Goals for today

Heap allocator

Bump allocator -> implicit list

Steps toward C mastery

C language, advanced edition

Hallmarks of good software

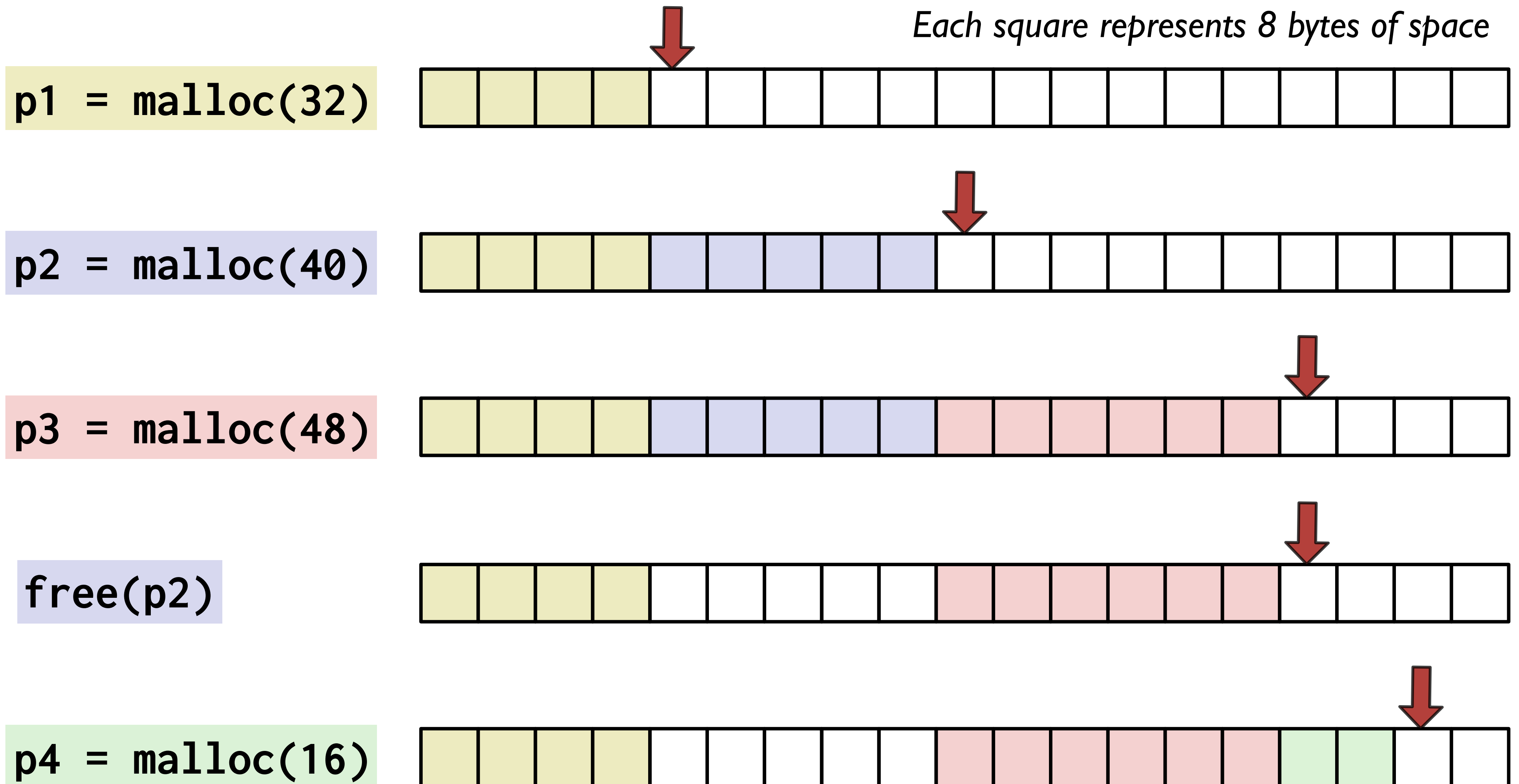
Tuning your process: best practices

Check-in

Halfway point on our journey

Celebrate/commiserate/share

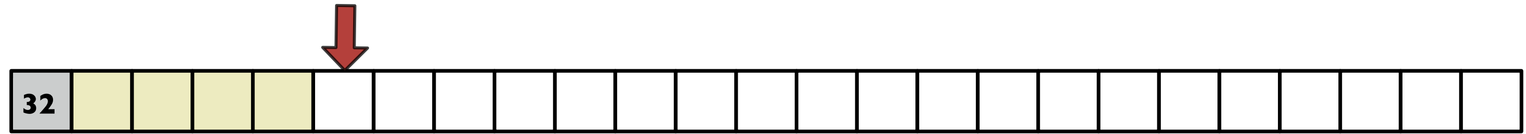
Tracing the bump allocator



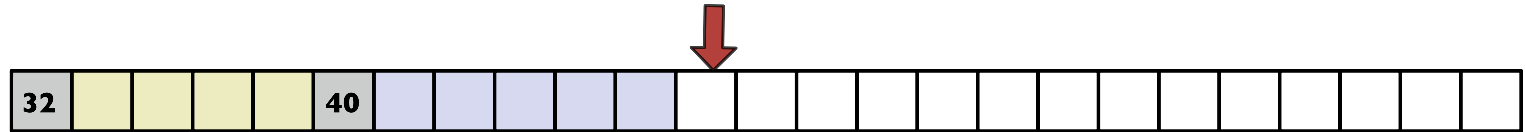
Pre-block header, implicit list

Each square represents 8 bytes of space, size recorded as count of 8-byte words

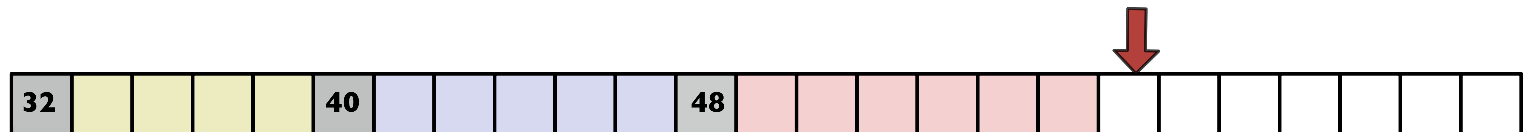
`p1 = malloc(32)`



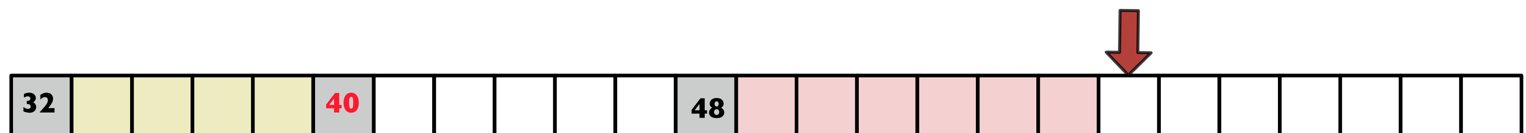
`p2 = malloc(40)`



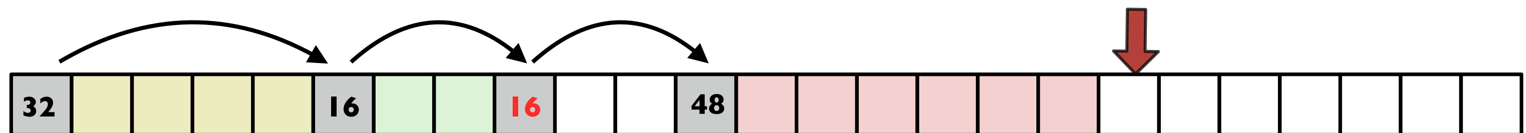
`p3 = malloc(48)`



`free(p2)`



`p4 = malloc(16)`



Header struct

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

enum { IN_USE = 0, FREE = 1};

static void *heap_end = &__bss_end__;



void *malloc(size_t nbytes)
{
    nbytes = roundup(nbytes, 8);

    struct header *hdr = (struct header *)heap_end;
    heap_end  = (char *)heap_end + nbytes + sizeof(struct header);
    hdr->size = nbytes;
    hdr->status = IN_USE;
    return hdr + 1;     // payload
}
```

Typecasts

Arrays/structs allow access neighboring data by name/index (underlying mechanism is base address + offset)

Impose order/layout on memory, improve on raw addresses

Pointer typecast allows you to claim the pointee type at address

Pointer operations that behave differently due to cast:

- Dereference: number of bytes read/written
- Pointer arithmetic, array access: scaling for index
- Field access by name (if struct pointee)

Cast cannot verify or convert pointee to matching type, you are responsible for being sure cast is correct

Powerful but no safety — use only when you absolutely must

Pointers, arrays, structures

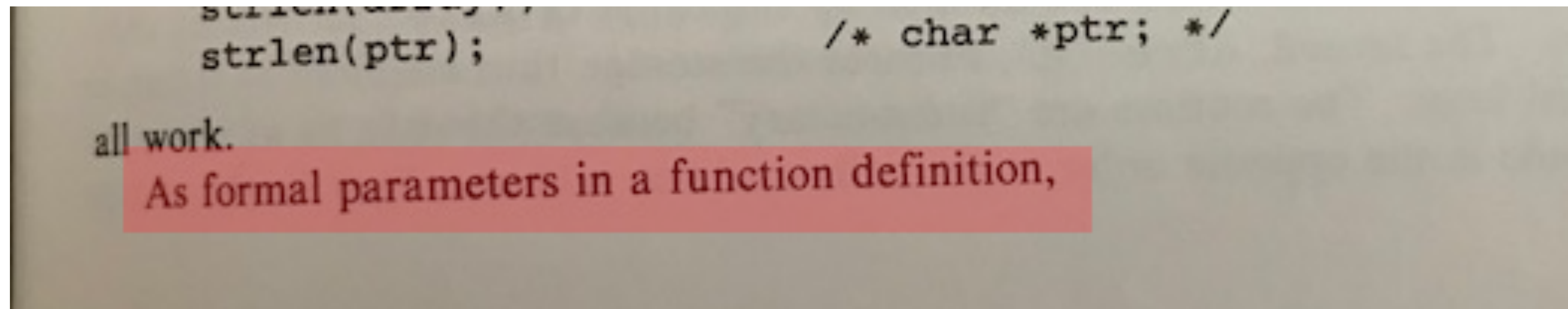
Will we ever know enough?

CULTURE FACT:

IN CODE, IT'S NOT CONSIDERED RUDE TO POINT.

A most unfortunate page break

K&R bottom of page 99

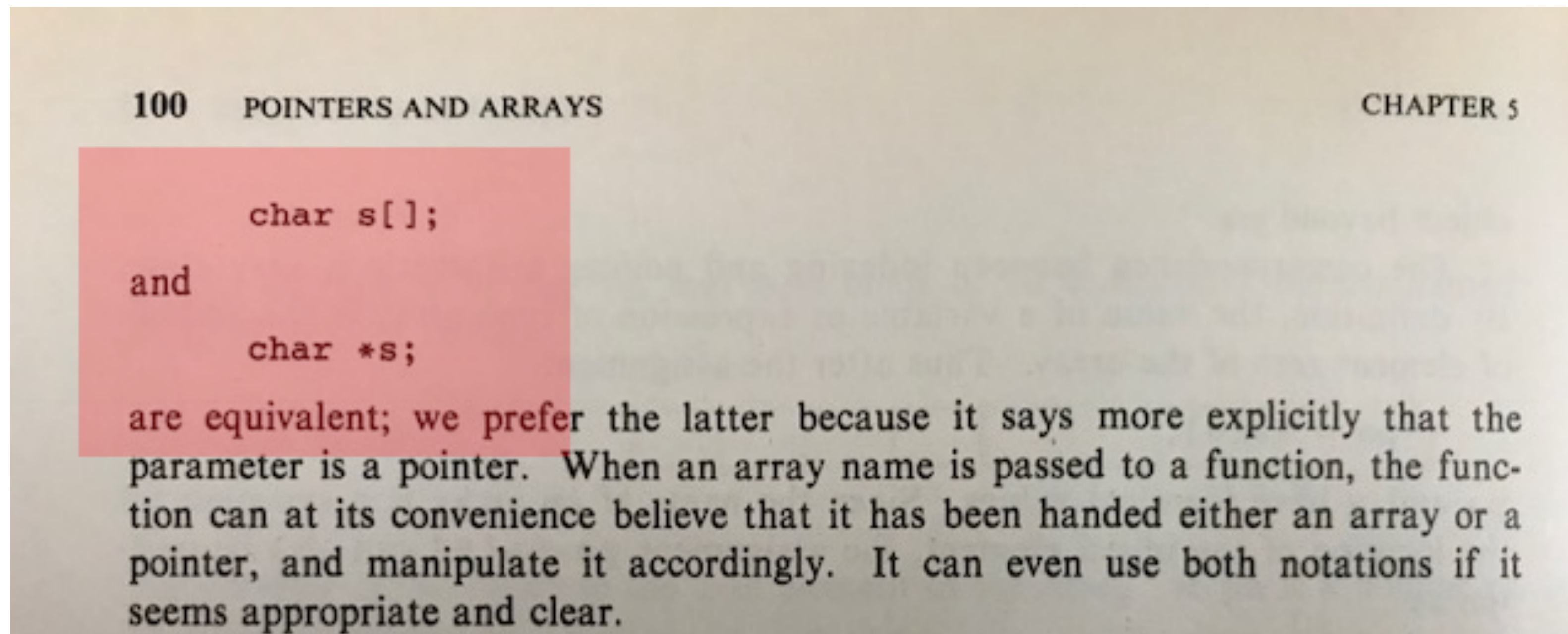


```
    strlen(ptr);                    /* char *ptr; */
```

all work.

As formal parameters in a function definition,

K&R top of page 100




```
    char s[];
```

and

```
    char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

Pointers and arrays, the same thing?

Arrays and pointers, arrays of pointers

```
void strings(void)
{
    char *a;
    char *b = "happy";
    char c[10];
    char d[] = "dopey";
    char *dwarves[] = {a, b, c, d, e};

    a[0] = 'z';
    b[0] = 'z';
    c[0] = 'z';
    d[0] = 'z';
}
```

C structs

Convenient and readable way to allocate memory and name offsets from a pointer

```
struct item {
  char name[10];
  int sku;
  int price;
};
```

How big is this structure?
How is it laid out in memory (on an ARM)?

Data alignment

"Natural" or "self" alignment

    4-byte store/load on address that is multiple of 4

    8-byte on multiple of 8

System typically optimized for natural alignment

Unaligned access may be allowed (usually at some performance cost) or disallowed (exception). Worst option would be allowed, but wrong. Guess which case we get on the Pi?

To avoid unaligned access, stack and heap always align/pad to 8 (sizeof largest primitive)

Function pointers

One of the more mind-bending features of C

Can treat functions as data, execute/refer to code by address

```
void censor(char *s, int (*pred)(char))
{
    for (int i = 0; s[i]; i++) {
        if (pred(s[i]))
            s[i] = '-';
    }
}
```

Useful tool
https://cdecl.org

Writing Good Systems Software

```c
void serial_init() {
    unsigned int ra;

    // Configure the UART
    PUT32(AUX_ENABLES, 1);
    PUT32(AUX_MU_IER_REG, 0); // Clear FIFO
    PUT32(AUX_MU_CNTL_REG, 0); // Default RTS/CTS
    PUT32(AUX_MU_LCR_REG, 3); // Put in 8 bit mode
    PUT32(AUX_MU_MCR_REG, 0); // Default RTS/CTS auto flow control
    PUT32(AUX_MU_IER_REG, 0); // Clear FIFO
    PUT32(AUX_MU_IIR_REG, 0xC6); // Baudrate
    PUT32(AUX_MU_BAUD_REG, 270); // Baudrate

    // Configure the GPIO lines
    ra = GET32(GPFSEL1);
    ra &= ~(7 << 12); //gpio14
    ra |= 2 << 12;    //alt5
    ra &= ~(7 << 15); //gpio15
    ra |= 2 << 15;    //alt5
    PUT32(GPFSEL1,ra);
    PUT32(GPPUD,0);
    for (ra = 0; ra < 150; ra++) dummy(ra);
    PUT32(GPPUDCLK0, (1 << 14) | (1 << 15));
    for (ra = 0; ra < 150; ra++) dummy(ra);
    PUT32(GPPUDCLK0, 0);

    PUT32(AUX_MU_CNTL_REG, 3);
}
```

```
void uart_init(void)
{
    gpio_set_function(GPIO_TX, GPIO_FUNC_ALT5);
    gpio_set_function(GPIO_RX, GPIO_FUNC_ALT5);

    int *aux = (int*)AUX_ENABLES;
    *aux |= AUX_ENABLE;

    uart->ier  = 0;
    uart->cntl = 0;
    uart->lcr  = MINI_UART_LCR_8BIT;
    uart->mcr  = 0;
    uart->iir  = MINI_UART_IIR_RX_FIFO_CLEAR |
                 MINI_UART_IIR_RX_FIFO_ENABLE |
                 MINI_UART_IIR_TX_FIFO_CLEAR |
                 MINI_UART_IIR_TX_FIFO_ENABLE;

    // baud rate ((250,000,000/115200)/8)-1 = 270
    uart->baud = 270;
    uart->cntl = MINI_UART_CNTL_TX_ENABLE |
                 MINI_UART_CNTL_RX_ENABLE;
}
```

A tale of two bootloaders

https://github.com/dwelch67/raspberrypi/blob/master/bootloader03/bootloader03.c

https://github.com/cs107e/cs107e.github.io/blob/master/_labs/lab4/code/bootloader/bootloader.c

Thank you, David Welch, we owe you!

If I have seen further than others, it is by standing upon the shoulders of giants.
— Isaac Newton

If I have not seen as far as others, it is because there were giants standing on my shoulders.
— Hal Abelson

The value of code reading

Consider:

Is is clear what the code intends to do?
Are you confident of the author's understanding?
Would you want to maintain this code?

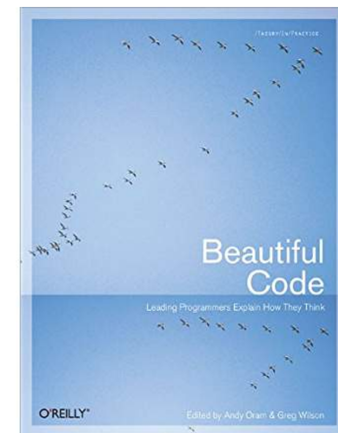
Open source era is fantastic!

https://github.com/dwelch67/raspberrypi

https://www.musl-libc.org

https://git.busybox.net/busybox/

https://sourceware.org/git/?p=glibc.git



Be a section leader: you will read a lot of student code and learn much!

What makes for good style?

Adopts the conventions of the existing code base

Common, idiomatic choices where possible

Logical decomposition, easy to follow control flow

Re-factored for code unification/re-use

Easy to understand and <u>maintain</u>

Writing good system software

Can be terse and unforgiving, details matter

Everything depends on it, bugs have consequences

Comment not just what code does, but <u>why</u>

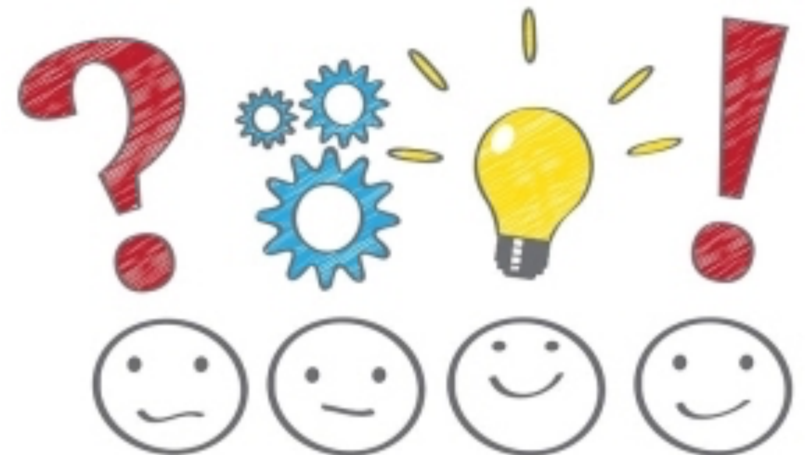
Lesson: If someone else had to fix a bug in my code, what could I do to make their job easier?

Thoughts on best practices

Designing, writing, testing, debugging, …

Which parts of your process are working well for you?

Which parts are not?



Development process

— Write the high-quality version first (and only!)

— Decompose <u>problems</u>, not <u>programs.</u>

— Implement from bottom up, each step should be testable

— Unifying common code means less code to write, test, debug, and maintain!

— Don't depend on comments to make up for lack of readability in the code itself

— One-step build

Tests are your friend!

Think of the tests as a specification of what your code should do. Assertions will clarify your understanding how it should work.

Implement the simplest possible thing first, then test it. A simple thing is more much likely to work than a complex thing. Go forward in epsilon-steps.

Never delete a test. Keep re-running all of them at each step. You may break something that used to work and you want to hear about it.

Debugging for the win

Rule #1: be systematic

Focus on what is testable/observable.

Hunches can be good, but if fact and hunch collide, fact wins.

Engineering habits

Test, test, test, and test some more; Test as you go

Always start from a known working state, take small steps

Make things visible (printf, logic analyzer, gdb)

Methodical (D&C), not random, search for solution. Form hypotheses and perform experiments

Fast prototyping, embrace automation, one-click build

Don't be frustrated by bugs, relish the challenge, take frequent breaks