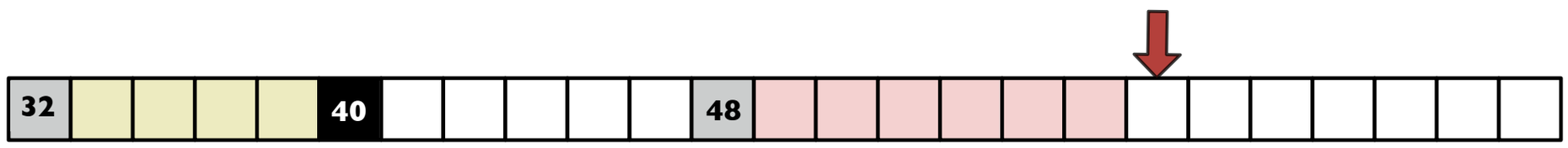
p2 = malloc(40)



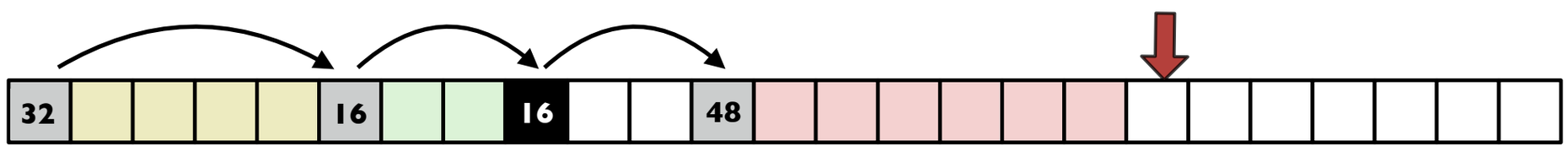
p3 = malloc(48)



free(p2)



p4 = malloc(16)



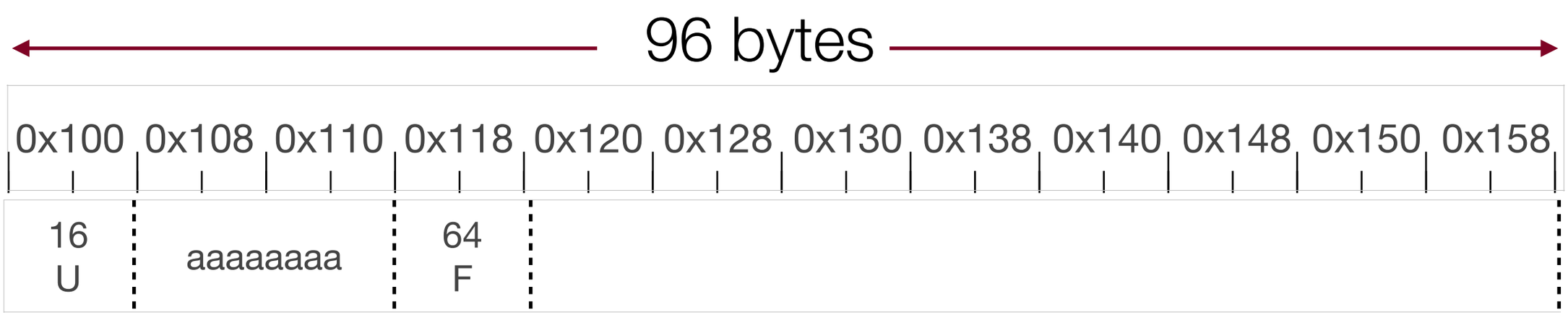
- In an implicit-free-list, the block header is actually *stored in the same memory area as the payload, and it generally precedes the payload.*

- Let's assume, for our purposes, that the heap starts at address 0x44000100, and we'll only show the last three digits on our diagram.

- 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.
- In this case, 8 bytes are taken up with the information that there are 88 Free (F) bytes ahead in the block.

```
a = malloc(16);
```

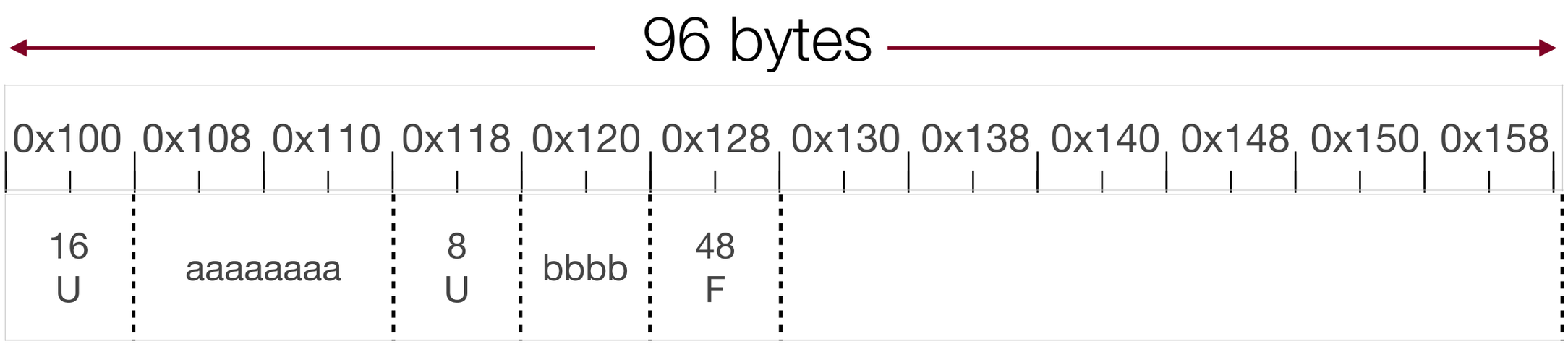
	Address	Value
e	0xffffe820	
d	0xffffe818	
c	0xffffe810	
b	0xffffe808	
a	0xffffe800	**0x108**



- 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);
```

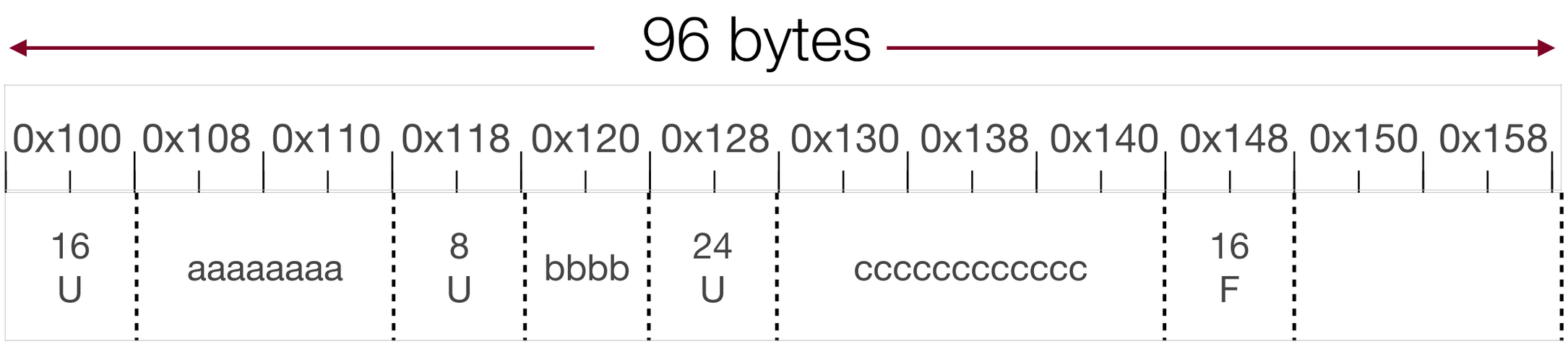
	Address	Value
e	0xffffe820	
d	0xffffe818	
c	0xffffe810	
b	0xffffe808	**0x120**
a	0xffffe800	**0x108**



- We changed the header to reflect the fact that 8 bytes are going to 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.

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

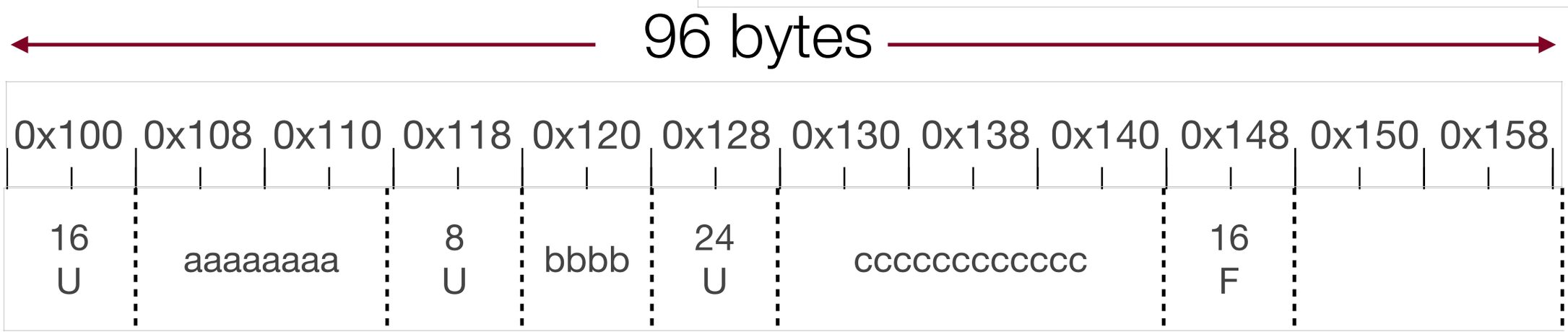
	Address	Value
e	0xffffe820	
d	0xffffe818	
c	0xffffe810	**0x130**
b	0xffffe808	**0x120**
a	0xffffe800	**0x108**



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

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

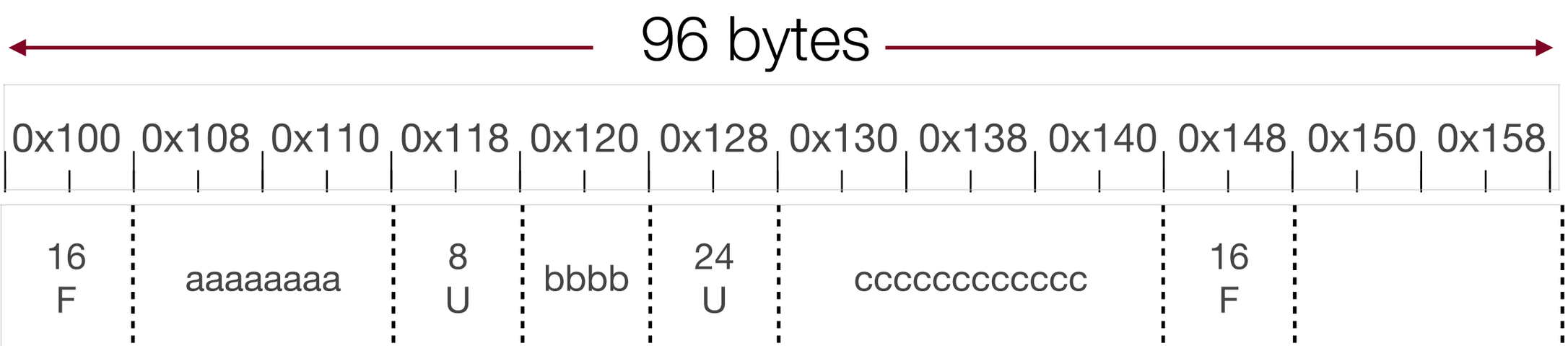
	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**



- 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.

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

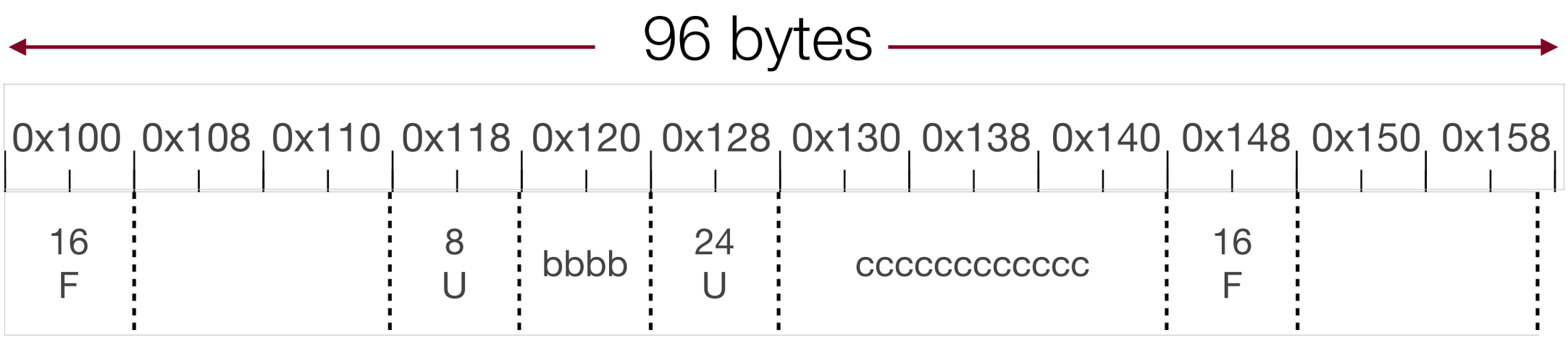
	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**



- The diagram now reflects the free.
- The change to the diagram was subtle — the *only* thing that changed was that the block header now says "F" (free) instead of "U" (used). This is because the data remains, but it can be written over any time after we reassign that block — this can cause bugs! For clarity sake, on the next page, we'll remove the `aaaaaaaa`, but know that the heap allocator doesn't wipe it clean (this another reason that **`free`** can be fast!)

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

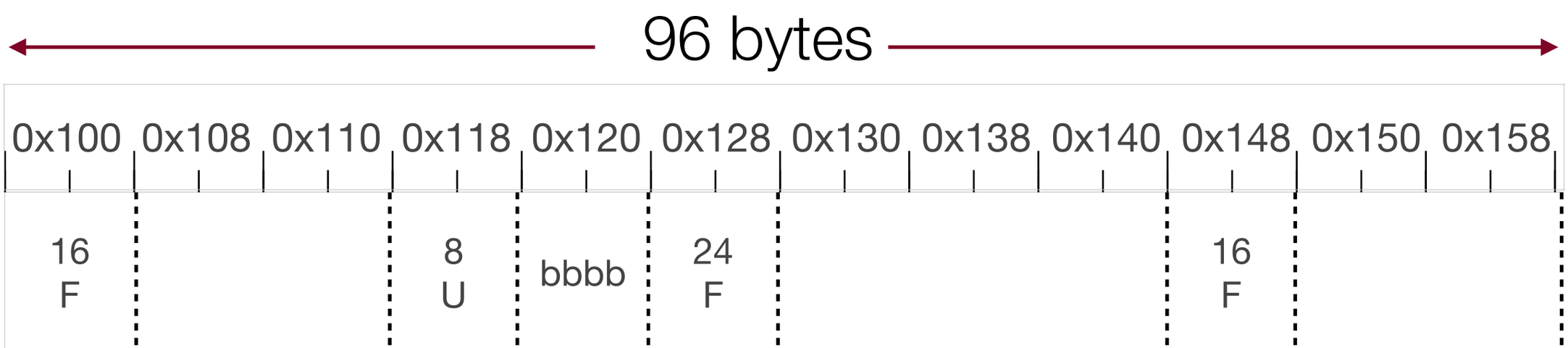
	Address	Value
e	0xffffe820	
d	0xffffe818	
c	0xffffe810	**0x130**
b	0xffffe808	**0x120**
a	0xffffe800	**0x108**



- 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.
- On the next slide, we will remove the `cccccccccccc`, but again: it is *not* cleared out, and we're just doing this for the sake of clarity on the diagram.

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

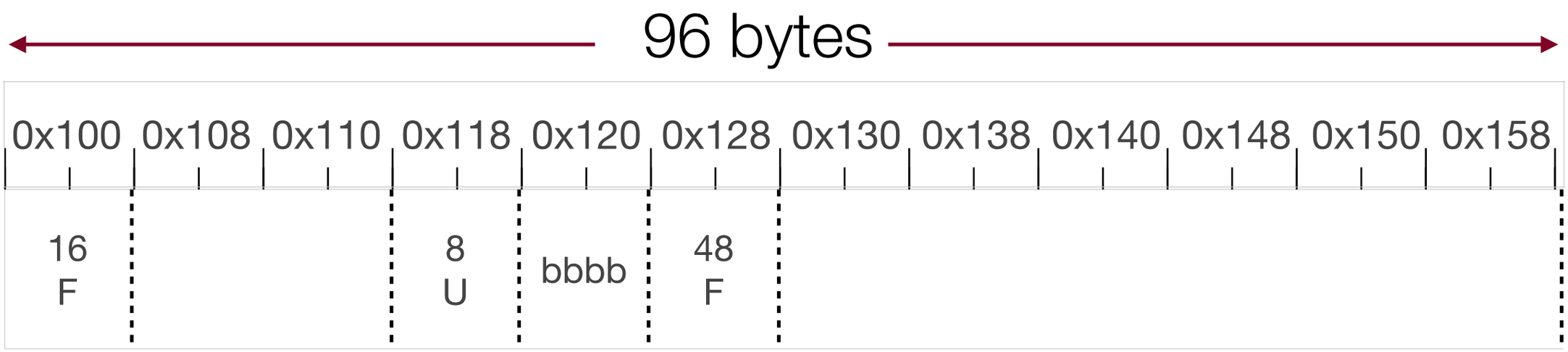
	Address	Value
e	0xffffe820	
d	0xffffe818	
c	0xffffe810	**0x130**
b	0xffffe808	**0x120**
a	0xffffe800	**0x108**



- This diagram shows one possible result of the `free`. 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!)

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

	Address	Value
e	`0xffffe820`	
d	`0xffffe818`	
c	`0xffffe810`	**`0x130`**
b	`0xffffe808`	**`0x120`**
a	`0xffffe800`	**`0x108`**



- 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.

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