# Best practices: the theoretical and practical underpinnings of writing code that's less bad

Axel Naumann, CERN EP-SFT

Openlab Summer Student Lectures, 2024-07-22

# How To Write Bad Code

Axel Naumann, CERN PH-SFT

Openlab Summer Student Lectures, 2024-07-22

# Bugs!

Axel Naumann, CERN PH-SFT

Openlab Summer Student Lectures, 2024-07-22

www.dan-dare.org

# <*prelude*>

# Why Axel?

# Why Axel?

- Because I can write expert-level bad code.

# Why Axel?

- 20 years of ROOT development: *the* tool for every physicist's analysis
- Member of the ISO C++ committee (e.g. std::variant)
- Experience from introducing a static analysis tool at CERN
- Chair of the CERN Open Source Program Office

# Disclaimer

- I am not your best practices superhero
- Focus on C++
  - experience, usage, need

blogs.elconfidencial.com

# Why you?

- Because you have an impact!
  - your code is part of XYZ, or on top of XYZ, or replaces XYZ
  - you have colleagues, we listen to people with ideas!
  - I see lots of coding in your future!

# Practices

- More than one dev or more than one user: need to agree on “how”
- CERN has decades of piles of code, lessons learned:

  1. be reasonable!
  2. but enforce!
  3. fix rules early, adapt new ones slowly

# Best Practices

- Don’t follow today’s best Best Practices blindly
  - it will be ridiculed in ten years anyway
- But defining best practices publicly helps new contributors integrate quickly
- CERN OSPO (= open-source experts from almost everywhere at CERN) currently compiles recommendations at https://ospo.docs.cern.ch/

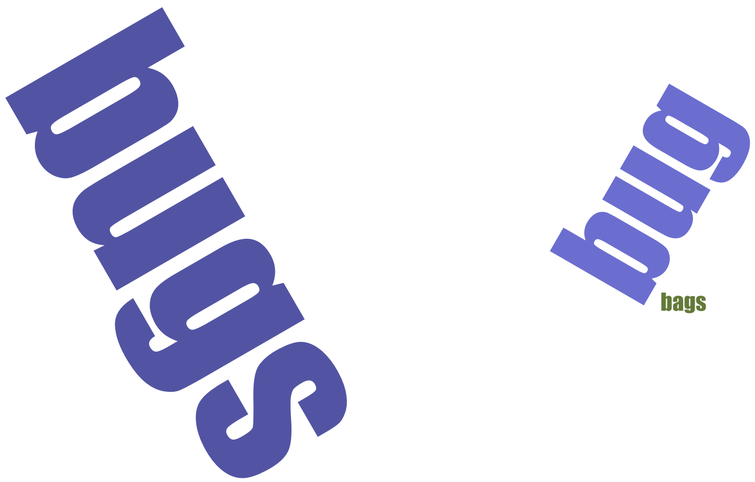
# Best Practices Context

- Collaborative development, across cultures and generations
- Software maintenance and software use over decades
- High-throughput, efficient scientific computing code on >1M cores, 24/7
- Libraries-as-a-tool (vs computer scientists who know what they are doing)
- Legal, security, policy, etc
- Resource cost: maintenance + conceptual burden, "how many positions is this worth"?

# Motivation

- Simpler, consistent read
  - improved communication with fellow coders
  - less ambiguities means more correct code
- Less bugs; better maintenance
- Best practices win against experimental coding

# Word Cloud



*</prelude>*

# Menu Du Jour

- Language
- Coding convention
- Interface convention
- Change management
- Multi-platform support
- Tests: code-correctness, functionality, static analysis, performance
- Due diligence, security
- AI

# Language Choice

# Language Features

- Some languages are better for a given job than others
  - high performance (C++!)
  - smaller problem, from conception to completion (Python!)
  - re-use available (library) code instead of coding yourself, e.g. networking (plenty), filesystem (bash!)
  - resource management, inherent security (Rust!)

# Generation of Safe Languages

- Most key languages designed decades ago
  - We've learned a lot since then, but backward compatibility prevents us from applying lessons learned to these languages
- Java => Kotlin
- JS => TS
- C++ => ???
- Or "based on the shoulders": Rust, Julia,...

# Available Tooling

- High-level versus low-level (web versus ASIC)
- Rule of thumb: the closer to silicon you go the better tools you will want (debugger, perf, tests)
- Pick the right language given available and needed tooling!

# You are not alone

- “Community” knowledge, now and future: no Haskell, please
- Your knowledge: no COBOL, please
- Practicality: no assembler, please
- Interfacing with relevant existing code: no Go, please

# Coding Convention

# Coding Convention

- What is this?

```
func(val);
```

# Coding Convention

- It’s a counter-example!

```
func(val);
```

- func: Member function? Data member / function pointer? Some global function pulled in from header?

- val: local variable declared 100 lines up in the same function? Or member? Or enum constant? And where can I find it’s declaration?

# Coding Convention

```
fFunc(fgVal);
```

- It’s ROOT - you can tell from the names!
- It’s a function call
- fFunc is a member - so it’s a function pointer!
- fgVal is a static data member; must be in same class (or base)

# Coding Convention

- Obvious case of improved clarity
- For APIs, user friendly:
  - get_track(), getTrack(), GetTrack() - or Track()?
  - IDEs can help - but not when *reading* code!
- Almost all projects employ it

# Coding Convention

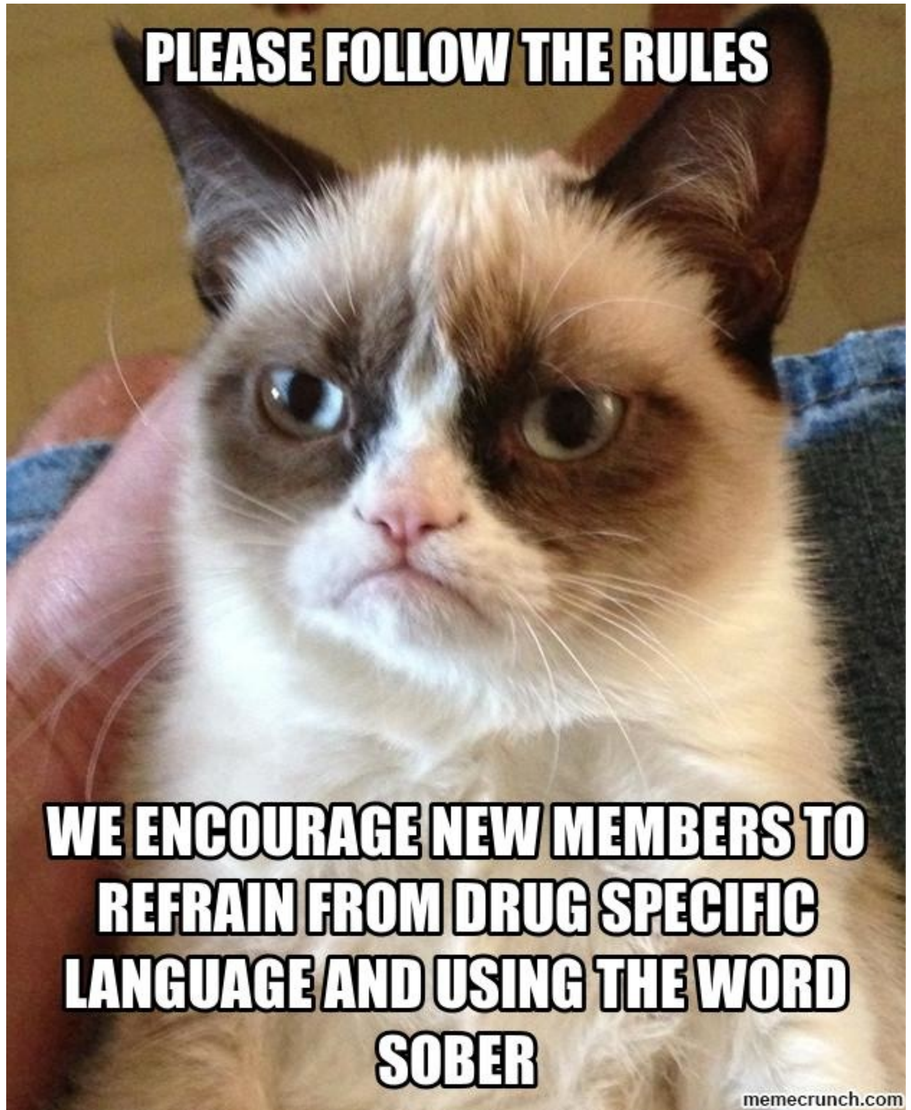
- Typical current examples for C++:
  - Joint Strike Fighter Air Vehicle C++ Coding Standards
  - MISRA C++
- Both absurd for reasonable environments
- Both have very reasonable ingredients: pick yours!

# Coding Convention

- Enforcing needs checkers
- Non-trivial; checker must understand C++: what is a function, what is a member etc
- Recommendation: clang-format to the rescue!

# Interface Conventions

# Interface Conventions




memecrunch.com

# Interface Conventions

- Consistency - we know that already
- Safe code through good APIs!
  - unique_ptr / shared_ptr instead of Type* where ownership is managed; never use “new Type()”, “delete var”
  - document also parameter pre- and post-condition:
    ```
    // arg1 must be != 0; arg2 will contain…
    ```

# Interface Conventions

- Maintain common idioms throughout API; example C++ std library:
  - iterators; functor; make_XYZ; allocator etc
- Don’t screw with your users
  - if interface looks like A, do **not** change it to do B even if it’s better for you. Change the interface in a backward-**incompatible** way instead.

# Interface Conventions

```
template <typename T>
requires std::integral<T> || std::floating_point<T>
constexpr double Average(std::vector<T> const &vec) {
```

- C++ Concepts, since C++20
  - Define interface expectations in code, compiler checkable!
- CERN is starting to collect experience with this
  - New libraries should consider the use of concepts to clarify expectations with user code and generate better error messages

# Concurrency Support

Distinguish

- code starts threads to compute faster (multithreaded)

from

- code supports being called concurrently (thread-safe)

from

- code operates on multiple values (vectorization support / SIMD)

At CERN, in scientific high-throughput code, we care about *all* of these

# Thread Safety

- Different types
  - function can be used on same object in multiple, concurrent threads without side-effects [*thread safe*]
  - function can be used on different objects in multiple, concurrent threads without side-effects (no unsync'ed statics) [*conditionally safe*]
  - must be locked when accessed through multiple threads [*not thread safe*]

1

```
int current = 0;
int add1() {++current;}
int getSum() {return current;}
```

2

```
class Sum {
  mutable int current = 0;
  int add1() const;
  int getSum() const;
};
```

3

```
class Sum {
  int current = 0;
  int add1();
  int getSum();
};
```

4

```
class Sum {
  int current = 0;
  int add1();
  int getSum() const;
};
```

5

```
int current = 0;
std::mutex mtx_current;
int add1() {
  const std::lock_guard<std::mutex> lock(mtx_current);
  ++current;
}
int getSum() {
  const std::lock_guard<std::mutex> lock(mtx_current);
  return current;
}
```

# Threading Support

- All kinds need to be clearly documented, thread-safe part of API needs to be visible
- Common contract nowadays:
  - const API means it’s conditionally safe: no unlocked mutables! no caches! no hidden state changes!
  - no static variables (without locks)! State is passed as arguments

# Threading Support

- Thus threading support is to some extent interface convention - plus good design enabling it
- C++ and concurrency continues to evolve
  - constexpr / consteval functions / std::executor / coroutines might play a bigger role soon
  - exposing to 256 threads changes requirements (Amdahl’s law!) + style: writing to memory: data layout conventions! (cache lines + false sharing)

# Interface Convention + Threading Support

- Automated checking (beyond coding convention) almost impossible
  - requires design work / understanding of the interfaces
  - concepts can help
- Employ change management instead!

# Change Management

moneygeek.com.au

# Change Management

- Monitor changes by a second pair of eyes: two brains are better than one, especially if one brain is biased
- Prevents some bugs from creeping in
- Also exposes code, new features to additional / backup developers
- Exposes changes to larger horizon: we all think of changes in different contexts
- Can be pre- or post-publication

# Change Management: Pre-publication

- Package owner merges changes
- Formalized patch review
- Pair programming

# Change Management

- Post-publication
  - commit review by package owner
- Post-review risks stability of HEAD of "main" / dev-branch
  - still reasonable for small changes
  - here, too: be pragmatic, not dogmatic

# Lessons at CERN

- If it works, it will break
  - new OS version, new compiler version, new language version
- Only way out: embrace change
  - put procedures in place to survive change
  - benefit from change instead of trying to mitigate it: more expressive code, improved tooling, tasty for new developers, influence future instead of crawling behind

# Multi-Platform Support

# Multi-Platform Support

- Problems:
  - memory: memory layout, alignment, cost of data transfers / locality (cache, code size etc)
  - OS API
  - compilers with limited language support
- Experienced developers will get a feel of which language constructs are causing problems

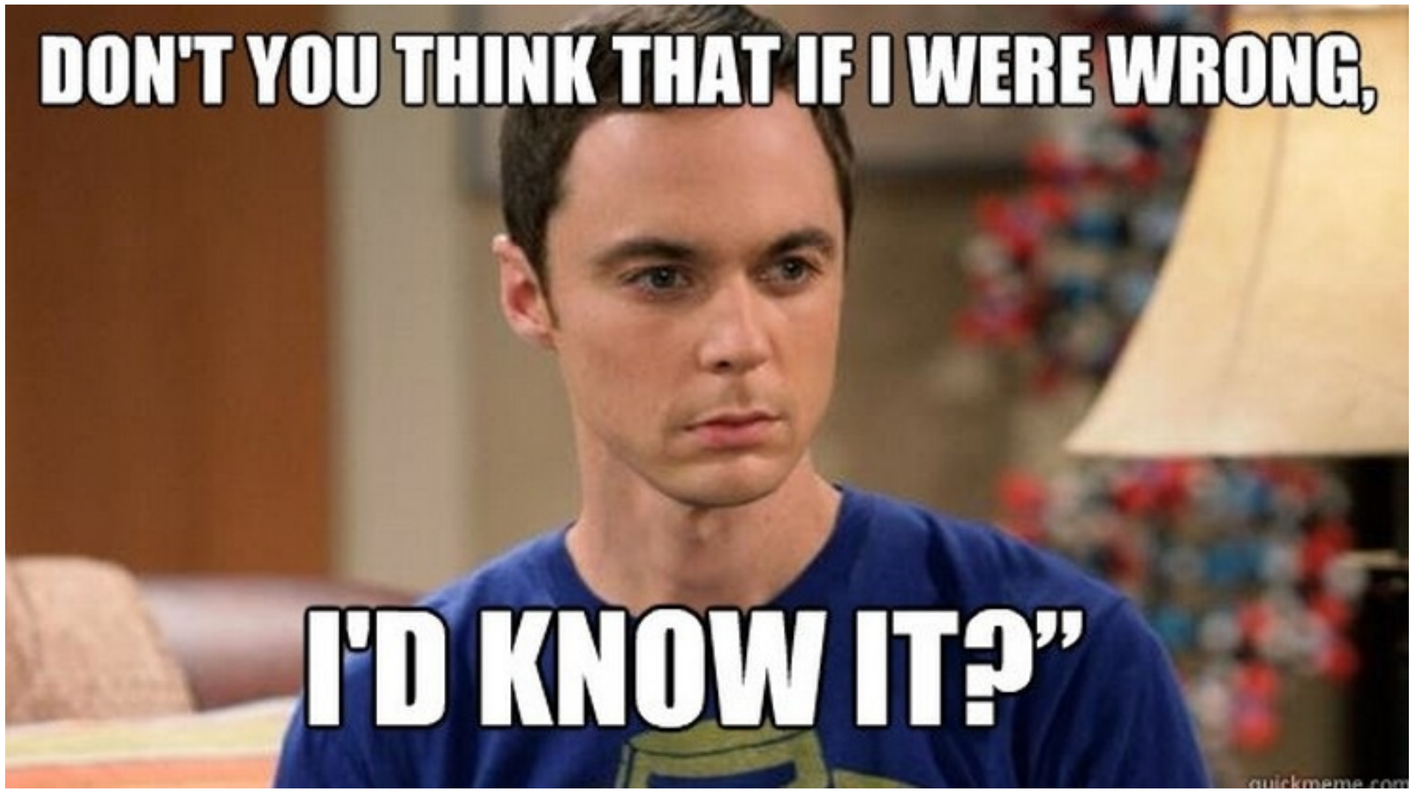
# Multi-Platform Support

- Advantages
  - increases general robustness
  - easier to follow architecture changes: Apple ARM!
  - will x86_64 be the instruction set of 2030?
  - more compilers = more opinions on code, more warnings: that’s a good thing!

# Multi-Platform Support

- Checking by building on many platforms, regularly
  - Code correctness tests!

# Tests


DON'T YOU THINK THAT IF I WERE WRONG,
I'D KNOW IT?”
quickmeme.com

# Code Correctness Tests

- Large matrix of builds
  - build on all supported platforms, with all supported configurations
- Ideally after every change to pinpoint culprits
- Current common grounds: `main` (formerly known as `master`) passes tests
  - possibly with dev branch, CI merges into `main` after validation
  - welcoming to contributors, enables user feedback on latest changes

# Code Correctness Tests

- Run build (incremental or full)
  - check for errors versus platform
  - also check for warnings!
- Run tests
- Build snapshot binaries
  - continuous delivery, for instance for bug fix verification

# Code Correctness Tests

- Needs automation
- Typical tools: ~~Jenkins; Bamboo~~; Github Actions / GitLab CI/CD and others
  - initiate build on all required machines
  - collect output; filter errors, warnings
  - report (web, email) versus code revision and author

# Functionality Tests

- “Does my software actually work?”
  - unit tests; regression tests; integration tests
  - rules when to write a test
  - coverage analysis
  - testing libraries: cppunit / GoogleTest / …
- Needs automation!

# Topical Tests

- Memory error checkers - use after free / before initialization
  - e.g. valgrind
- Thread error checkers
  - e.g. hellgrind, Vtunes

# Static Analysis

- Analyzes source code without running it; creating branch graph to follow possible `if` etc combinations

- Finds use after delete; impossible `if` conditions; memory errors etc

# Static Analysis

- Several tools out there, for instance
  - basic checker: compiler warnings!
  - clang static analysis: clang-tidy
  - ~~as part of CI: GitHub CodeQL~~ Houston, we have a problem.
- Differ in set of bugs checked; tracing capabilities (through function calls etc); user interface; **false positive rate**

# CERN Lessons

- Static analysis **cannot** be replaced by test suite: it tests the things that “never happen”
- Improves code stability
- Developers feel “watched”: improves overall code quality

# Performance Test

- Changes can deteriorate performance:
  - takes more CPU cycles to get an answer
  - takes more RAM
  - takes more I/O operations
  - takes more disk space
- Criteria vary depending on product

# Performance Test

- Usually part of release baking
- Better yet: automate
- Problem: which changes are intentional?
- Tools vary with criteria; e.g. cgroups; massif; CDash
- CI/CD integration currently still lacking!

# Due Diligence, Security

# Due Diligence

- Code written at CERN (and published) should not contain swear words, passwords, sensitive data, etc.
- Code published from CERN should satisfy minimal quality requirements: best practices!
- And then the legal aspect of "did these people really write this code all by themselves"?!
- No entity at CERN does that for you: you are responsible!

# Security

- Code written at CERN (and published) should not contain security issues.
- Code published from CERN should satisfy minimal quality requirements: best practices!
- No entity at CERN does that for you: you are responsible!

# Due Diligence + Security

- Just like scientists know they cannot copy chapters of someone else's publication, and cannot "tweak measurements": we expect that people writing code behave professionally

- Professionalism includes knowing basic security and due diligence traps and how to avoid them

# Learning Due Diligence + Security

- Due diligence: https://ospo.docs.cern.ch/recommendations/due-diligence/ is a start
- Security: excellent trainings by our CERN Computer Security friends
  - and the openlab summer student lecture by Stefan Lüders!

Photo by Austris Augusts on Unsplash

# AI

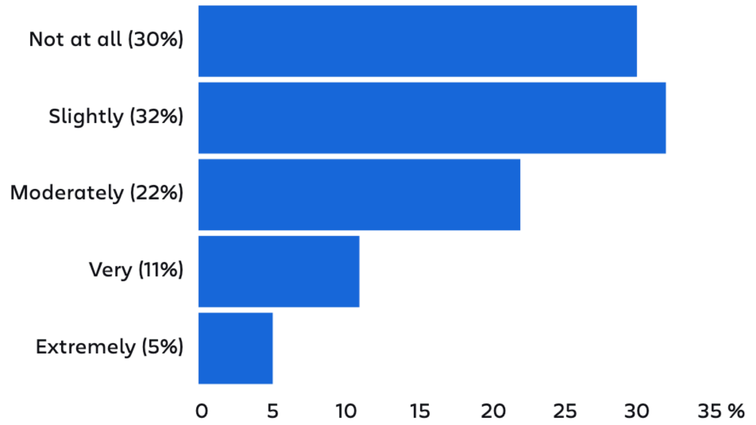
# ML

- Yes, we call it "Machine Learning" because we don't need to sell stuff. And here I'm really just referring to LLMs for programming.
- We have little expertise on the consequences for code. Just like the rest of the world. But we'll get there.
  - E.g. copyright (of generated code, of models, of weighs) still not clear.
- Interesting study: https://symflower.com/en/company/blog/2024/dev-quality-eval-v0.5.0-deepseek-v2-coder-and-claude-3.5-sonnet-beat-gpt-4o-for-cost-effectiveness-in-code-generation/

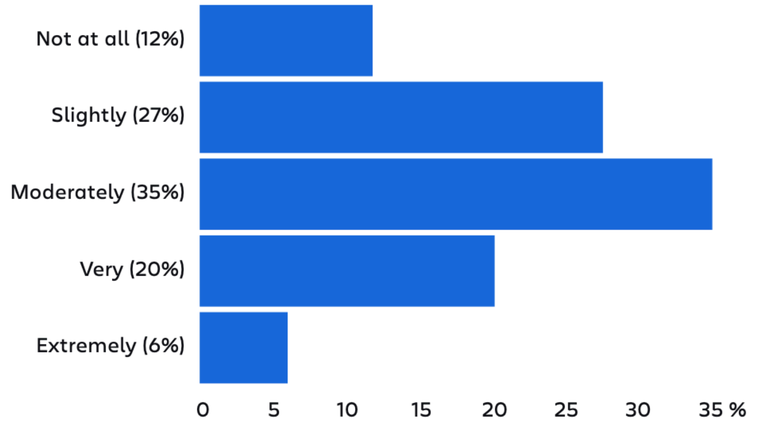
# Atlassian Report on AI

- State of developer experience report 2024
- **All** of management thinks developers are more productive thanks to AI!

**How much AI tools are improving developer productivity today**




**How much AI tools will improve developer productivity within the next two years**



# LLMs instead of us?

- From that study:
  - Only 10% of models have just 80% of generated code compilable! Overall only 58% of generated code compiles. Some models reach 0%... a programmer that cannot write code that compiles 🥳
  - Newer languages (go) are a real challenge: less training data
- So *for now* you're safe. But the more code you write, the more training material exists!

100%

# Current Challenges

- Massive multi-threading
- Data-oriented programming
- Evolution of C++ standards
- Move every tool into the FOSS world - thanks, clang!
- ML! (Yes yes, "AI".)
- Complexity, layering, indirections, "modularity" (looking at you, docker!)

# Complexity Challenge

- A simple next.js web app:

```
up to date, audited 609 packages in 1s

150 packages are looking for funding
```

- Best-practice *that*!

# Conclusion (1/4)

- Good software development is an art by itself
  - complex; many aspects; need to juggle many tools and often conflicting goals
- Not a reason to avoid it, but needs brain energy
- Need to find compromise between coding productivity and control

# Conclusion (2/4)

- Using the right tools pays off:
  - 1 hour more work for one dev can mean 10 minutes saved for 10k users *each*

```
$ python3 -c 'print(10.*1E4/60/24/5, "weeks!")'
13.888888888888889 weeks!
```

  - users will trust your software more

# Conclusion (3/4)

- Help your team define missing procedures
- Review procedures, review tools, review effectiveness
  - cover all aspects: runtime + performance tests, static analysis - none of that is optional
  - automate, reduce developers’ pain: increases acceptance
- Please invent new things: GitHub action for AI code reviews learning from bug reports and past commits?!

# Conclusion (4/4)

- Go out and write good code!

axel@cern.ch