Admin

- Halfway on our journey!
- Share/celebrate/commiserate

https://thefreethinkingmindblog.files.wordpress.com/2016/10/sisyphusgif.gif

Today: Steps toward C mastery

C Language, advanced edition, loose ends

Hallmarks of good software

Tuning your development process:

Pro-tips and best practices

Pointers, arrays, structures

Will we ever know enough???

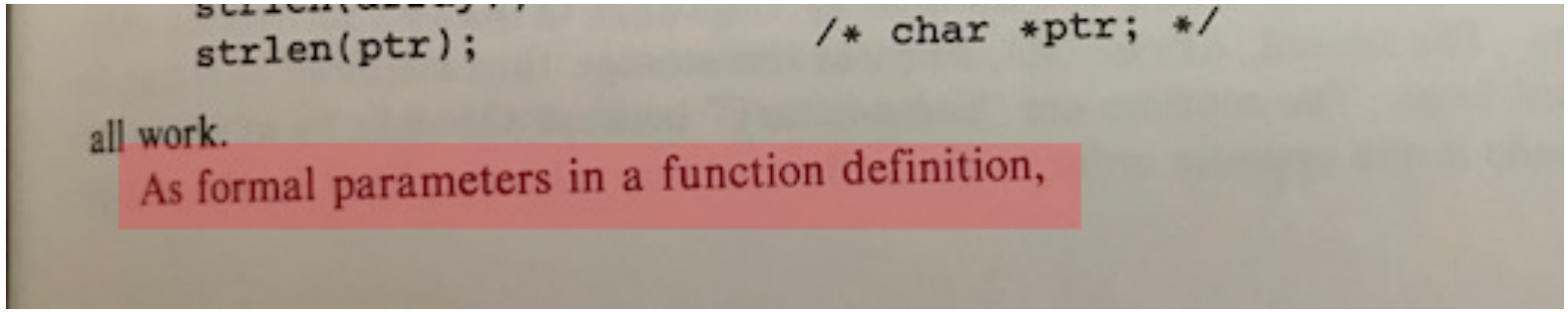
Pointers, address arithmetic exposed in C,
but not the only/best way to access memory

Array/structures provide abstraction
Improvement over raw address

Access to related data by index/offset/name
(underlying mechanism is base address + delta)

A most unfortunate page break

K&R bottom of page 99

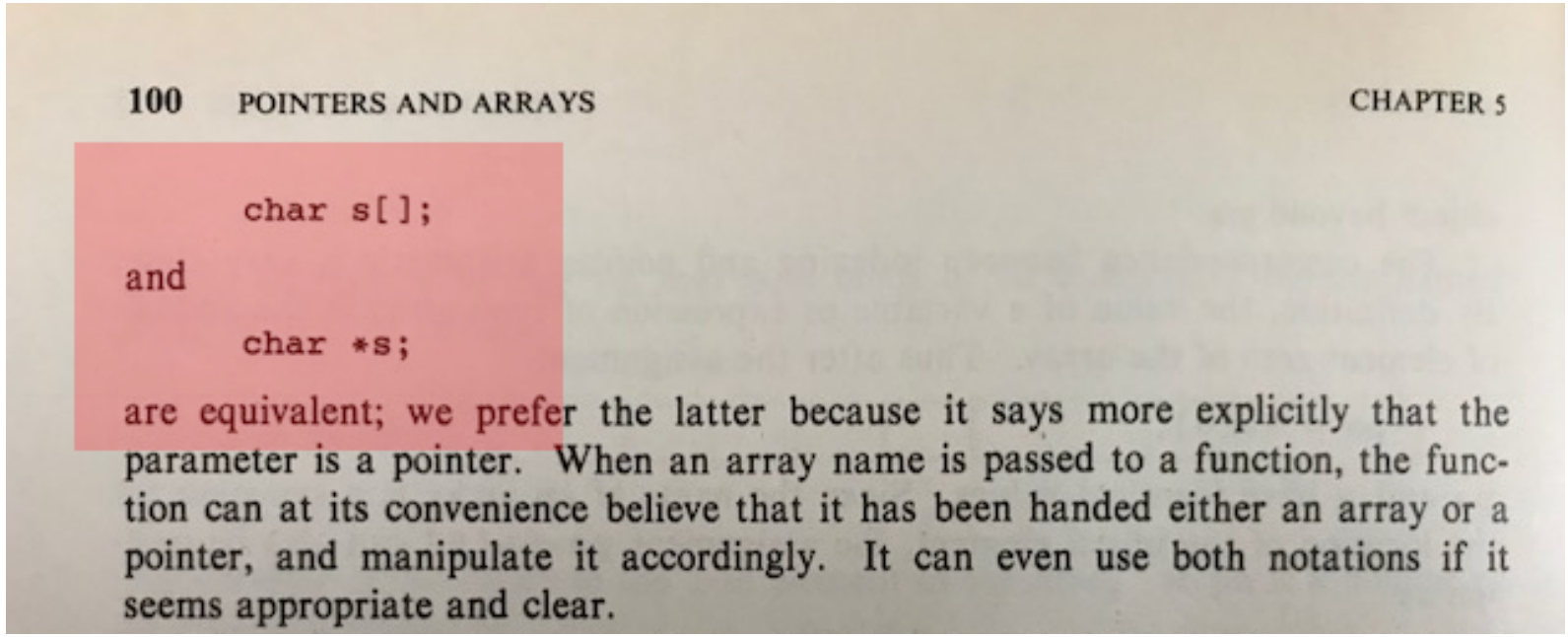


```
    strlen(ptr);                /* char *ptr; */
```

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 ....

```
void strings(void)  {
     // where/how much space is allocated for each string
     // how is each initialized?

    char a[10];
    char b[] = "dopey";
    const char *c = "happy";
    char *d = malloc(10);
    memcpy(d, "grumpy", 7);


    // which of these memory locations are valid to write?
    *a = 'A';
    *b = 'B';
    *c = 'C';
    *d = 'D';

    // what is printed?
    printf("%s %s %s %s\n", a, b, c, d);
```

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)?

Pointers and structs

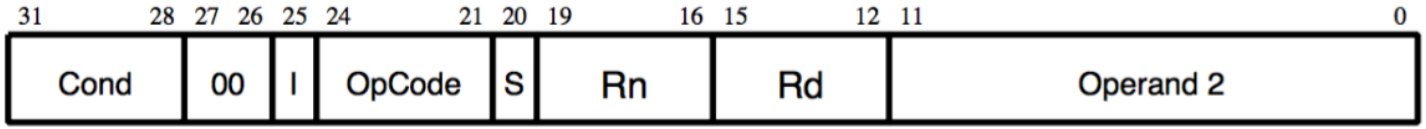
```
struct gpio {
  unsigned int fsel[6];
  unsigned int reservedA;
  unsigned int set[2];
  unsigned int reservedB;
  unsigned int clr[2];
  unsigned int reservedC;
  unsigned int lev[2];
};
```

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R

```
volatile struct gpio *gpio = (struct gpio *)0x20200000;

gpio->fsel[0] = ...
```

Structs and bitfields




```
struct insn  {
    uint32_t reg_op2:4;
    uint32_t one:1;
    uint32_t shift_op: 2;
    uint32_t shift: 5;
    uint32_t reg_dst:4;
    uint32_t reg_op1:4;
    uint32_t s:1;
    uint32_t opcode:4;
    uint32_t imm:1;
    uint32_t kind:2;
    uint32_t cond:4;
};

void change(struct insn *ptr) {
    ptr->reg_dst = 7;
}
```

Compiler generates this asm

```
change:
        ldrb    r3, [r0, #1]
        bic     r3, r3, #128
        orr     r3, r3, #112
        strb    r3, [r0, #1]
        bx      lr
```

Typecasts

C type system

Each variable/expression has type

Warns/disallows operations that don't respect type

But— also allows typecast to suppress/subvert ...

What does typecast actually do? Why is it allowed? Is it essential?

Is is sensible/necessary to:

- Cast to different bitwidth within same type family?
- Cast to add/remove qualifier (const, volatile)?
- Cast a pointer to different type of pointee?

Typecast is powerful but inherently unsafe

Rule: **work within type system!**

cast only when you absolutely must

Understanding the debugger

Indispensible, but sometimes a frustrating frenemy, too

Consider how debugger operates:

- Presents as if executing C source, but this is an illusion
- Executes asm, uses mapping from asm -> C source to orient
- Access to variables is similar, mapping symbol name -> storage location

When illusion breaks down, remember you have knowledge of **ground truth**!
Can trace asm, info registers, examine memory, dissect stack, ... make sense of it

Differences in software simulation vs hardware debugger

- No peripherals
- Contents of memory at program start

What you need to write good software

- Productive development process
- Effective testing
- Proficient debugging strategy
- Priority on good design/readability/maintainability

What is different about **systems software**?

Terse and unforgiving, details matter

All depend on it, bugs have consequences

Not enough to know what code does, but also how/why

```
void uart_init() {
    unsigned int ra;

    // Configure the UART
    PUT32(AUX_ENABLES, 1);
    PUT32(AUX_MU_IER_REG, 0);
    PUT32(AUX_MU_CNTL_REG, 0);
    PUT32(AUX_MU_LCR_REG, 3);
    PUT32(AUX_MU_MCR_REG, 0);
    PUT32(AUX_MU_IER_REG, 0);
    PUT32(AUX_MU_IIR_REG, 0xC6);
    PUT32(AUX_MU_BAUD_REG, 270);
    ra = GET32(GPFSEL1);
    ra &= ~(7 << 12);
    ra |= 2 << 12;
    ra &= ~(7 << 15);
    ra |= 2 << 15;
    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;
 }
```

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?

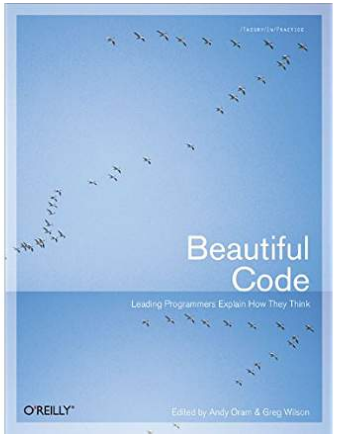
Open source era is fantastic!

https://github.com/dwelch67/raspberrypi

https://musl.libc.org

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

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



***Section lead CS106**: will read a lot of 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 **maintain**

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

> "There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."
> - C.A.R. Hoare

Development process

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

— Decompose problems, not programs

— 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.

Everything is happening for a reason, even if it doesn't seem so at first.

Engineering best practices

Test, test, test, and test some more

Start from a known working state, take small steps

Make things visible (printf, logic analyzer, gdb)

Methodical, systematic. Form hypotheses and perform experiments to confirm.

Fast prototyping, embrace automation, one-click build, source control, clean compile

Don't let bugs get you down, natural part of the work, relish the challenge -- you will learn something new!

Wellness important! ergonomics, healthy sleep/fuel, maintain perspective

Share your stories and pro-tips

Design, write, test, debug, …

Which parts of your approach/process are working well for you?

Which parts are not?