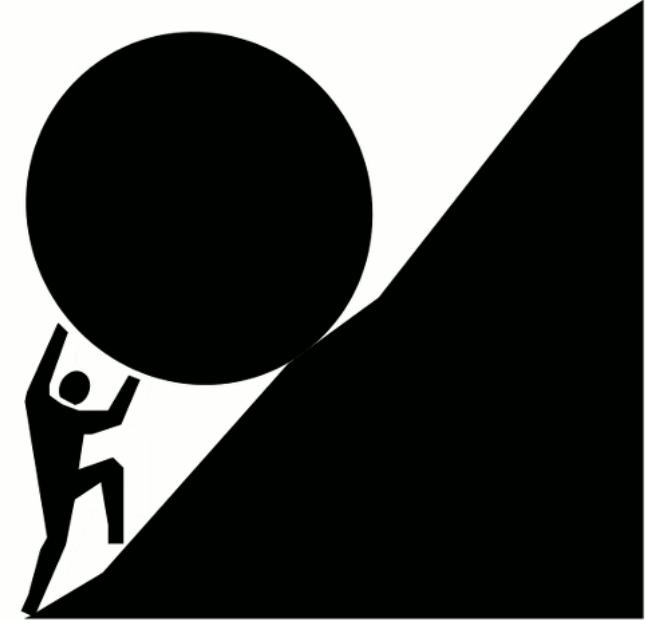


Admin

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



Today: Steps toward C mastery

C Language, advanced edition, loose ends

Hallmarks of good software

Tuning your development process:

Pro-tips and best practices



Revise & resubmit

Github issue posted if unit test does not pass

Read comment for log of output vs expected, diff

Priority P1 P2 P3 (P1 highest, essential to fix)

After you fix and test:

commit, tag assignN-retest, push

Weekly retest on repos with retest tag, update issues to show latest result

Fixing bugs puts you on path to earning full system bonus!

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

100 POINTERS AND ARRAYS

CHAPTER 5

```
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);
```

Segmentation fault

Access any memory by address — what could go wrong?

Q. For what reasons might a pointer be invalid?

Q. What is consequence of accessing invalid address
...in a hosted environment?
...in a bare-metal environment?



"The fault, dear Brutus, is not in our stars,
But in ourselves, that we are underlings."
[Julius Caesar \(I, ii, 140-141\)](#)



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?

Pointers and structs

9.7.4 Register List

Ref: [DI-H User Manual p.1093](#)

Module Name	Base Address
GPIO	0x02000000

```
struct gpio {  
    unsigned int cfg[4];  
    unsigned int data;  
    unsigned int drv[4];  
    unsigned int pull[2];  
};
```

Register Name	Offset	Description
PB_CFG0	0x0030	PB Configure Register 0
PB_CFG1	0x0034	PB Configure Register 1
PB_DAT	0x0040	PB Data Register
PB_DRV0	0x0044	PB Multi_Driving Register 0
PB_DRV1	0x0048	PB Multi_Driving Register 1
PB_PULL0	0x0054	PB Pull Register 0

```
volatile struct gpio *pb = (struct gpio *)0x02000030;
```

```
pb->cfg[0] = ...
```

Structs and bitfields

32-bit instruction format



```
struct insn {  
    uint32_t opcode: 7;  
    uint32_t reg_d: 5;  
    uint32_t funct3: 3;  
    uint32_t reg_s1: 5;  
    uint32_t reg_s2: 5;  
    uint32_t funct7: 7;  
};
```

Compiler generates this asm

```
void change(struct insn *ptr) {  
    ptr->reg_d = 10;  
}
```

change:

lw	a5,0(a0)
li	a4,-4096
addi	a4,a4,127
and	a5,a5,a4
ori	a5,a5,1280
sw	a5,0(a0)
ret	

Typecasts

C type system

Each variable/expression has type

Warns/rejects code that doesn't respect type

But can apply 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

Data alignment

"Natural" or "self" alignment

- 4-byte store/load at address that is multiple of 4
- 8-byte at multiple of 8

System optimized for natural alignment

Unaligned access may be allowed (perhaps at performance penalty) or disallowed (exception). Worst option allows, but behave wrong. (ARM did that, ack!)

Heap align to 8 (sizeof largest primitive) to avoid alignment woes

Invest in learning your tools

Editor

Efficient, direct, keep hands on keyboard

Shell

Navigation, history, repetition, customization, scripting

Build

make, assembler, preprocessor, compiler, linker

Learn to interpret error messages, identify stage, how to resolve

Version control

Infinite "undo", history, team collaboration

Navigating debugger

Indispensable, 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 know where to find
ground truth! Disassembly, info registers, examine memory, dissect stack, ...
to 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(void) {
    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); //((250,000,000/115200)/8)-1 = 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_PB8, GPIO_FN_ALT6);
    gpio_set_pullup(GPIO_PB8);
    gpio_set_function(GPIO_PB9, GPIO_FN_ALT6);
    gpio_set_pullup(GPIO_PB9);

    // set baud rate
    uart->regs.fcr = 1;
    uint32_t udiv = sys_clock_rate / (16 * baud);
    uart->regs.lcr |= LCR_DLAB;
    uart->regs.dll = udiv & 0xff;
    uart->regs.dlh = (udiv >> 8) & 0xff;
    uart->regs.lcr &= ~LCR_DLAB;

    // data-parity-stop = 8-N-1
    uint8_t data = 0b11, parity = 0b0, stop = 0b0;
    uint8_t settings = (parity << 3) | (stop << 2) | (data << 0);
    uart->regs.lcr = (uart->regs.lcr & ~0b1111) | settings;
}
```

The value of code reading

Consider:

Is it 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!

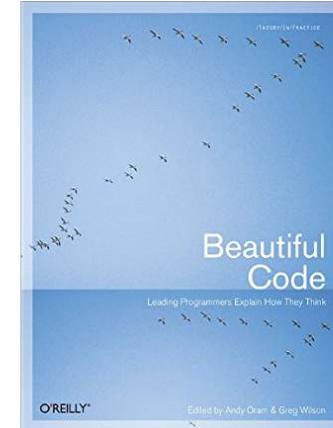
<https://github.com/dwelch67/raspberrypi>

<https://musl.libc.org>

<https://git.busybox.net/busybox/>

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

<https://opensource.apple.com>



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

Good style leads to good code

- Don't treat style as afterthought to fix after code works
- 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

Engineering best practices

Test, test, test, and test some more. Never delete a test, catch regressions

Start from a known working state, take small steps

Make things visible/observable (printf, logic analyzer, gdb)

Be methodical, systematic. Form hypotheses and perform experiments to confirm. Everything is happening for a reason, even if it doesn't seem so at first

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

Learn from one another!

Share war stories, observe, ask and answer questions

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

Which parts are not?

