ALICE

# Static C++ Code Analysis

Sandro Wenzel (CERN)

## Disclaimer

- This is not a technology survey and naturally incomplete ... Will mention
  - ***What are use cases?***
  - ***Example Frameworks***
  - ***A major (usability) shortcoming***

- Material here based on
  - Static Analysis Suite (SAS)
  - AliceO2 codechecker

## What is Static Code Analysis? Why would one want to use it?

- A tool for **checking, analysing, and potentially changing** (C++) code without executing it
  - i.e. before / during compilation

- To check if your code is
  - **(Semantically) correct -- according to user specification --** beyond being just valid C++ text
  - Following **coding guidelines**
  - Following **modern practices** (C++11/14/17 in favor over C++98)
  - Free of performance bottlenecks
  - Thread safe
  - ….

## Cooked up example : Just to show what is possible

```
class AbstractWorker {
 public:
  virtual void doWork() = 0;
1 float mX;
  float mInvX; // the inverse of X
};


class Worker : public AbstractWorker {
 public:
  void doWork() override {
2          static int state = 0;
      // do some work
3          float y = 1./mX /*...*/;
  }
};
```

Imagine you have a class … and you want to make sure that

1 Member variables follow **naming convention**

2 That implementations of function doWork() in classes deriving from AbstractWorker do **not contain plain static variables** … because they are used in a **threaded context**

3 That no one is **dividing by 'mX'** because that is **expensive** and I have in any case cached the inverse of it.

Then, you are most likely in need to write a custom static analysis (check)

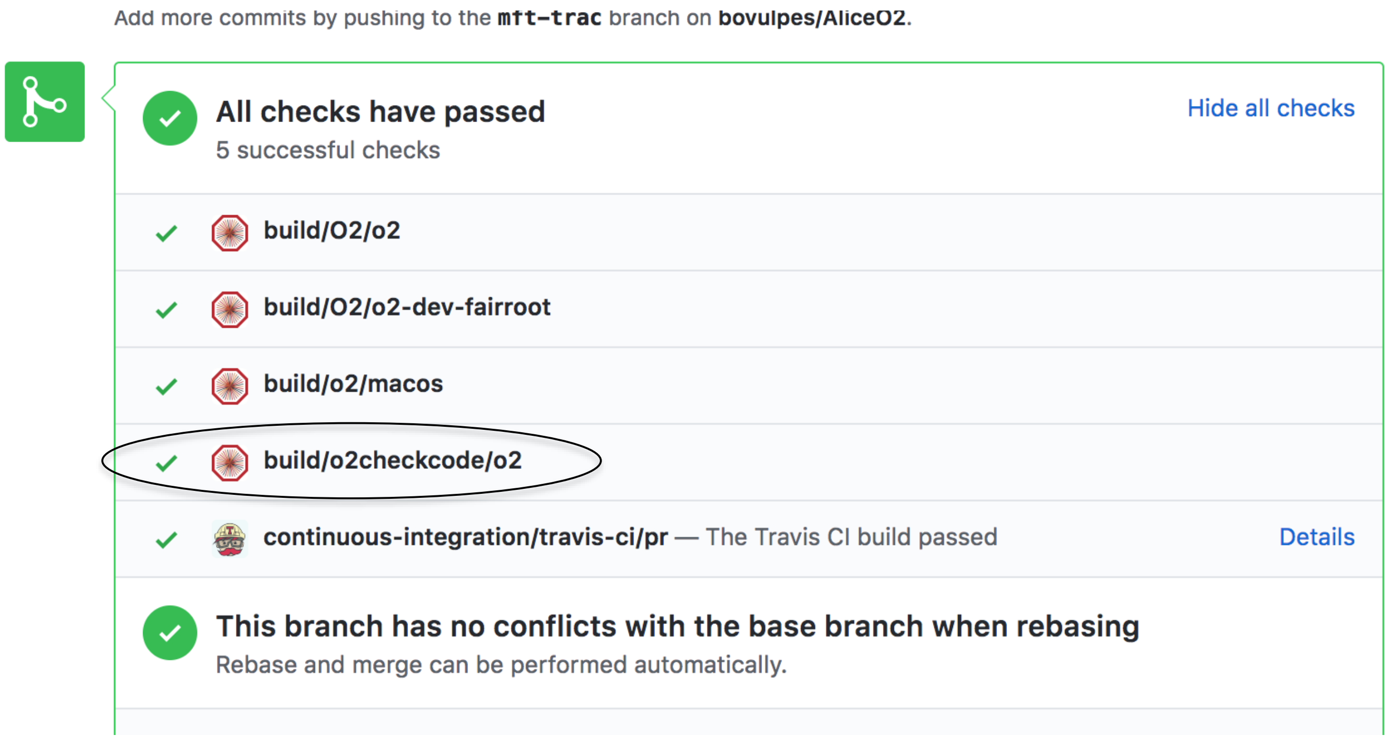
## Code reporting / indexing

We have used static code analysis tools also for **reporting / indexing tasks** which can be **helpful for optimizations**:

- For a given class, **list all the functions that are used** in a certain project
- Given a virtual class**, find out if this class is ever sub-classed** in a given set of code (if not the class does maybe not need to be virtual)
- Find all the **loops in which trigonometric functions are called** (for instance with the goal to vectorize these math function calls)

## Integration into CI

Static analysis checks can naturally be integrated into CI. This is next to standard compilation, unit tests, ... and you will only merge really good code.



Add more commits by pushing to the **mft-trac** branch on **bovulpes/AliceO2**.

**All checks have passed**
5 successful checks

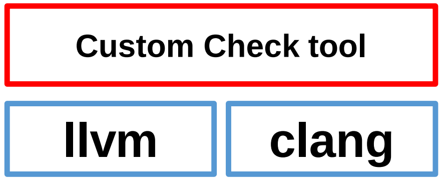
✓ **build/O2/o2**
✓ **build/O2/o2-dev-fairroot**
✓ **build/o2/macos**
✓ **build/o2checkcode/o2**
✓ **continuous-integration/travis-ci/pr** — The Travis CI build passed

**This branch has no conflicts with the base branch when rebasing**
Rebase and merge can be performed automatically.

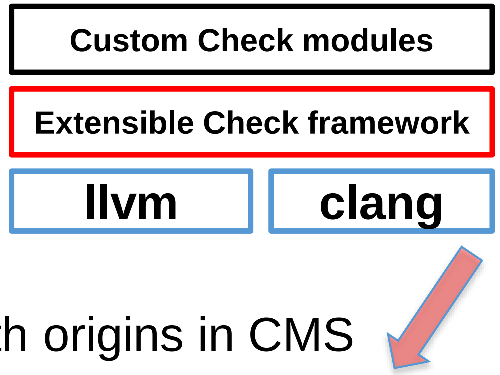
## Static Analysis: How?

- Static code analysis tools **needs access to the C++ abstract syntax tree (AST)** … as such it has a lot in common with an actual compiler
- Most of the static analysis approaches use the **llvm/clang infrastructure** since this provides **libraries and APIs to access the AST**.
- Variations include

Fully custom tool directly based on llvm headers/libraries



“Frameworks” that do heavy lifting ... and are extensible with custom checks



In HEP, one example is Static Analyser Suite (SAS) with origins in CMS

Relatively recent clang-tidy framework as open source project outside HEP

## clang-tidy

http://clang.llvm.org/extra/clang-tidy/

- "**clang-tidy** is a modular static analysis framework and provides a convenient interface for writing new checks."
  - Example in the backup slides

- **large industry** community behind
- already implements wide range of checks and growing
- integrated **ability** to **autocorrect/fix errors in place**
- **very easily extensible**

- Currently the most reasonable choice for most use cases

## Shortcomings?

- Nice to have extensible static analysis frameworks …

- For me a **major inconvenience** (true for clang-tidy or SAS) is the fact that one has to do the **extension within the source environment of the tools**
  - Fork (+ maintain) clone of the git repository

- Rather, we would like to be **able to write custom checks fully outside the framework's source tree** and **extend it dynamically at runtime**.

## Common interest here?

- (Continue) Developing common tools for HEP (see effort by SAS) ?
- Common training / education ?
- Community push for a true dynamic plugin approach to ease development ?

## BACKUP

# Example

```
struct A {
  virtual void foo(int) = 0;
};

struct B : public A {
  virtual void foo(int x) {
    if (x==1)
      printf("hello");
  }
};
```

automatic fix →

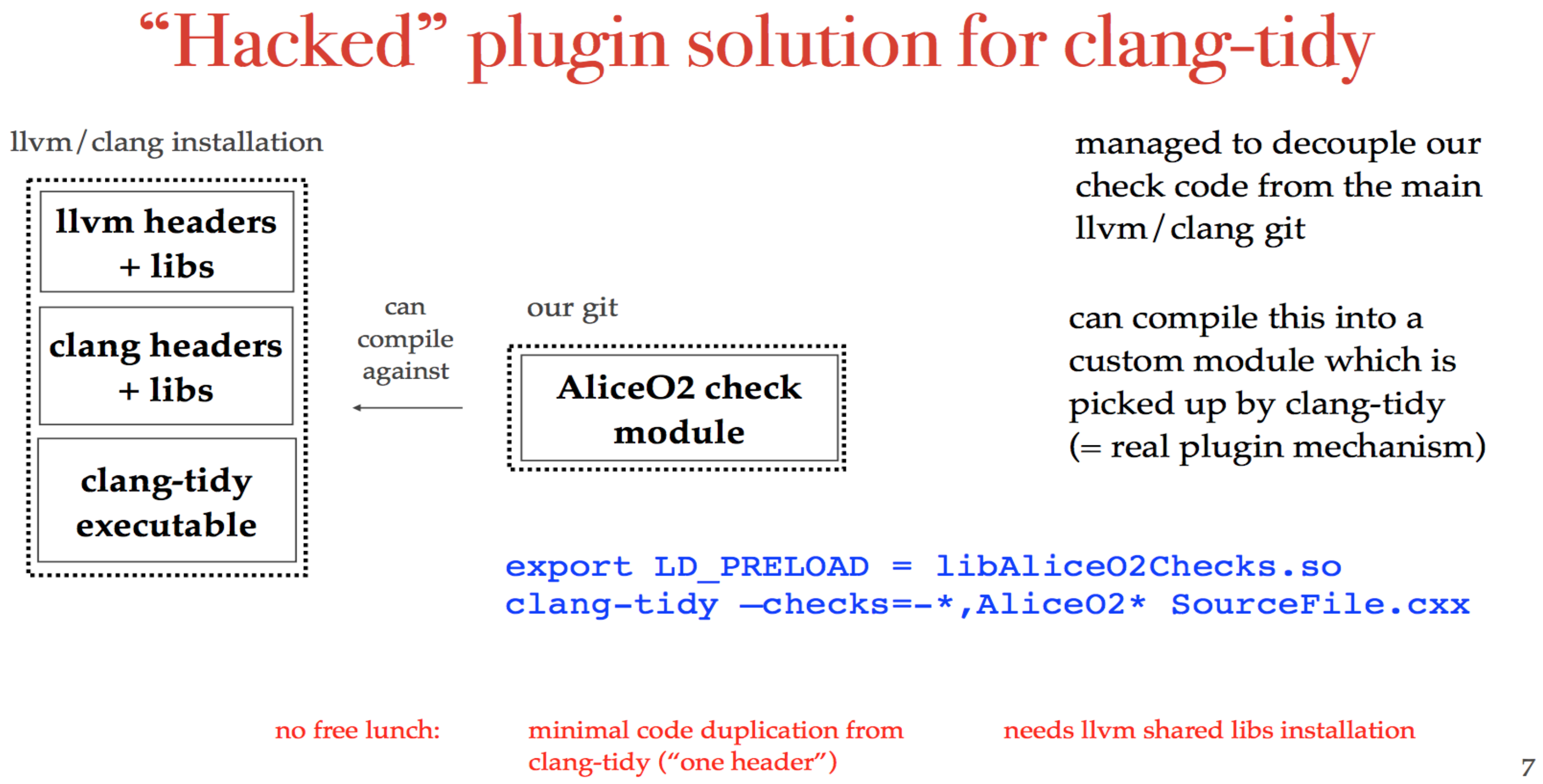
```
struct A {
  virtual void foo(int) = 0;
};

struct B : public A {
  void foo(int x) override {
    if (x==1) {
      printf("hello");
    }
  }
};
```

```
clang-tidy  -checks=-*,moder*over*,read*braces* test.cxx -- -std=c++11
```

```
/Users/swenzel/test.cxx:8:16: warning: prefer using 'override' or (rarely) 'final' instead of
'virtual' [modernize-use-override]
  virtual void foo(int x) {
               ^
                          override
/Users/swenzel/test.cxx:9:13: warning: statement should be inside braces [readability-braces-around-
statements]
    if(x==1)
```

4

## Towards a separation of framework / custom module



### “Hacked” plugin solution for clang-tidy

llvm/clang installation

- llvm headers + libs
- clang headers + libs
- clang-tidy executable

can compile against

our git

AliceO2 check module

managed to decouple our check code from the main llvm/clang git

can compile this into a custom module which is picked up by clang-tidy (= real plugin mechanism)

```
export LD_PRELOAD = libAliceO2Checks.so
clang-tidy —checks=-*,AliceO2* SourceFile.cxx
```

no free lunch: minimal code duplication from clang-tidy (“one header”) needs llvm shared libs installation

7