Programming can be fun, so can cryptography; however they should not be combined.

– Kreitzberg and Shneiderman
It's only a clever hack if you're the one who wrote it

Essential

Hating Other People’s Code

@ThePracticalDev
Anti-Patterns:
- Code compiles with warnings
- Warnings are turned off or over-ridden
- Insufficient warning level set
- Language safety features over-ridden

Make sure the compiler understands what you meant
- A warning means the compiler might not do what you think
  - Your particular language use might be “undefined”
- A warning might mean you’re doing something that’s likely a bug
  - It might be valid C code, but should be avoided
- Don’t over-ride features designed for safe language use
The C Language Doesn’t Always Play Nice

- Defined, but potentially dangerous
  - if (a = b) { ... }       // a is modified
  - while (x > 0); {x = x-1;} // infinite loop

- Undefined or unspecified ➔ dangerous
  - You might think you know what these do …
    … but it varies from system to system
  - int *p = NULL;  x = *p;    // null pointer dereference
  - int b;   c = b;         // uninitialized variable
  - int x[10]; …  b = x[10];    // access past end of array
  - x = (i++) + a[i];       // when is i incremented?
Language Use Guidelines & Tools

- **MISRA C, C++**
  - Guidelines for critical systems in C (e.g., no malloc)
  - Portability, avoiding high risk features, best practices

- **CERT Secure C, C++, Java**
  - Rules to reduce security risks (e.g., buffer overflows)
  - Includes list of which tools check which rules

- **Static analysis tools**
  - More than compiler warnings (e.g., strong type warnings)
  - Many tools, both commercial and free. Start by going far past “–Wall” on gcc

- **Dynamic Analysis tools**
  - Executes the program with checks (e.g., memory array bounds)
  - Again, many tools. Start by looking at Valgrind tool suite
Rule 13.4 The result of an assignment operator should not be *used*

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32]

**Category**  Advisory
**Analysis**   Decidable, Single Translation Unit

**Amplification**
This rule applies even if the expression containing the assignment operator is not evaluated.

**Rationale**
The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement making it more difficult to avoid the undefined behaviour covered by Rule 13.2.

**Example**

```c
x = y;         /* Compliant */
a[ x ] = a[ x = y ]; /* Non-compliant - the value of x = y is used */

/*
 * Non-compliant - value of bool_var = false is used but
 * bool_var == false was probably intended
 */
if ( bool_var = false )
{
}
```

[MISRA C-2012 Guidelines; Fair Use]
Let the Language Help!

- Use enum instead of int
  - `enum color {black, white, red}; // avoids bad values`

- Use const instead of #define
  - `const uint64_t x = 1; // helps with type checking`
    `uint64_t y = x << 40; // avoids 32-bit overflow bug`

- Use inline instead of #define
  - If it’s too big to inline, the call overhead doesn’t matter
  - Many compilers inline automatically even without keyword

- Use typedef with static analysis
  - `typedef uint32_t feet; typedef uint32_t meters;`
    `feet   x = 15;`
    `meters y = x; // feet to meters assignment error`

- Use stdint.h for portable types
  - `int32_t` is 32-bit integer, `uint16_t` is 16-bit unsigned, etc.
2012 Open Source Coverity Scan

- **Sample size:** 68 million lines of open source code
  - Control flow issues: 3,464 errors
  - Null pointer dereferences: 2,724
  - Resource leaks: 2,544
  - Integer handling issues: 2,512
  - Memory – corruptions: 2,264
  - Memory – illegal accesses: 1,693
  - Error handling issues: 1,432
  - Uninitialized variables: 1,374
  - Uninitialized members: 918

- **Notes:**
  - Warning density 0.69 per 1,000 lines of code
  - Most open source tends to be non-critical code
  - Many of these projects have previously fixed bugs from previous scans

Deviations & Legacy Code

- Use deviations from rules with care
  - Use “pragma” deviations sparingly; comment what/why

- What about legacy code that generates *lots* of warnings?
  - Strategy 1: fix one module at a time
    - Useful if you are refactoring/re-engineering the code
    - Sometimes might need to keep warnings off for 3rd party headers
  - Strategy 2: turn on one warning at a time
    - Useful if you have to keep a large codebase more or less in synch
  - Strategy 3: start over from scratch
    - If the code is bad enough this is more efficient ... if business conditions permit
Desirable language capabilities:
- Type safety and strong typing (e.g., pointers aren’t ints)
- Memory safety (e.g., bounds on arrays)
- Robust static analysis (language & tool support)
- In general, no surprises

Spark Ada as a safety critical language
- Formally defined language; verifiable programs
  - The language doesn’t have ambiguities or undefined behaviors
- You can prove that a program is correct
  - E.g., can prove absence of: array index out of range, division by zero
  - (In practice, this makes you clean up your code until proof succeeds)
- Key idea: design by contract
  - Preconditions, post-conditions, side effects are defined

procedure Increment (X : in out Counter_Type) with Global => null,
  Depends => (X => X),
  Pre => X < Counter_Type'Last, 
  Post => X = X'Old + 1;
Language Style Best Practices

- Adopt a safe coding style (or a safe language)
  - MISRA C & CERT C are good starting points
  - Specify a static analysis tool and config settings
    - To degree practical, let machines find the style problems
  - When static analysis is set up, add dynamic analysis

- The point of good style is to avoid bugs
  - Let the compiler find many bugs automatically
  - Reduce chance of compiler mistaking your intention

- Coding style pitfalls:
  - “The code passes tests, so warnings don’t matter”
  - Real bugs lost in a huge mass of warnings
  - Making it too easy to deviate from style rules
Ugh, I hate reading your code.

I know, I know.

It's like you ran OCR on a photo of a Scrabble board from a game where JavaScript reserved words counted for triple points.

It looks like someone transcribed a naval weather forecast while woodpeckers hammered their shift keys, then randomly indented it.

It's like an e.e. cummings poem written using only the usernames a website suggests when the one you want is taken.

This looks like the output of a Markov bot that's been fed bus timetables from a city where the buses crash constantly.

Whatever, it runs fine for now.

So does a burning bus.