Electrical & Computer ENGINEERING

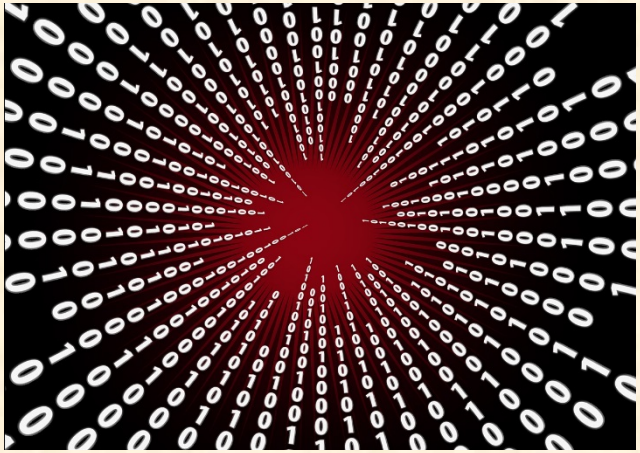
18-642:
Code Style for Compilers

9/25/2017

Carnegie Mellon University

Coding Style: Language Use

- **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`

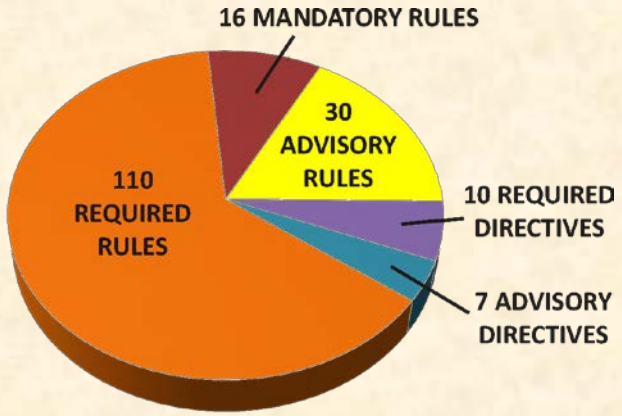
- **Undefined or unspecified ➔ dangerous**
  - You might thing you know what these do …

… 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



MISRA C:2012 with Security

- **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

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

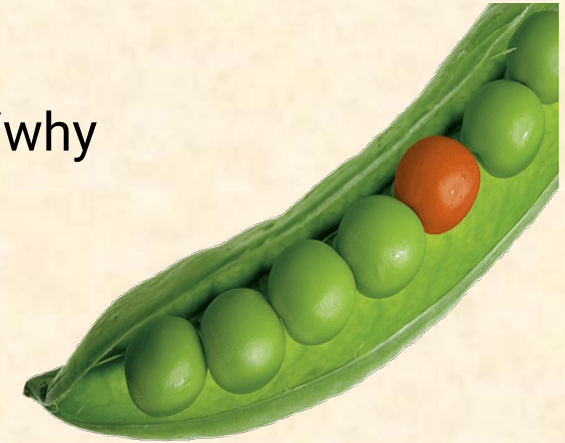
- **Sample size: 68 million lines of open source code**
  - Control flow issues: 3,464 errors
  - Null pointer dereferences: 2,724
  - Resource leaks: 2,544
  - Integer handling issues: 2,512
  - Memory – corruptions : 2,264
  - Memory – illegal accesses: 1,693
  - Error handling issues: 1,432
  - Uninitialized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

http://www.embedded.com/electronics-blogs/break-points/4415338/Coverity-Scan-2012?cid=Newsletter+-+Whats+New+on+Embedded.com

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

Language Style Best Practices

- **Adopt a safe coding style**
  - 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