



# Writing Better C

Common Library Usage, Error Handling, Formatting, and Readability



# Who and Why

- I sit on a lot of BSLE panels
  - I'm tired of seeing the same issues
- It makes it easier to not have to make 100x comments during code review
- Readability/Reliability of C
  - If another person has to think “what is that, does that actually work?” about C syntax, something has gone wrong
- So you don't get stuck on the small stuff



# What this talk is

- Brief History of C
- Examination of pitfalls common in C
- Overview of Barr-C Standards, and some explanations
- Intro to libraries that handle common tasks
- Some quick demos
- Tool usage to make writing good C easier
  - Clang, Clang-Tidy, and Clang Format
  - Very brief intro to this area, as it has been covered

If you have questions during the talk, throw up a hand. Group discussion of these topics is excellent.



# Disclaimers

No list in this talk is not all inclusive, as there are an almost infinite ways to write bad programs.

All opinions expressed are my own, or sourced from some authority (either a style guide or prominent figure).

My way isn't necessarily the right way, but it's usually not terrible.

Somebody please stop me if I start ranting about tabs vs spaces or something like that.

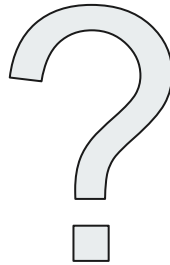


# What this talk is not

- A C tutorial (we may talk about pointers, but only cause they're cool)
- A resource to be copied and pasted when you need it
  - Understand the lesson and use it to do better, copy-pasta is a scourge in modern programming
- A definitive guide on how to write C correctly
  - Opinions expressed are my own
  - There isn't a "right" way to program, but there are definitely wrong ways
- A BSLE guide
  - Any topics covered here are standard programming tasks, and since the BSLE is a standard programming test, they may overlap
- A "gotcha" on anyone
  - All bad code on this demo is mine
  - I'm not picking on specific people
  - If you feel like I'm demoing a piece of your code:
    - Don't take it personally, my trainee code is rough too
    - Pay attention



**Any Questions Up Front?**





## C - A criminally short history

- Developed at Bell Labs in 1972 by Dennis Ritchie
  - Ritchie is one of the underappreciated geniuses of the computing revolution
  - Helped create Unix along side of Ken Thompson
  - C and Unix were designed to go together
- C is:
  - General Purpose (can be used for anything on anything)
  - Procedural (relies heavily on blocks and scope)
  - Standardized in several releases:
    - C99 is the BARR-C standard
    - C17 is the “Current” C standard, with C2X in the works
    - The Linux Kernel technically doesn’t have a standard, but a very comprehensive style guide
      - LKCS is far superior to BARR-C, but I just work here



## C - History cont.

- Because C is an older language, it has some quirks
  - More on this later
- Some supported operations are actively disliked by parts of the community (not inclusive):
  - ``goto`` (more on this later)
  - Variable Length Arrays - VLAs (sometimes called Flexible Array Members)
    - Basically any array where the size is determined at runtime
    - Some contention on this
    - Linus Torvalds hates them
      - “AND USING VLA'S IS ACTIVELY STUPID! It generates much more code, and much `_slower_` code (and more fragile code), than just using a fixed key size would have done.” - [LT](#)
    - I recommend not using them (more on this later)
  - Compound Literals
    - Again, some contention on this point
  - Being able to throw a ``case`` label any
  - Short Circuit operators
    - Don't use these





**Any Questions So Far?**





## Common Pitfalls - Overview

“The C language is like a carving knife: simple, sharp, and extremely useful in skilled hands. Like any sharp tool, C can injure people who don’t know how to handle it. This paper shows some of the ways C can injure the unwary, and how to avoid injury.”

- Andrew Koenig, ‘C Traps and Pitfalls’, 1989

(The paper, not the book, although the book is good too)



# Common Pitfalls - Overview

- 3 major trouble areas
  - Grammar
    - How the C source is literally written (think sentence structure)
  - Design
    - How the C program flows and fits together (think effectiveness of writing)
  - Style guide violations
    - Less concerning because it's a readability issue most of the time
    - I recommend picking a style and sticking to it
    - If you're having trouble trying to make your code style compliant, you probably did it wrong



# Common Pitfalls - Grammer

- Lexical Pitfalls
  - '=' not being equivalent to '=='
  - Specific Characters have Specific Meanings
    - "A programmer who substitutes one of these operators for the corresponding operator from the other class may be in for a surprise: the program may appear to work correctly after such an interchange but may actually be working only by coincidence." - Koenig
- Syntactic Pitfalls
  - Operator precedence
  - `switch` statement flow
  - "The Dangling `else` Problem"
- Semantic Pitfalls
  - Most of the problems I see in this area fall under this category
    - More to follow



# Common Pitfalls - Common Semantic Pitfalls

- Expression evaluation sequence
- Pointers are **NOT** Arrays
- Casting in general
- Something Koenig calls the “Eschew Synecdoche”
  - Basically confusing a pointer with the data to which it points
- The ``NULL`` pointer is not equivalent to a null string
  - You can technically dereference a null string, but cannot dereference a ``NULL`` pointer
  - ``char *str = ""`` or ``char *str = '\0``; create NULL strings
  - ``char *str = NULL``; sets the actual value of the pointer to 0 (`#define NULL 0`)
  - Valid: ``if (str == (char *) 0) ...``
  - Invalid: ``if (strcmp (str, (char *) 0) == 0) ...`` because ``str`` gets dereferenced, but `NULL` cannot be dereferenced
- Integer overflows
  - ``uint32 t overflow = 0xffffffff + 1``

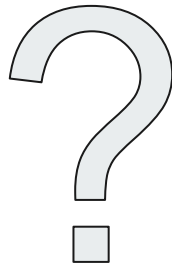


# Common Pitfalls - Design

- Strange Control Flow
  - Putting ``while`s` and such in a ``switch`` statement
  - Multiple ``return`` statements
- Incredibly long functions
  - “Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.” - LKCS
  - “All reasonable effort shall be taken to keep the length of each function limited to one printed page, or a maximum of 100 lines.” - BARR-C 6.2.a
- Complex nested cases
  - Clang-Tidy yells at you if you do this (nested complexity warning)
  - LKCS sums this up well:
    - “... if you need more than 3 levels of indentation, you’re screwed anyway, and should fix your program.”



**Any Questions So Far?**





## BARR-C - Intro

“Barr Group's Embedded C Coding Standard was developed from the ground up to minimize bugs in firmware, by focusing on practical rules that keep bugs out--while also improving the maintainability and portability of embedded software.”

- Latest is BARR-C:2018
- Much stricter on the small stuff than LKCS
- BSLE/CSD standard (usually)
- Designed to eliminate much of the Lexical, Syntactic, and Semantic Pitfalls
  - Produces extremely verbose code that can get annoying
  - Nothing is more annoying than getting 100s of suggestions on a code review
- Very particular on naming of functions and variables
- Produces very “proper” C

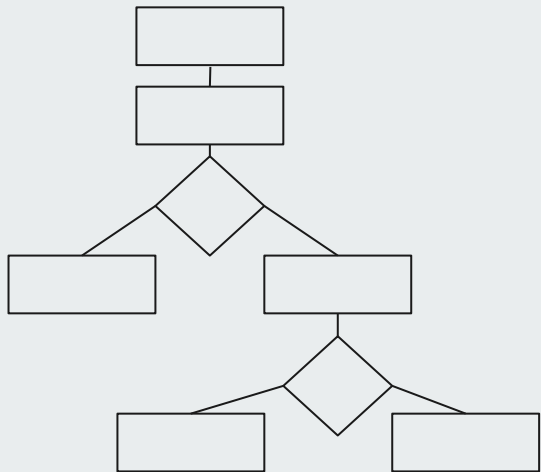




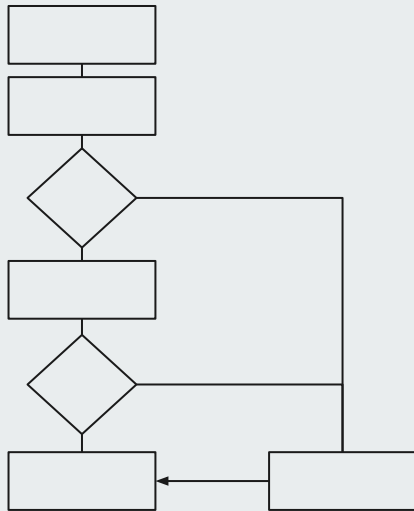
## BARR-C - Highlights

- 80 character lines, 100 line functions
- Pretty much everything is separated by one space (assignments, keywords, `if`s, etc)
- Braces are **always** used on code blocks, and **always** appear on their own line (1.3.a/b)
- Parenthesis are **always** specified when order of operations could be an issue, or when logical operators are used.
- Goto Usage:
  - "1.7.c. It is a preferred practice to avoid all use of the goto keyword. If goto is used it shall only jump to a label declared later in the same or an enclosing block."
- Avoid the use of `continue`
- Tab (0x09) should never be used (only spaces), and each "indentation" should be 4 spaces
  - vs LKCS 8 spaces per indent level
- "6.2.c. It is a preferred practice that all functions shall have just one exit point and it shall be via a return at the bottom of the function."
- There's a bunch of other rules, check before you write C
  - Or, use clang-format with [this](#) repo

## BARR-C - Examples - One Return Statement w/ gotos



## Very “Branchy” - Too many Return



## Consolidated Return

```

4 bool
5 branching_even_div_by_3(int num)
6 {
7     if (0 != num % 2)
8     {
9         return false;
10    }
11    else
12    {
13        if (0 == num % 3)
14        {
15            return true;
16        }
17        else
18        {
19            return false;
20        }
21    }
22 }
23
24 bool
25 consolidated_even_div_by_3(int num)
26 {
27     bool valid = true;
28
29     if (0 != num % 2)
30     {
31         valid = false;
32         goto END;
33     }
34
35     if (0 != num % 3)
36     {
37         valid = false;
38         goto END;
39     }
40
41     ;
42
43     return (valid);
44 }

```

Fun Fact: GCC compiled both options to be logically equivalent to the right side.



**Any Questions So Far?**





# Common Libraries - Common Tasks

- Endianness Conversion
  - Endian.h
- Network/File IO
  - Various depending on what you're doing
- String/Memory comparison
  - String.h
- Memory Allocation and associated checks
  - stdlib.h



# Common Libraries - Common Issues

- file or socket IO
  - Not reading all data
  - Not closing file descriptors
- Incorrect/manual endian conversions
- Incorrect/insecure/manual string comparison
  - Trusting user input - don't do this
- Memory Management Issues
  - Not checking allocations
  - Not properly freeing memory



## A Quick Talk about Endianness

- Endianness is the order in which bytes of a variable are stored in memory
  - Big Endian - Most Significant Bit in Lowest Memory Address
    - More intuitive (until you get used to LE)
    - Used in Networked Packed bytes (usually)
  - Little Endian - Least Significant Bit in Lowest Memory Address
    - Type agnostic
    - Used on most common CPUs (Intel x86/x86-64)

0x41424344	0x1000	0x1001	0x1002	0x1003
BE	0x41	0x42	0x43	0x44
LE	0x44	0x43	0x42	0x41



## Endianness - Cont.

The advantages of Little Endian are:

- It's easy to read the value in a variety of type sizes. For example, the variable `A = 0x13` in 64-bit value in memory at the address `B` will be `1300 0000 0000 0000`. `A` will always be read as `19` regardless of using 8, 16, 32, 64-bit reads. By contrast, in Big Endian we have to know in which size we have written the value to read it correctly.
- It's easy to cast the value to a smaller type like from `int16_t` to `int8_t` since `int8_t` is the byte at the beginning of `int16_t`.
- Easily to do mathematical computations "because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write."

Some main advantages of Big Endian are

- We can always test whether the number is positive or negative by looking at the byte at offset zero, so it's easy to do a comparison.
- The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

# Endianness - Cont.

The main library for handling binary conversions in C is Endian.h

- Do not manually swap bytes (bad)
- Do not write weird bitshift operations (worse)
- Do not ignore endianness of variables
  - this will probably get you failed

## JUST USE ENDIAN FUNCTIONS

Examples are pulled directly from “man endian”

```
$ ./a.out
x.u32 = 0x44332211
htole32(x.u32) = 0x44332211
htobe32(x.u32) = 0x11223344
```

## Program source

```
#include <endian.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    union {
        uint32_t u32;
        uint8_t arr[4];
    } x;

    x.arr[0] = 0x11;    /* Lowest-address byte */
    x.arr[1] = 0x22;
    x.arr[2] = 0x33;
    x.arr[3] = 0x44;    /* Highest-address byte */

    printf("x.u32 = %#x\n", x.u32);
    printf("htole32(x.u32) = %#x\n", htole32(x.u32));
    printf("htobe32(x.u32) = %#x\n", htobe32(x.u32));

    exit(EXIT_SUCCESS);
}
```





## Demo

The main demo we have today is called netdog

- Basic functionality of netcat
- Technically could be a real utility
- Connects to a remote server and:
  - Either sends input from a file
    - Will then try to change to interactive if '-i' is specified
  - Sends/Recv's from interactive sessions

Demonstrates:

- Arg parsing
- Socket operations
- Basic Threading
- Str->Num conversion
- Setting DEBUG macros
- File/Network IO
- General BARR-C
- Some VSCode

Integrations

- Built based on LT
- Paris's TED Talk



# Questions?

