Goals for today

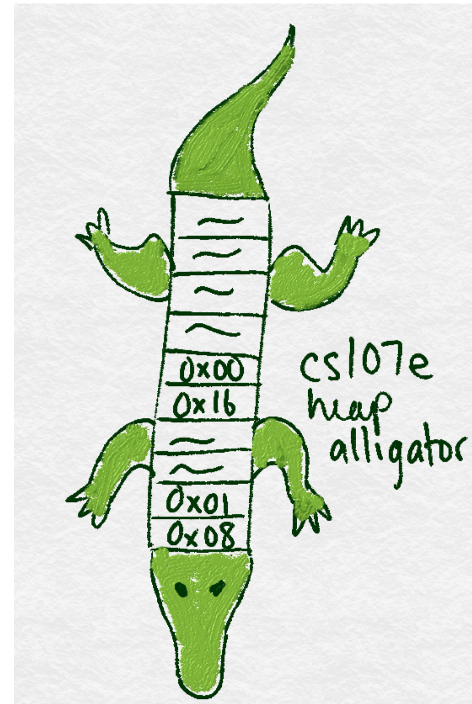
Heap allocator

Bump allocator -> implicit list

Steps toward C mastery

Hallmarks of good software

Tuning your process: best practices

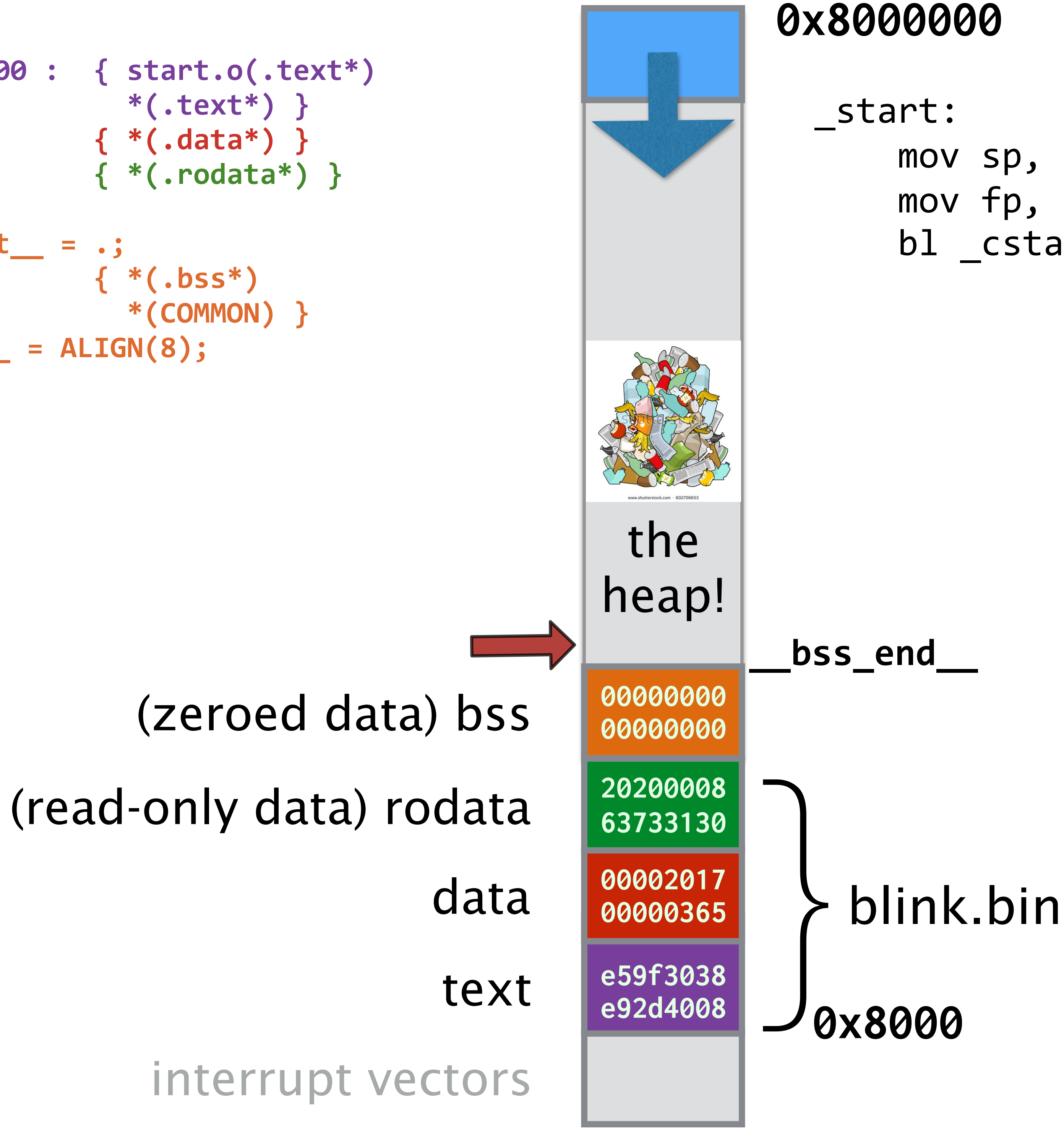


Admin

Keeping everyone sane, happy, and healthy

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

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




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

Tracing the bump allocator

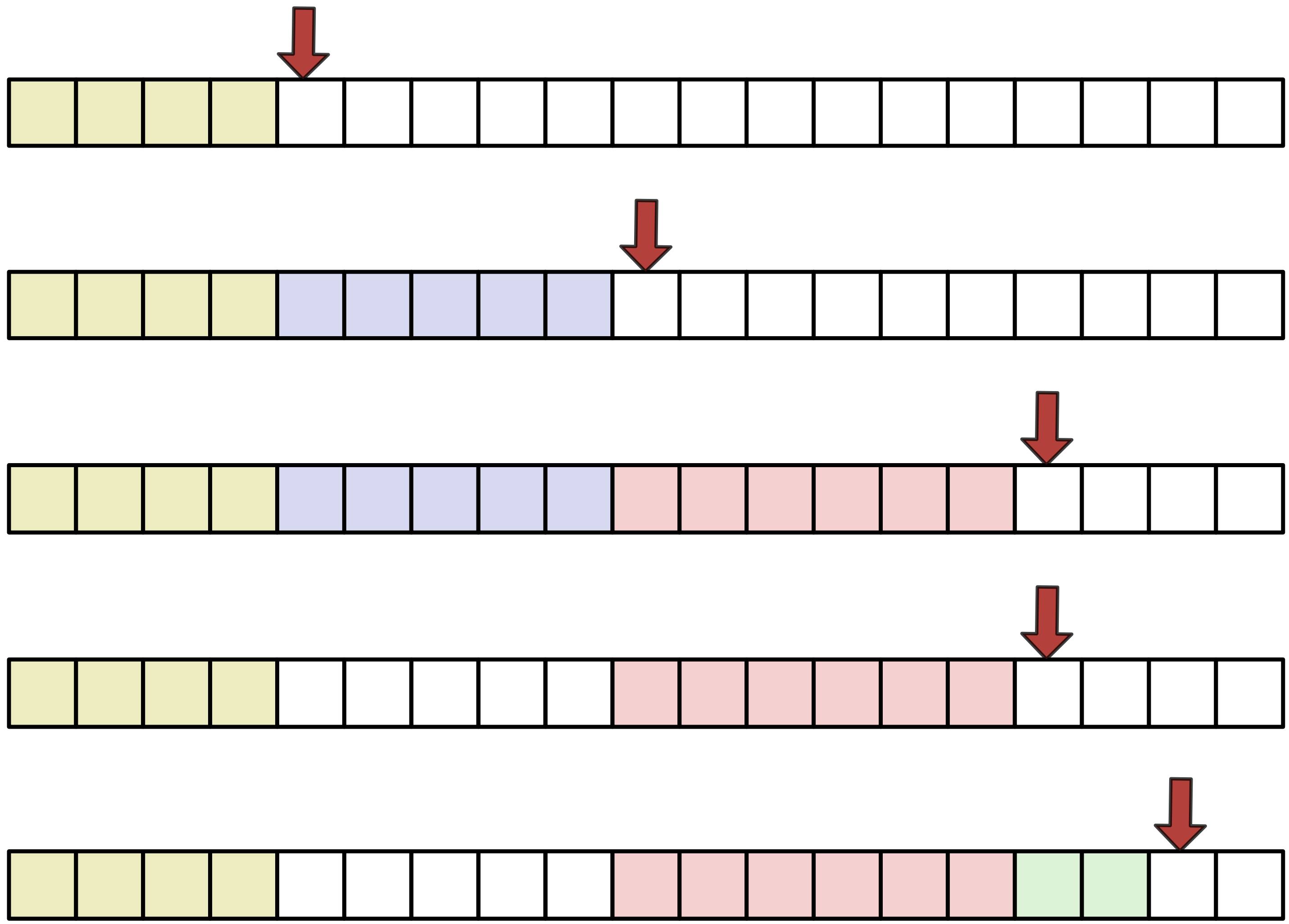
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

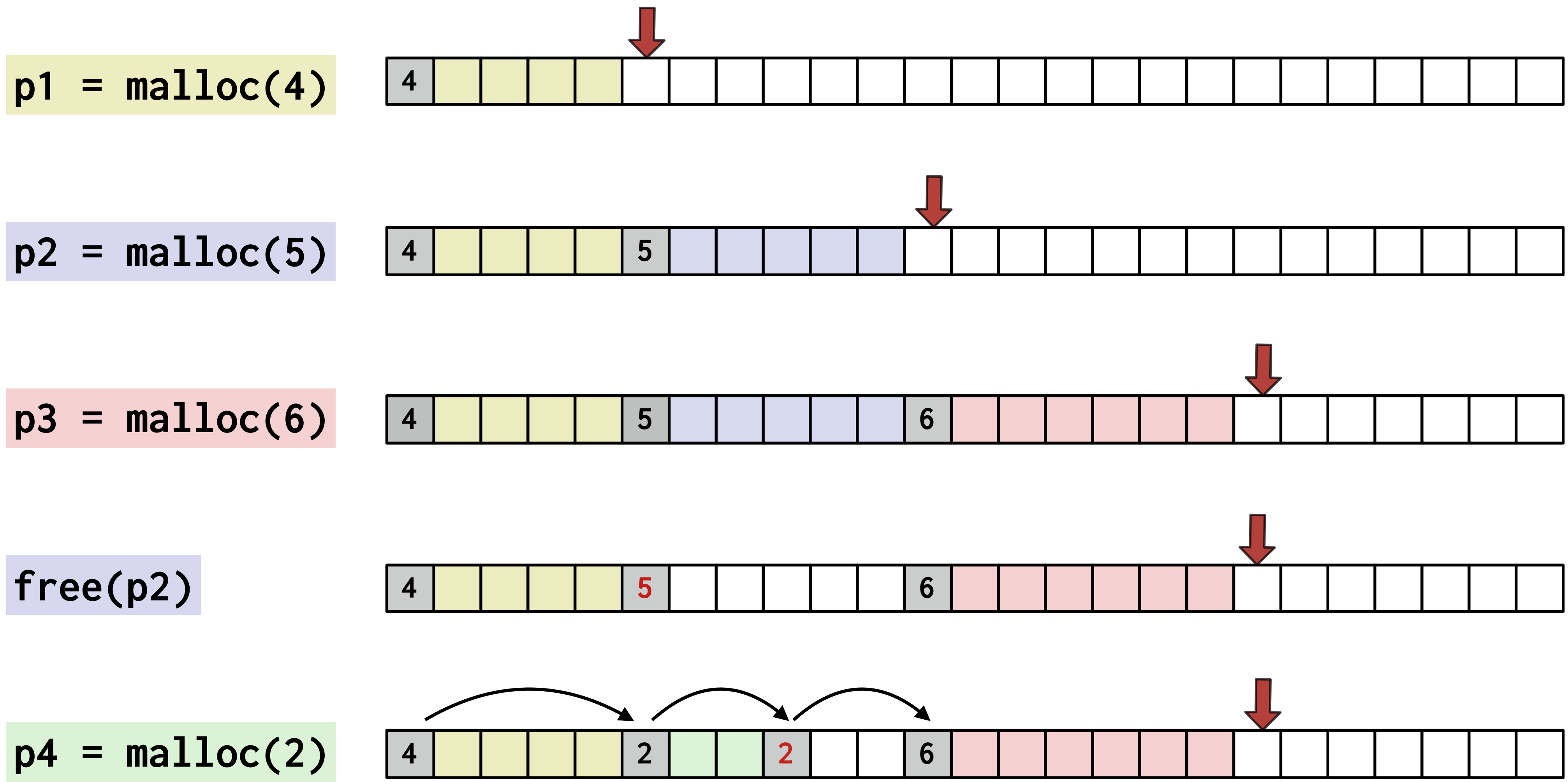
`p4 = malloc(2)`



Bump Memory Allocator

`malloc.c`

Pre-block header, implicit list



Header struct (with bitfield)

```
struct header {
    unsigned int size : 31;
    unsigned int status : 1;
};

enum { IN_USE = 0, FREE = 1};

void *malloc(size_t nbytes)
{
    if (!heap_max)
        heap_max = (char *)heap_next + TOTAL_HEAP_SIZE;
    nbytes = roundup(nbytes, 8);
    int total_needed = nbytes + sizeof(struct header);
    if ((char *)heap_next + total_needed > (char *)heap_max)
        return NULL;

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

Challenges for malloc client

1) Correct allocation (size)

2) Correct access to block (within bounds, not freed)

3) Correct free at correct time

What happens if you…

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

Challenges for malloc implementor

Tricky code (pointer math, typecasts)

Testing is hard (even more than usual)

Critical system component

correctness is non-negotiable, ideally also fast and compact

Survival strategies:

draw pictures

printf (you’ve earned it!!)

early tests on examples small enough to trace by hand if need be

Writing Good Systems Software

```
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

Open source era is fantastic! Some suggestions:

https://github.com/dwelch67/raspberrypi

https://www.musl-libc.org

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

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

Well-written software is easy for someone to read and understand.

Well-written software is easy for someone to read and understand.

Code that is easier to understand has fewer bugs.

Long comments != easy to read and understand.

Understand at the line, function, module, and structural levels.

Lesson: Imagine someone else has to fix a bug in your code: what should it look like make this easier?

Hint: be a section leader, you'll have to read student code and you’ll learn a lot!

Systems Code Is Terse But Unforgiving

Systems Code Is Terse But Unforgiving

Think about your PS/2 scan code reader: if any part of it is wrong, you won't read scan codes. It's only 20-30 lines of code!

The mailbox code you'll use for the frame buffer is ~10 lines of code: we once debugged it for 9 hours.

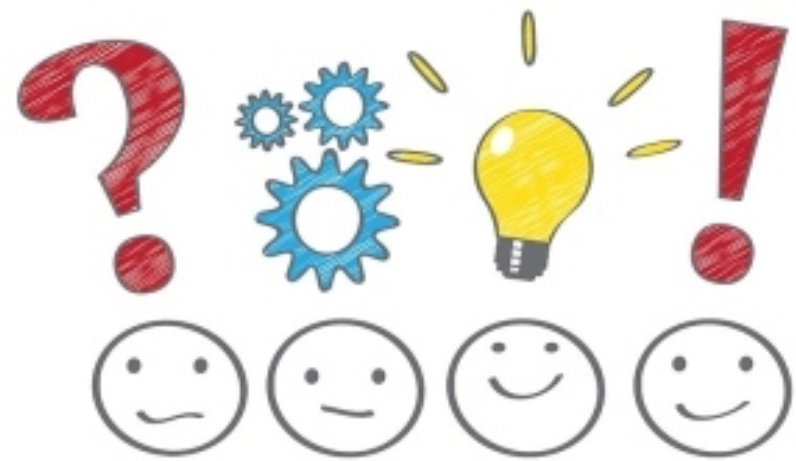
Lesson: if you know exactly what you have to do, it can take only minutes; throwing away and rewriting can often be *faster*. Sunk cost fallacy!

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 mai

ntain!

— 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 baby 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, 1-click build

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