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

Axel Naumann, CERN EP-SFT

Openlab Summer Student Lectures, 2018-07-31

How To Write Bad Code

Axel Naumann, CERN EP-SFT
Openlab Summer Student Lectures, 2018-07-31

Bugs!

Axel Naumann, CERN EP-SFT
Openlab Summer Student Lectures, 2018-07-31

<prelude>

Why Axel?

Why Axel?

- Because I can write expert-level bad code.

Why Axel?

- 15 years of ROOT development: *the* tool for every physicist's analysis
- Member of the C++ committee
- Introduced static analysis tool at CERN

Disclaimer

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

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

Best Practices

- Don’t follow today’s best Best Practices blindly
  - it will be ridiculed in a year anyway
- But defining best practices publicly helps new contributors integrate quickly
- See e.g. Bjarne Stroustrup @ CppCon http://sched.co/3vVp

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

</prelude>

Menu Du Jour

- Language choice
- Coding convention
- Interface jargon
- Change management
- Multi-platform support
- Tests: code-correctness, functionality, static analysis, performance

Language Choice

Language Choice

Language Choice

Language Features

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

Available Tooling

- High-level versus low-level (ASICs versus web)
- Rule of thumb: lower language means performance more relevant means you want better tools (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 other code: no Go, please (and in principle, no C++, 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 its 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++

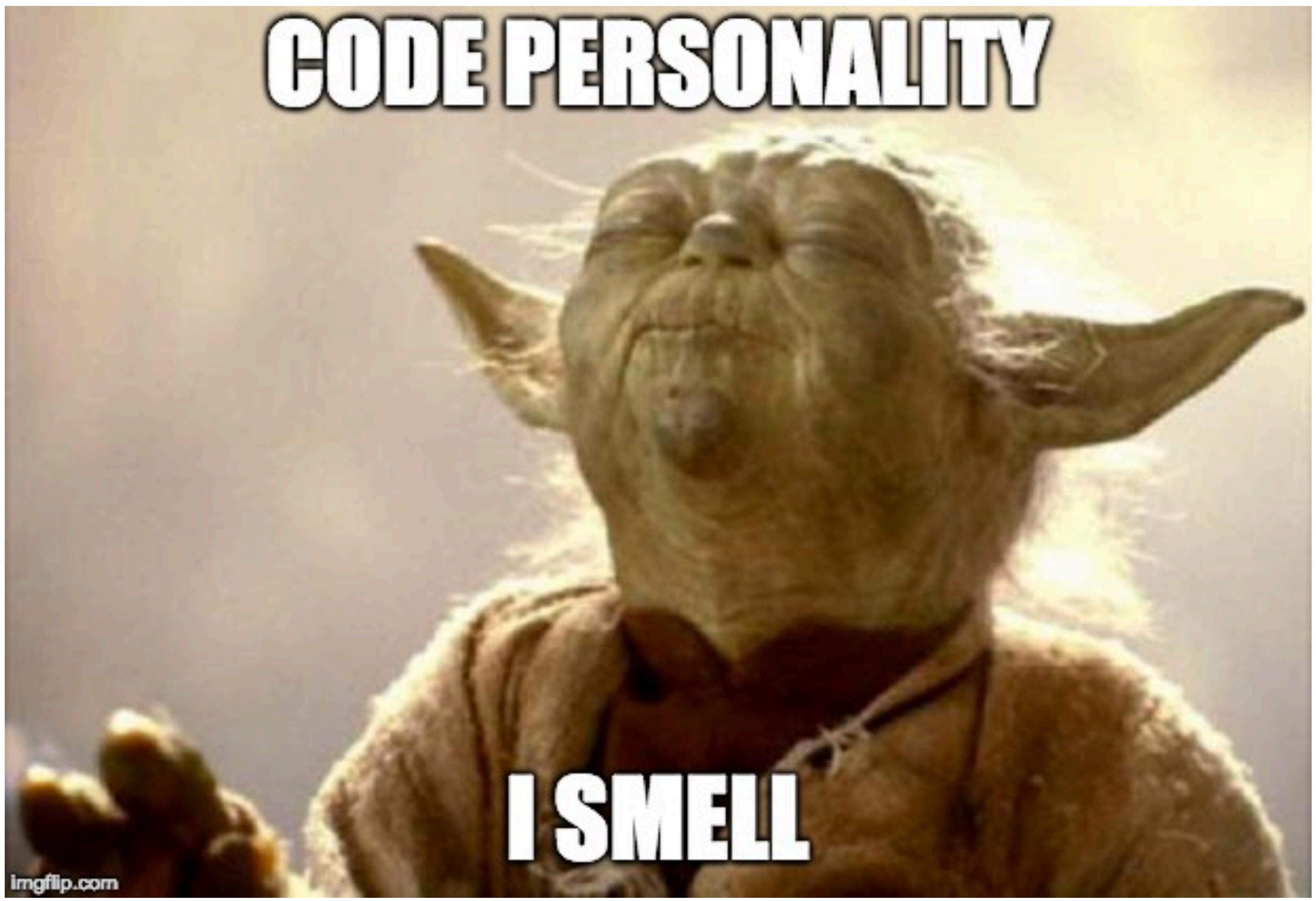
- Both absurd for reasonable environments
- Both have very reasonable ingredients: pick yours!

Coding Convention

- Enforcing needs checkers
- Non-trivial; checker must understand C++: what is a function, what is a member etc
- Many C-coding convention checkers (indentation!), few C++, even less open source
  - clang-format is becoming a reasonable alternative

Interface Jargon

Interface Jargon



Interface Jargon

- 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 Jargon

- 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, don’t make it do B even if it’s better for you. Change the interface in a backward-incompatible way instead.

Interface Jargon

- Don’t screw with your users
  - if interface looks like A, don’t make it do B even if it’s better for you. Change the interface in a backward-incompatible way instead.

Concurrency Support

Distinguish

- code starts threads to compute faster (multithreaded)

from

- code operates on multiple values at the same time (SIMD / vectorized)

from

- code can be called concurrently (thread safe)

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*]

Not Thread Safe

```
struct Counter {
  static int m_count;
  void incr() { ++m_count; }
};
```

Conditionally Thread Safe

```
struct Counter {
  int m_count;
  void incr() { ++m_count; }
};
```

Thread Safe

```
struct Counter {
  std::atomic<int> m_count;
  void incr() { ++m_count; }
};
```

Thread Safe
By Pure Functions

```
struct Counter {
  int incr(int oldval) { ++oldval; }
};
```

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 jargon (plus good design)
- This is work in progress; has changed rather recently
    - expect further changes; constexpr / pure functions might play a bigger role soon
    - exposing to >64 threads might again change requirements (Amdahl's law!) + style

Interface Jargon + Threading Support

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

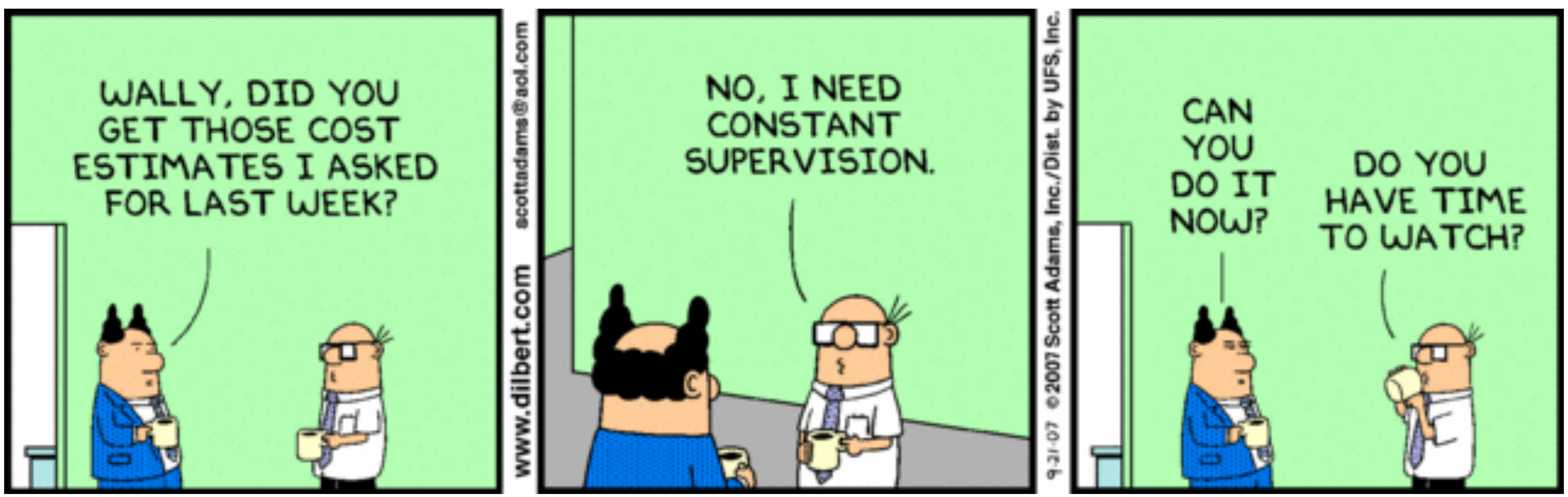
Change Management

CC BY-NC 4.0 pngimg.com

Change Management

- Monitor by a second pair of eyes: two brains are better than one, especially if one brain is biased
- Avoids 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

Change Management



Change Management

- Can be pre- or post-publication
- Pre-publication
    - package tags / tag collector (dying concept)
    - package owner merges changes
    - formalized patch review
    - pair programming

Change Management

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

Always-HEAD

- New attempt to always require newest code, e.g. Abseil: no versions, just "today"
- Requires excellent control of new code
  - must never break HEAD
- Difficult orchestration of many packages
  - always build everything

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
    - streamline it instead of mitigating it

Multi-Platform Support

Multi-Platform Support

- Problems:
  - big- versus little-endian
  - OS API
  - compilers with limited language support
- Experienced developers will get a feel of which language constructs are causing problems

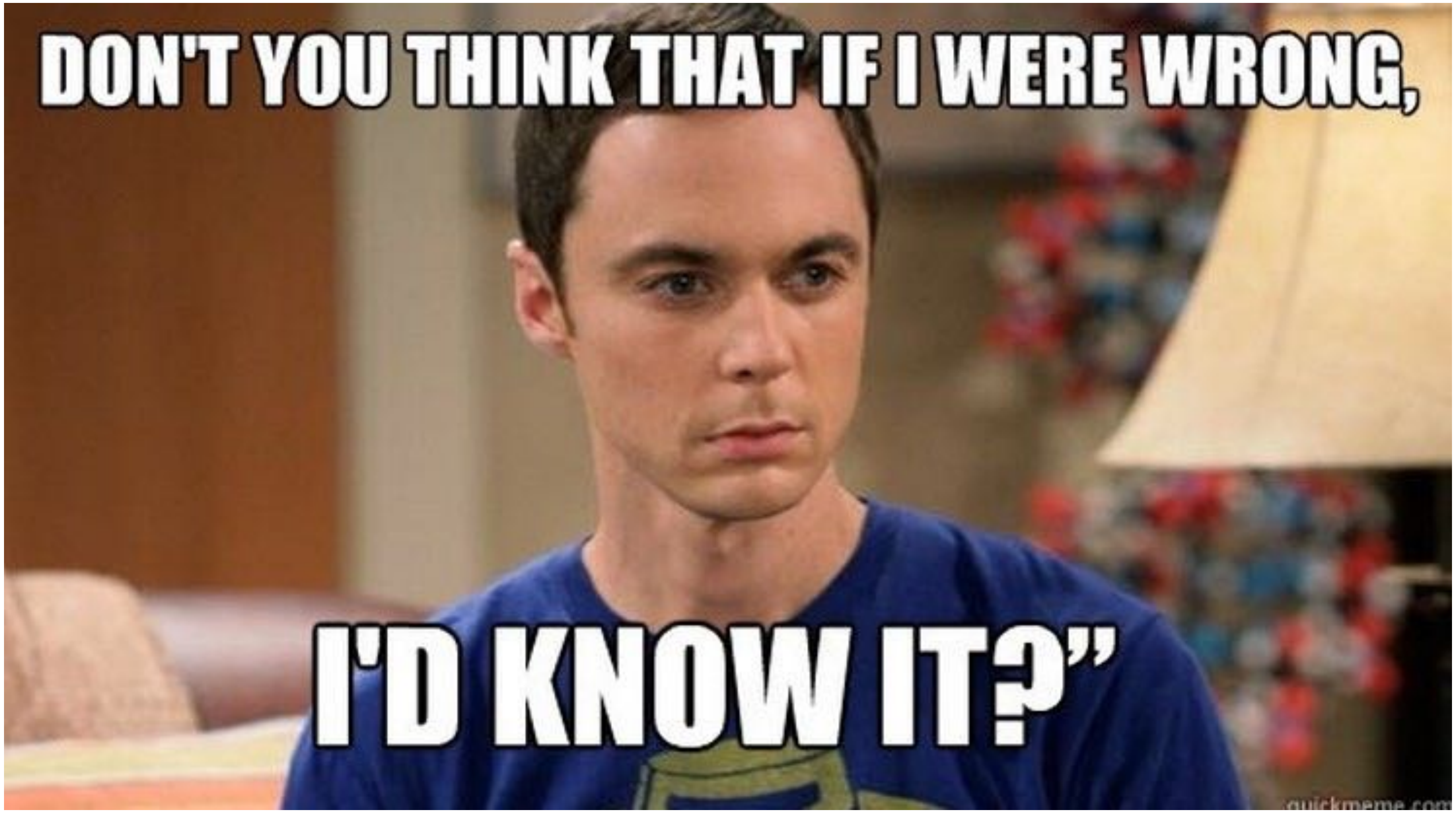
Multi-Platform Support

- Advantages
  - increases general robustness
  - easier to follow architecture changes
  - 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?"

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: the HEAD works
  - possibly with dev branch, CI merges into master after validation

Code Correctness Tests

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

Code Correctness Tests

- Needs automation
- Typical tools: Jenkins; Bamboo; TeamCity; BuildBot and others
  - schedule and initiate build on all required machines
  - collect output; filter errors, warnings
  - report (web, email) versus code revision

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

```
int func(char* buf) {
 strcat(buf, “<default>”);
 int pos;
 if (buf[0] != ‘<’) {
   std::cout << “Number between 0 and 8:\n”;
   std::cin >> pos;
 }
 buf[pos] = 0;
 if (!buf) return -1;
 return pos;
}
```

- What’s wrong in this snippet?

Voluntary “Homework”

- Do a code review, simulating a static analysis tool
- Compile it here: https://godbolt.org/g/7UAWCt
- Send your optimal version of
  int func(char* buf)
  to axel@cern.ch and I’ll send you mine
    - let’s review one another’s version
    - by Friday 23:59

Static Analysis

- Several tools out there, for instance
  - basic checker: compiler warnings!
  - clang static analysis
  - Coverity
- 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


ATOMIC
Rooster

100%

CC BY-NC 4.0 pngimg.com

Current Challenges

- Massive multi-threading
- Data-oriented programming
- C++11 and up
- Move every tool into the FOSS world

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.*10E3/60/24/5, "weeks!")'
    13.88888888888889 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
    - automatize, reduce developers' pain to increase acceptance

Conclusion (4/4)

- Go out and write good code!