clang-tidy

Lint-like checks and beyond

(Daniel Jasper - djasper@google.com)

Goal

- **Lint - a C program verifier**:
  Flag code that is *“.. likely to be bugs, to be non-portable, or to be wasteful.”* (lint man page)

- With all the knowledge from **clang**, detect:
  - **Bug prone** coding **patterns**
  - Enforce **coding conventions**
  - Advocate **modern** and **maintainable** code

Example: Unnecessary copies

```
vector<string> SomeStrings = ...;
for (string SomeString : SomeStrings) {
  someFunction(SomeString);
}
```

Example: Unnecessary copies

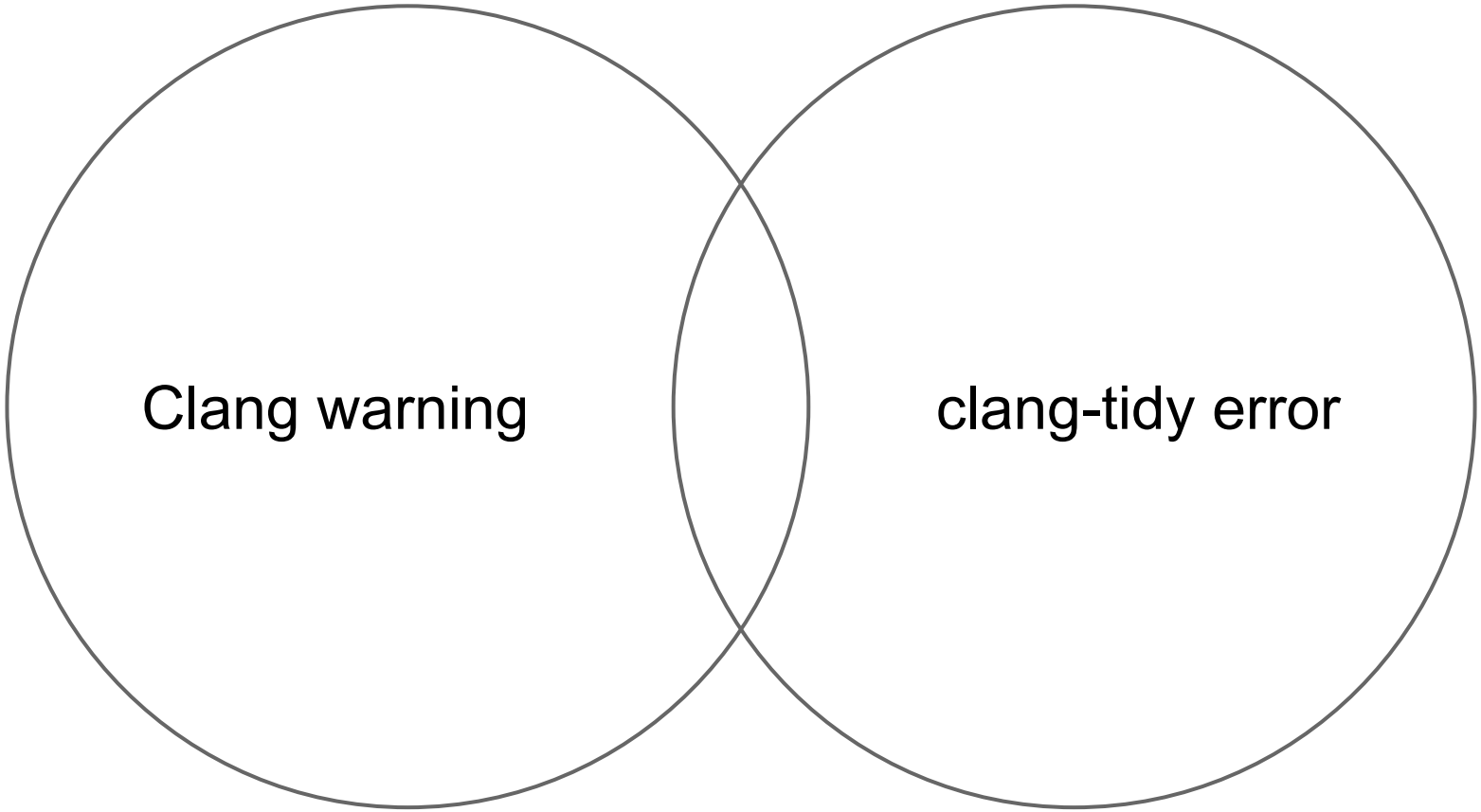
```
map<string, set<int>> SomeStringMap;
for (const pair<string, set<int>> &e :
     SomeStringMap) {
  v.push_back(&e.second);
}
```

Example: Unnecessary copies

```
vector<string> SomeStrings = ...;
for (auto SomeString : SomeStrings) {
  someFunction(SomeString);
}
```

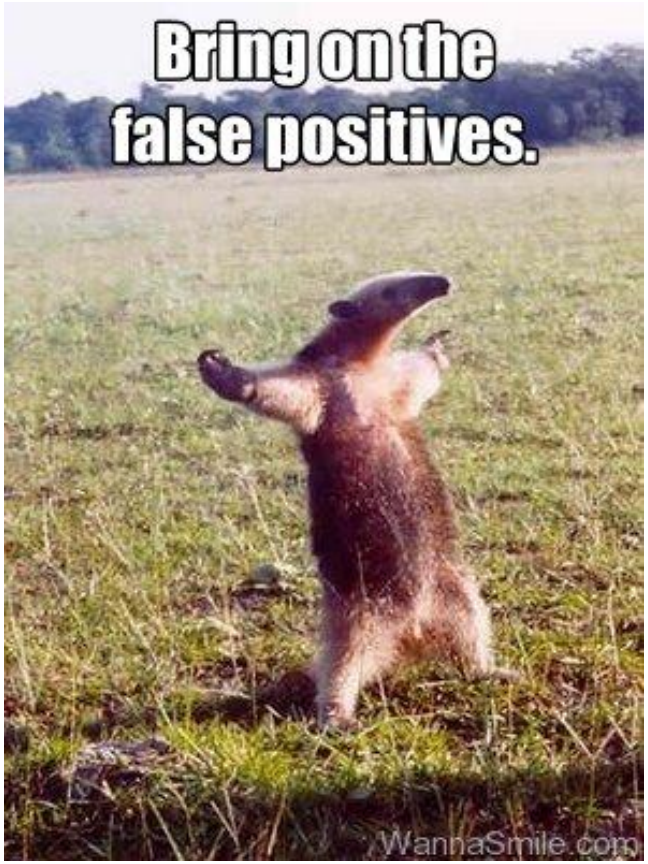
Why not build warnings into Clang?

Why not build warnings into Clang?



False positives

```
vector<int> v;
assert(!v.empty());
for (int i = 0;
      i < v.size() - 1;
      ++i) {
  llvm::outs() << v[i];
}
```

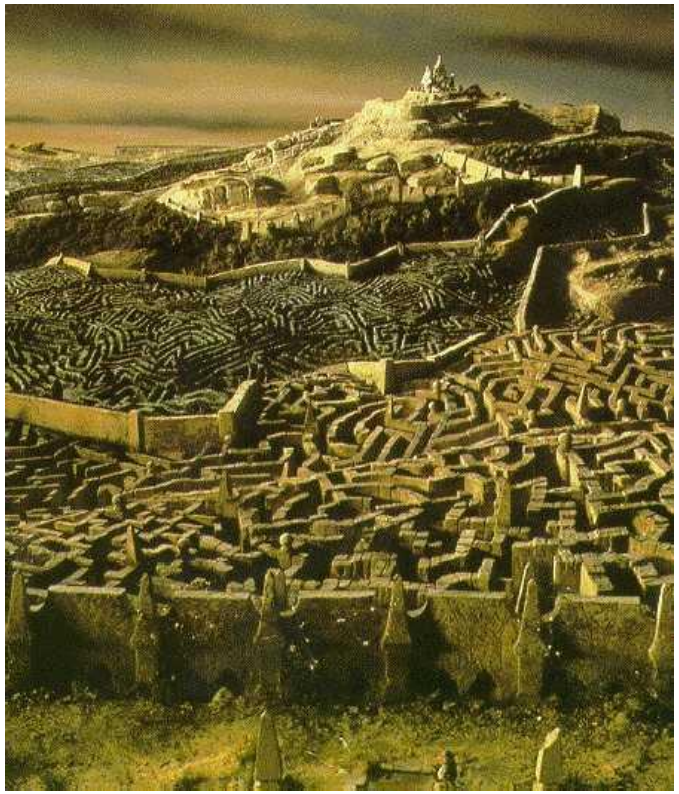


Encapsulated implementation

`include/clang/Sema/Sema.h` (8.2k LOC)

Implemented in 35 files
(`lib/Sema/` - 146k LOC)

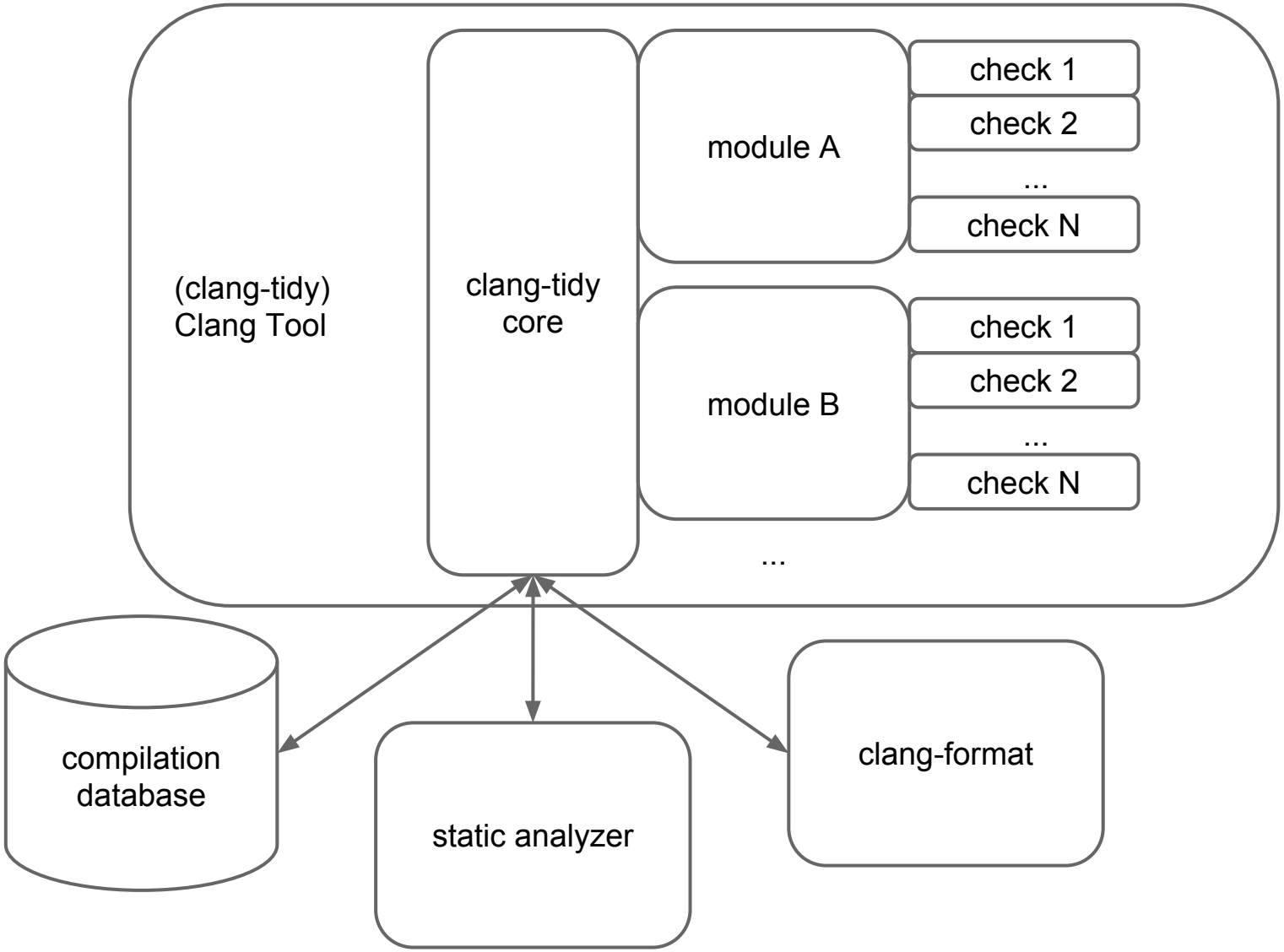
E.g. -Wunused-private-field:
`Sema.cpp`,
`SemaDeclCXX.cpp`,
`SemaExprMember.cpp`



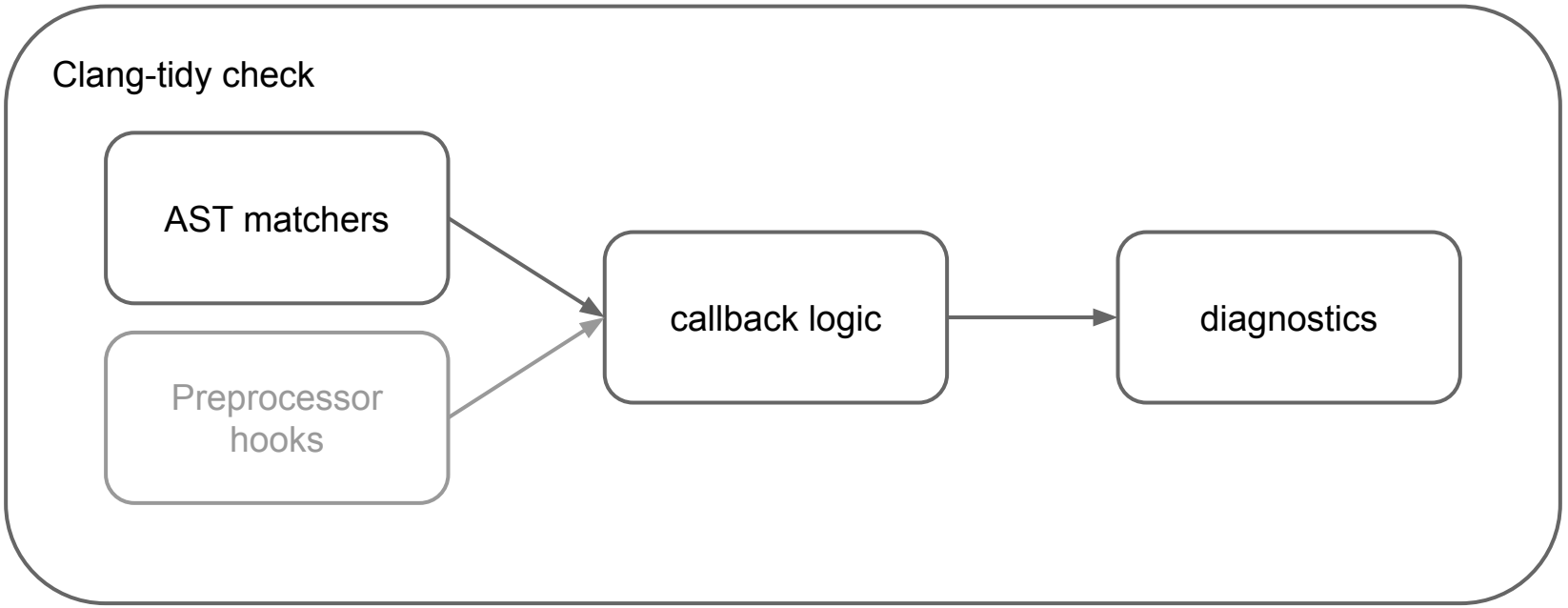
General applicability

- Lint checks ..
  - .. can be **project specific**
  - .. can be more **expensive** than Clang's compilation (e.g. doing static code analysis)
  - .. are not needed during every compilation

Architecture

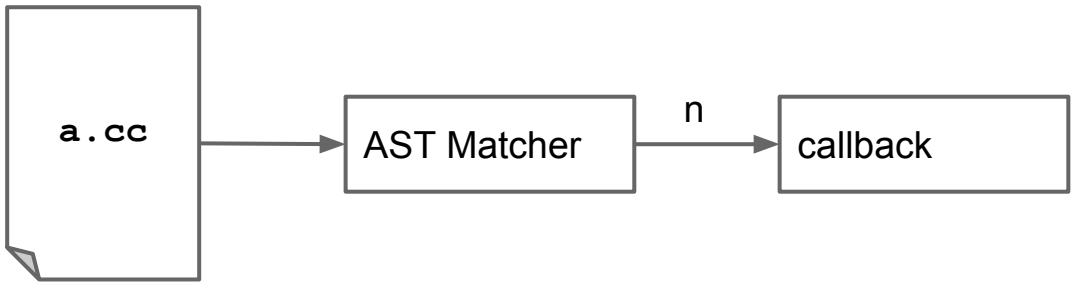


Architecture



AST matchers

- Syntax tree (AST) **matchers**
- A matcher finds specific **entries of the AST**
- A **callback** gets invoked on every match
  - Can access the AST



AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr()
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee()
)
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee(methodDecl())
)
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee(methodDecl(hasName("Get")))
)
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee(methodDecl(hasName("Get"))),
  thisPointerType()
)
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee(methodDecl(hasName("Get"))),
  thisPointerType(recordDecl())
)
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee(methodDecl(hasName("Get"))),
  thisPointerType(recordDecl(
      hasName("Elements")))
)
```

AST matchers - Example

Match calls to method *Get* on class *Elements*

```
e = elements.Get(42);
f = fish->Get(23);
f.Cook();
feed();
```

```
callExpr(
  callee(methodDecl(hasName("Get"))),
  thisPointerType(recordDecl(...)),
  callee(memberExpr().bind("callee")))
)
```

Example: Explicit constructors

From Google's C++ style guide:

"Use the C++ keyword `explicit` *for constructors with one argument. ..."*

- Avoids accidental object construction

Example: Explicit constructors

```
class ExplicitConstructorCheck : public ClangTidyCheck {
public:
  virtual void registerMatchers(ast_matchers::MatchFinder *Finder) {
    Finder->addMatcher(constructorDecl().bind("constructor"), this);
  }
};
```

Example: Explicit constructors

```
class ExplicitConstructorCheck : public ClangTidyCheck {
public:
  virtual void registerMatchers(ast_matchers::MatchFinder *Finder) {
    Finder->addMatcher(constructorDecl().bind("constructor"), this);
  }

  virtual void check(const ast_matchers::MatchFinder::MatchResult &Result) {
    const CXXConstructorDecl *Ctor =
        Result.Nodes.getNodeAs<CXXConstructorDecl>("constructor");
    if (!Ctor->isExplicit() && !Ctor->isImplicit() &&
        Ctor->getNumParams() >= 1 && Ctor->getMinRequiredArguments() <= 1) {
      SourceLocation Loc = Ctor->getLocation();
      diag(Loc, "Single-argument constructors must be explicit")
          << FixItHint::CreateInsertion(Loc, "explicit ");
    }
  }
};
```

Demo time

Missing pieces

- Many, many, many … more checks
- Cross-TU changes (A::SomeFunction() → A::someFunction())
- Per-project configuration (.clang-tidy file)
- VCS integration (Only check changed lines)
- clang-modernize integrations
- ...

Thank you!

(Now go and develop checks)