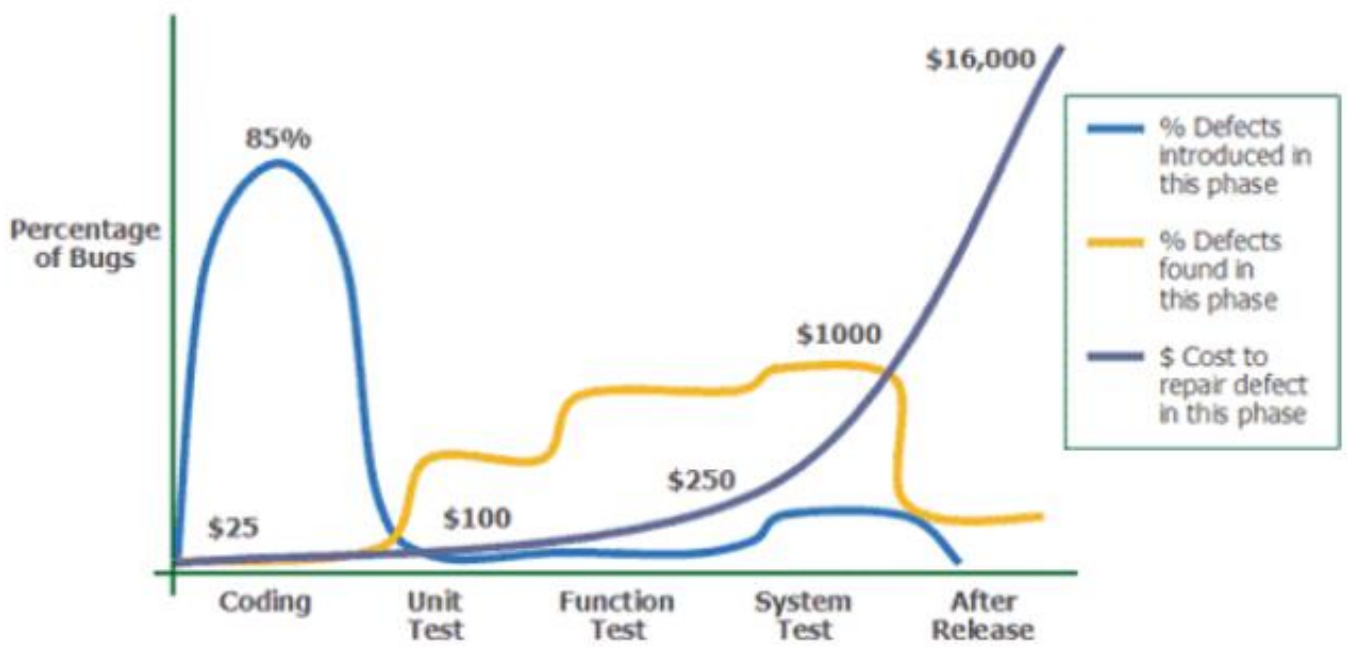
Clang-tidy for Customized Checkers and Large Scale Refactoring

Vince Bridgers

Overview

- Why use tools like Syntax and Static Analyzers?
- How do these tools fit into a process flow?
- Examples of text matchers using clang-query, compare and contrast with analysis
- Simple example clang-tidy check – “soup to nuts”
- References for “homework” ☺

Why tools like Clang-tidy?: Cost of Software Development



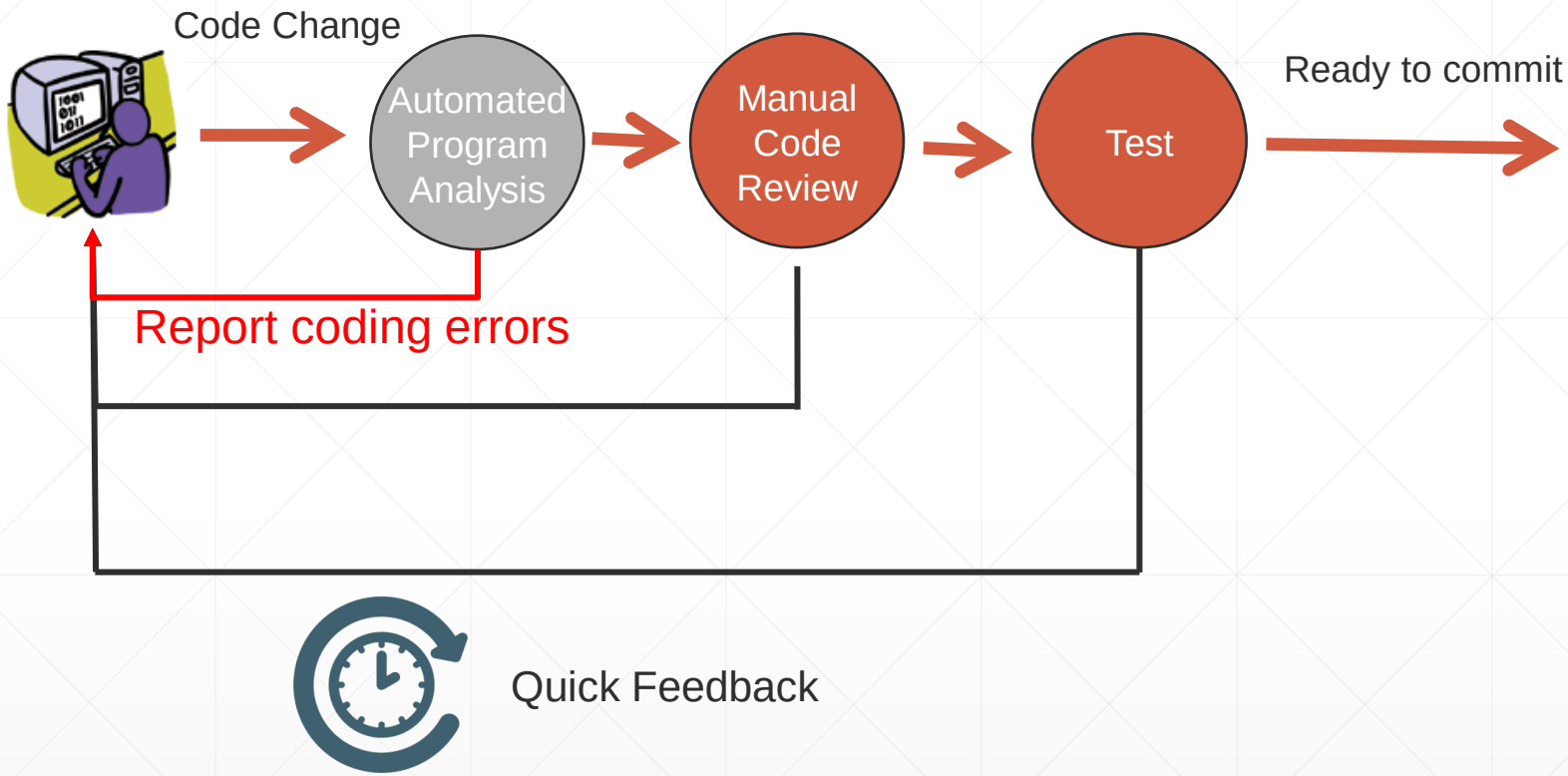
- **Notice most bugs are introduced early in the development process, and are coding and design problems.**
- **Most bugs are found during unit test, where the cost is higher**
- **The cost of fixing bugs grow exponentially after release**
- ***Conclusion: The earlier the bugs found, and more bugs found earlier in the development process translates to less cost***

Source: Applied Software Measurement, Caspers Jones, 1996

Four Pillars of Program Analysis

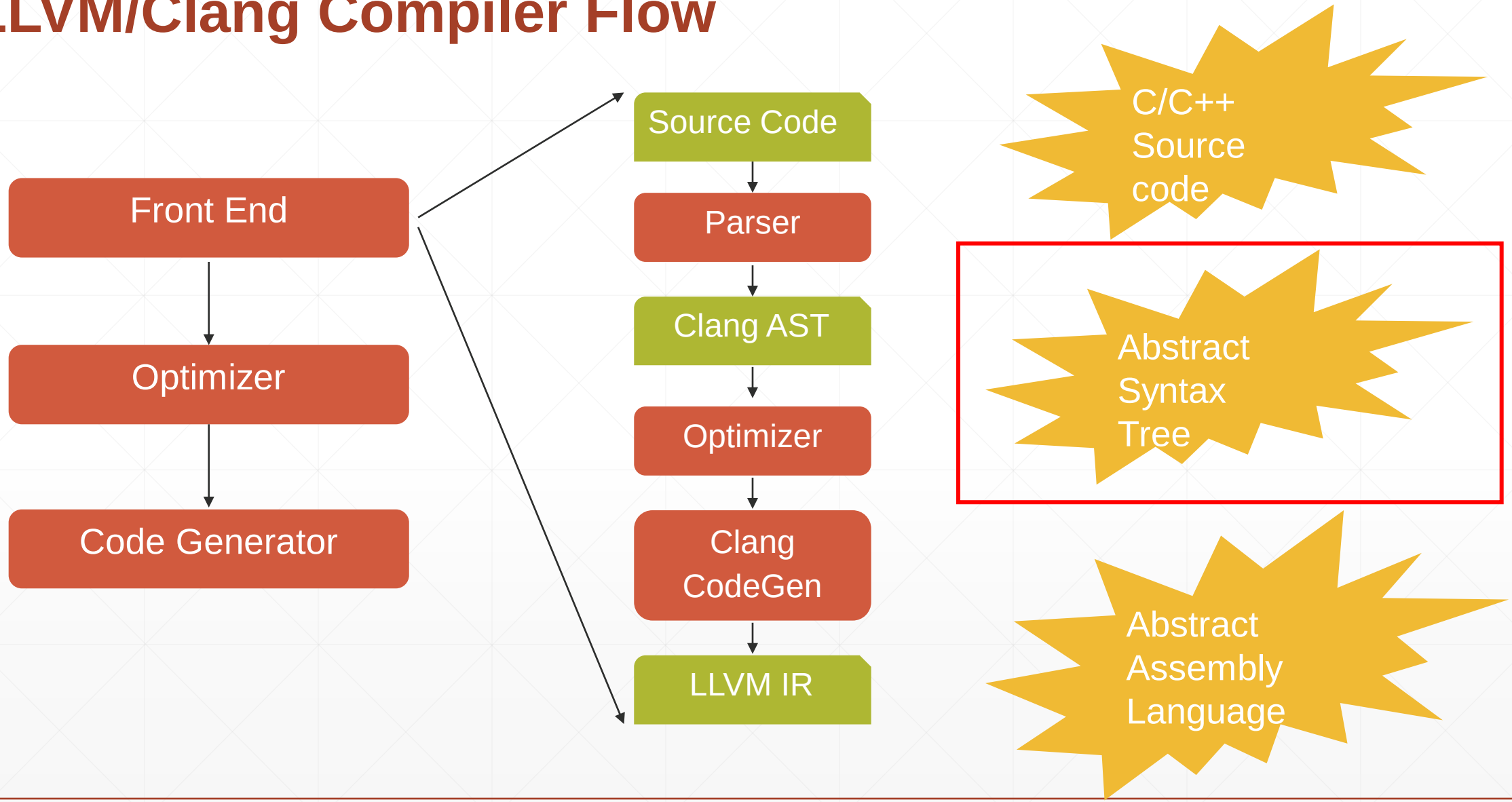
	Compiler diagnostics	Linters, style checkers	Static Analysis	Dynamic Analysis
Examples	Clang, gcc, cl	Lint, clang-tidy, Clang-format, indent, sparse	Cppcheck, gcc 10+, clang	Valgrind, gcc and clang
False positives	No	Yes	Yes	Not likely, but possible
Inner Workings	Programmatic checks	Text/AST matching	Symbolic Execution	Injection of runtime checks, library
Compile and Runtime affects	None	Extra compile step	Extra compile step	Extra compile step, extended run times

Typical CI Loop with Automated Analysis



Syntax, Semantic, and Analysis Checks:
Can analyze properties of code that cannot be tested (coding style)!
Automates and offloads portions of manual code review
Tightens up CI loop for many issues

LLVM/Clang Compiler Flow



https://www.youtube.com/watch?v=m8G_S5LwlTo – LLVM IR Tutorial

Clang-tidy & Static Analyzers – Compare and Contrast

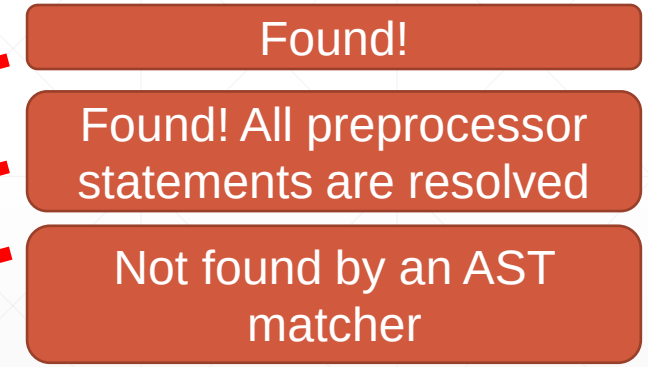
- Clang Static Analysis uses Symbolic Execution
- Clang-tidy uses AST Matchers
  - Finds patterns, optionally replace/add/remove patterns
- Both use the AST

AST Matcher compared to Symbolic Execution

- How to find all instances of possible division by zero before run time?

```
binaryOperator(hasOperatorName("/"),
  hasRHS(integerLiteral(equals(0)).bind(KEY_NODE)));
```

```
#define ZERO 0
int function(int b)
{
  int a,c;
  switch (b){
    case 1:  a = b/0; break;
    case 2:  a = b/ZERO; break;
    case 4:  c = b-4;
             a = b/c; break;
    };
  return a;
}
```



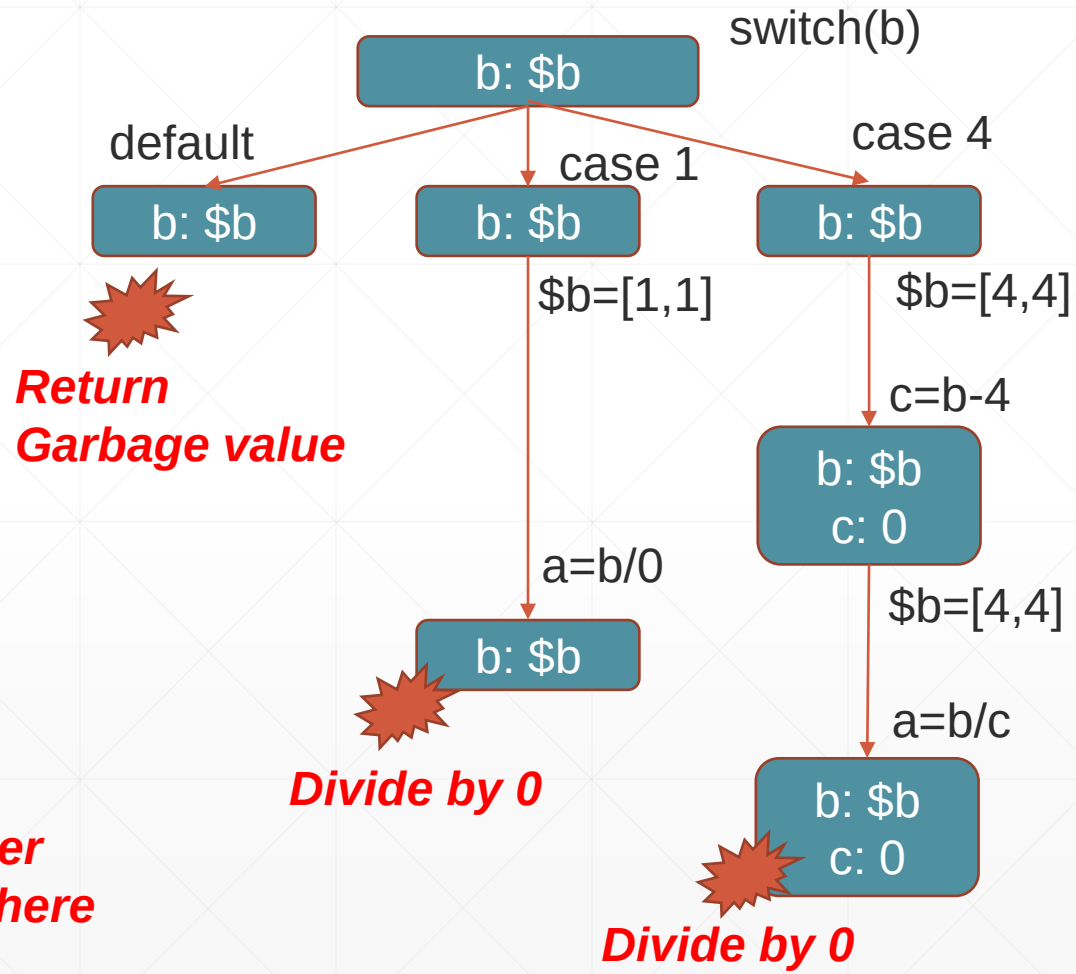
Clang Static Analyzer – Symbolic Execution

- Finds bugs without running the code
- Path sensitive analysis
- CFGs used to create exploded graphs of simulated control flows

```
int function(int b) {
    int a, c;
    switch (b) {
        case 1: a = b / 0; break;
        case 4: c = b - 4;
                a = b/c; break;
    }
    return a;
}
```

Compiler warns here

Static Analyzer warns here



Source: Clang Static Analysis - Gabor Horvath - Meeting C++ 2016

Clang-tidy

- Now with this perspective, shifting focus to clang-tidy
- A Clang based C++ Linting tool framework
- Full access to the AST and preprocessor
- Clang-tidy is extensible – custom checks are possible
- More than 200 existing checks
  - Readability, efficiency, correctness, modernization
  - Highly configurable
  - Can automatically fix the code in many place

See http://clang.llvm.org/extra/clang-tidy, list of checks here https://clang.llvm.org/extra/clang-tidy/checks/list.html.

Clang-tidy Quick Demo (demo1)

- Dump AST : clang –cc1 –ast-dump init.cpp
- clang-tidy –list-checks
- clang-tidy –list-checks –checks=*
- clang-tidy --checks=-*,cppcoreguidelines-init-variables init.cpp --
- clang-tidy --checks=-*,cppcoreguidelines-init-variables --fix init.cpp –

Clang-tidy Uses

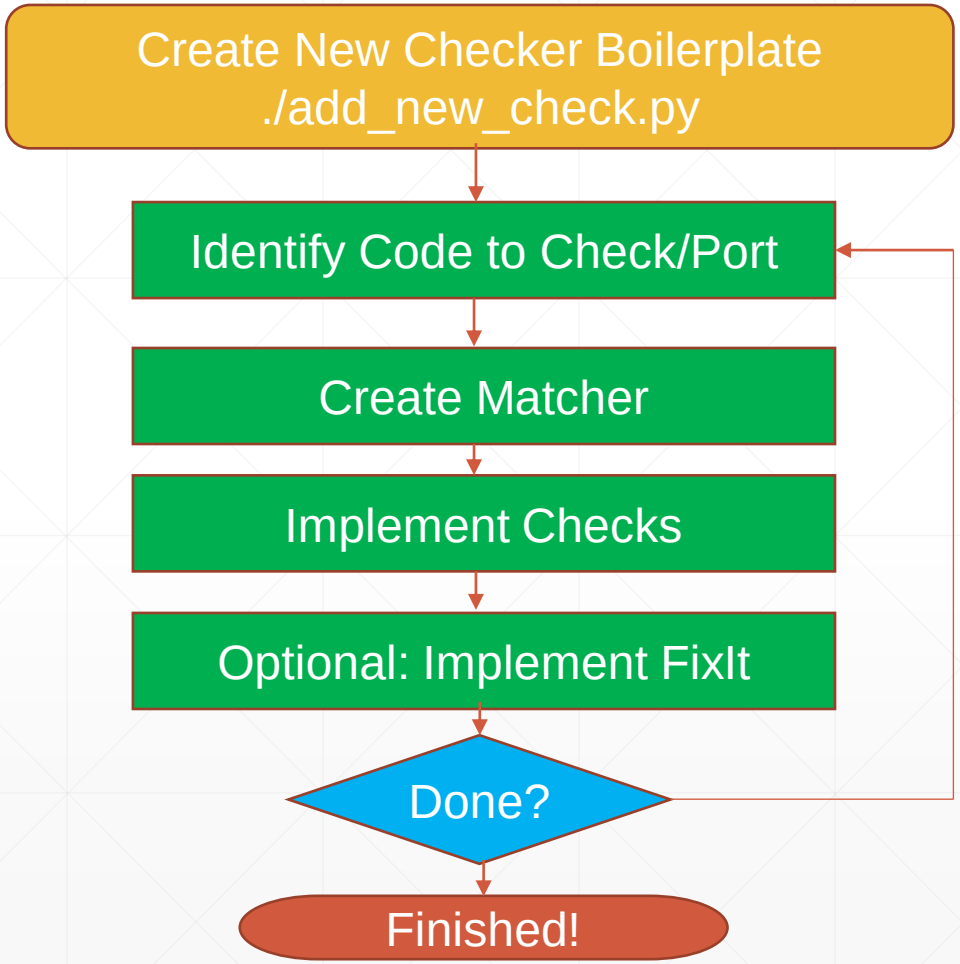
- Implement checks and changes that require semantic knowledge of the language
- Implement specialized checks for your organization
- Create acceptance tests for delivery of third-party work product
- Large scale refactoring
- Used by developers interactively during development & test
- Integration into your CI flow – Automated and repeatable
  - Moves subjectivity of the code review process to objective computer automation

Clang-tidy Notes

- Not all checkers have “Fix”’s. See list of existing checkers for an example.
- Why would not all checkers have fixes?
  - Some checks are not perfect, but “good enough” – 80% rule.
  - Highlight certain patterns for further scrutiny
  - Custom checks
- Can pass compiler commands to the compiler, example …
  - `clang-tidy --extra-arg="-DMY_SWEET_DEFINE=1" --checks=-*,cppcoreguidelines-init-variables init.cpp  --`
- What’s that “--” at the end?
  - Says that we’re not using a compile_commands.json – more on that later.

See http://clang.llvm.org/extra/clang-tidy, list of checks here https://clang.llvm.org/extra/clang-tidy/checks/list.html.

Clang-tidy check dev process



Imagine your manager wants a new API

- You have this cool new processor architecture that needs a “special” allocator because of a bug in first silicon (This has *never* happened before \\\ ☺).
- Change all instances of “`void *malloc(size_t)`” to “`void *acme_zalloc(size_t)`” in a test repo of about 10,000 files spread across maybe 50 directories.
  - Don’t look for a new job yet – there’s an opportunity to be a “hero”, get that “cup of coffee” bonus your manager pays out for extraordinary accomplishments
- Ok, maybe you really can do this with a simple shell or Python script – but imagine this as a first step, and you don’t know what other problems the hardware guys left in store for you.
- So, we’ll use the clang tools Python script to create boilerplate for this …

Clang-tidy Adding a Check (demo2)

- cd to <root>/clang-tools-extra/clang-tidy
- ./add_new_check.py misc change-malloc (See output)
- Rebuild …
- Check listed checkers – new one should show up!
  - clang-tidy --list-checks –checks=* | grep change
- To run the new checker (not yet though, we need a few changes) …
  - clang-tidy --checks=-*,misc-change-malloc file.c
  - clang-tidy --checks=-*,misc-change-malloc --fix file.c

We'll need to explore a code sample

```
#include <stdlib.h>

void *acme_zalloc(size_t s) {
    void *ptr = malloc(s);
    memset(ptr, 0, s);
    return ptr;
}

void * foo(int s) {
    return malloc(s);
}
```

Our new implementation

Don't touch this one (I'll show ya)

Change to acme_zalloc()

Let's see what the AST looks like first … (demo3)

Extending clang-tidy …

- See https://clang.llvm.org/docs/LibASTMatchersReference.html
- Many existing matchers, and can be extended (subject for another day)
- If you’re overwhelmed so far – no worries! This *is* difficult. Hang in there, we’ll go through some simple examples to get started.
- We’re driving towards our simple tutorial example – best place to start!

Clang AST for our sample (demo3)

```
#include <stdlib.h>
#include <memory.h>

int foo(void) {
    void *ptr = malloc(4);

    free(ptr);
    return 0;
}

int fee(int i) {
    return i*2;
}

int gee(int i) {
    return i/2;
}

int anError(int i) {
    return i/0;
}
```

- For demo purposes, I'll use this code, we'll come back to our manager's code
- See references at the end for Intro to AST, and AST matchers.
- I'll go through a few example explorations specific to the problem posed with some hints for optimizing your explorations.

Step 1: Replace “malloc”

- Most of the difficult work is done – we have a basic matcher expression we can use.
- From our exploration …
  - Matcher -> `callExpr(callee(functionDecl(hasName("malloc"))))`
- How to translate to code? In our registerMatchers override …

```
void ChangeMallocCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(callExpr(callee(functionDecl(hasName("malloc")))).bind("malloc"),this);
}
```

- This adds a matcher and binds to a name “malloc” for us to use in our check override.

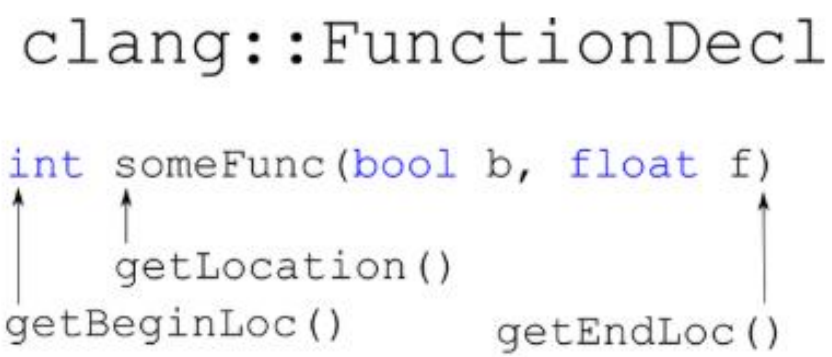
Step 1: Replace “malloc” …

- In our “check” override …

```
void ChangeMallocCheck::check(const MatchFinder::MatchResult &Result) {
  const CallExpr *callExpr = Result.Nodes.getNodeAs<CallExpr>("malloc");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_zalloc() instead of malloc()")
      << FixItHint::CreateReplacement(SourceRange(start, start.getLocWithOffset(strlen("malloc")-1)),
         "acme_zalloc");
  }
}
```

- This code uses our match, and creates a replacement for “malloc”, with a diagnostic, and an optional “fix”
- What are these calls for Source Range and BeginLoc()?

Source location



- There exists methods to help with source replacement
- Each AST node has location associated with it that can be retrieved.
- I'll not spend too much time on this, but there's more to explore and learn here.
- Let's compile the example and try it out!

Step 2: “If you give a mouse a cookie …”

- Someone discovered we need to change a few thousand files to use a new API
  - This is contrived, I know – please suspend logic for now, this is a tutorial after all ☺
- Transform “`void *malloc(size_t)`” -> “`void *acme_zalloc(size_t, int)`”, and “`void free(void *)`” -> “`void acme_free(void **)`”. Let’s assume all of our files include a single top level include that we can add new interface prototypes and defines too.
- First step – extend the matchers …

```
void ChangeMallocCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(callExpr(callee(functionDecl(hasName("malloc"))))).bind("malloc"),this);
  Finder->addMatcher(callExpr(callee(functionDecl(hasName("free"))))).bind("free"),this);
}
```

Step 2: Replace “free”, extend “malloc”

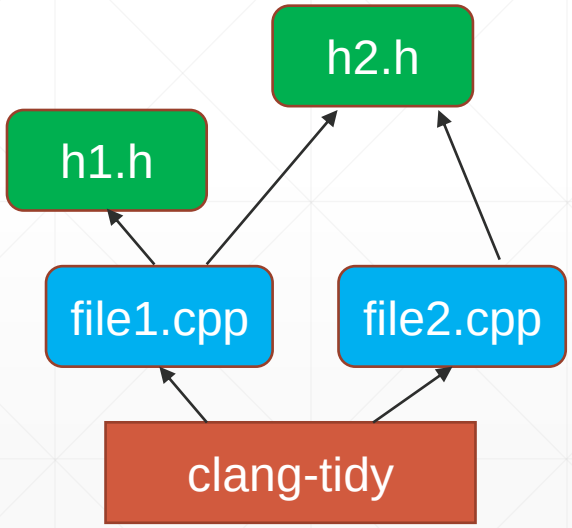
```
void ChangeMallocCheck::check(const MatchFinder::MatchResult &Result) {
  SmallString<64> NewArgument;
  const CallExpr *callExpr = Result.Nodes.getNodeAs<CallExpr>("malloc");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_zalloc() instead of malloc()")
      << FixItHint::CreateReplacement(SourceRange(start, start.getLocWithOffset(strlen("malloc")-1)),
         "acme_zalloc");
    NewArgument = Twine(", ZERO_INITIALIZE").str();
    const auto InsertNewArgument = FixItHint::CreateInsertion(callExpr->getEndLoc(), NewArgument);
    Diag << InsertNewArgument;
  }
  callExpr = Result.Nodes.getNodeAs<CallExpr>("free");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_free() instead of free()")
      << FixItHint::CreateReplacement(SourceRange(start, start.getLocWithOffset(strlen("free")-1)),
         "acme_free");
    Diag << FixItHint::CreateInsertion(callExpr->getArg(0)->getBeginLoc(), "(void **)&");
  }
}
```

Demo3 – Repeat with new changes

- Rebuild, retry …

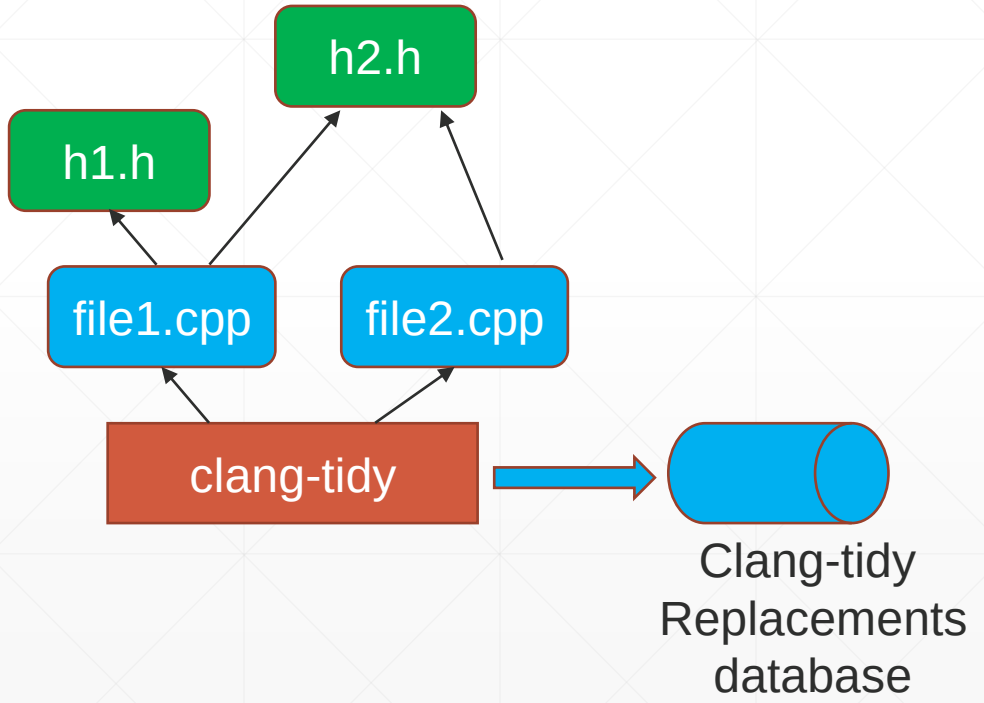
Clang-tidy for Projects

- Examples shown so far are for clang-tidy for one file.
- What if we want to process multiple files across a source repo?
- file1.cpp, h1.h, and h2.h are modified first step.
- Then file2.cpp is modified, but could fail to compile properly.
- How to address?
- There is a solution!



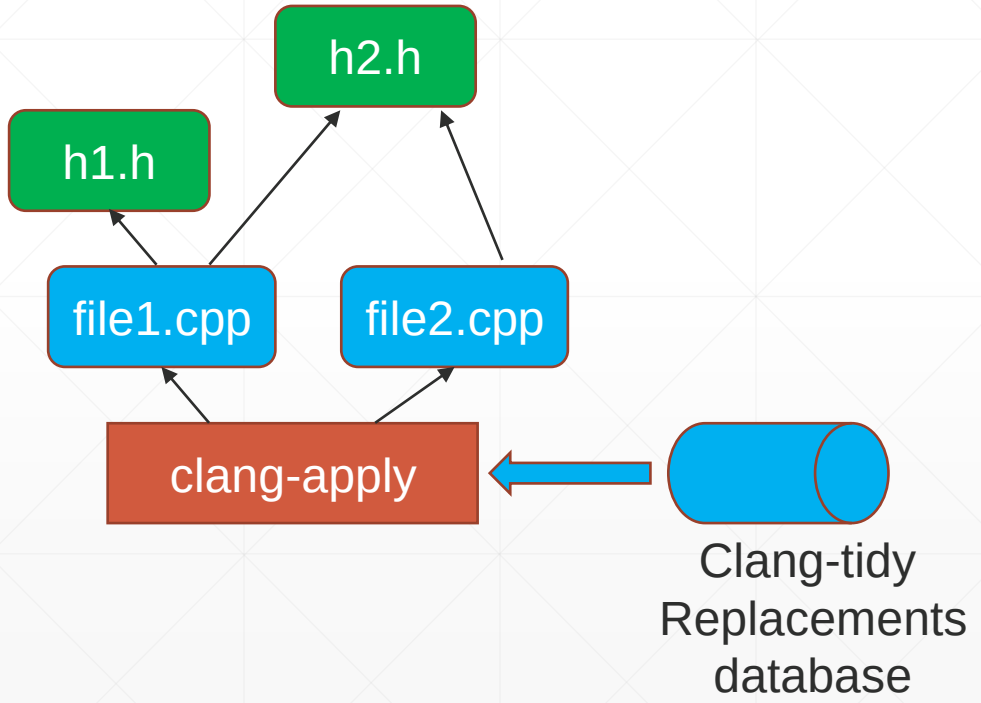
Clang-tidy for Projects

- file1.cpp, h1.h, and h2.h are processed, and modifications stored in a yaml file.
- file2.cpp is processed, changes stored to a yaml file.



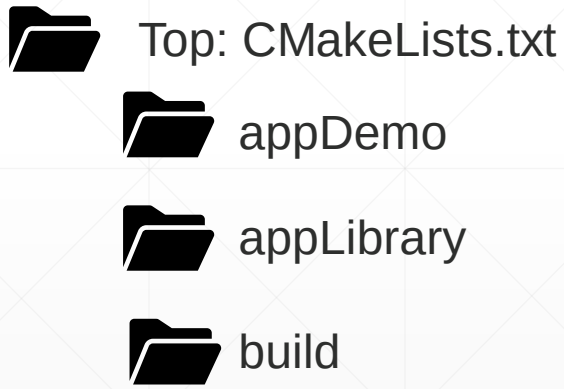
Clang-tidy for Projects

- The clang-apply-replacements tool will process the changes after clang-tidy is complete.
- No problem!
- clang-tidy/tool/run-clang-tidy.py
  - Runs clang-tidy in parallel
  - Can use matching patterns
  - Handles deferred replacements



Example – Transforming Large Scale Project

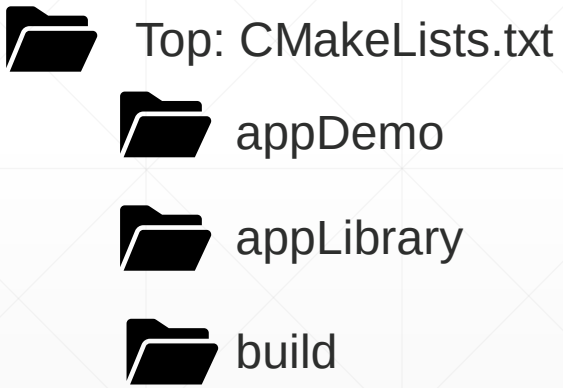
- In this case – cmake based. Cmake supports compile_commands.json generation.
- Application directory and library directory.
- Build: cd build & …
  - cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -G Ninja ../
- Clang-tidy checks on project
  - run-clang-tidy.py -header-filter='.*' -checks='-*,misc-change-malloc'
- Apply our fixes – use –fix
- Avoid applying multiple fixes simultaneously – use just one at a time, test, commit then repeat iteratively.



Compile commands JSON: *https://sarcasm.github.io/notes/dev/compilation-database.html#how-to-generate-a-json-compilation-database*

Example – Transforming Large Scale Project

- Demo4



Supporting LIT Test case

```
// RUN: %check_clang_tidy %s misc-change-malloc %t
void f() {
  void *p=malloc(1);
// CHECK-MESSAGES: warning: use acme_zalloc() instead of malloc() [misc-change-malloc]
// CHECK-FIXES: void *p=acme_zalloc(1, ZERO_INITIALIZE);
  free(p);
// CHECK-MESSAGES: warning: use acme_free() instead of free()
// CHECK-FIXES: acme_free((void **)&p);
}
```

- We *always* want a supporting LIT test case for every new checker.
- Positive ***and*** *negative* use cases

Supporting LIT Test case

- Demo5 – LIT test case

Conclusion

- “Soup to nuts” – how to build a simple clang-tidy base checkers and refactoring tool.
- Not covered today – Preprocessor callbacks, adding include files
- Lot’s to explore!
  - Resources in the references
  - Try clang-query using different source examples. Get creative with AST matcher expressions.
  - Improve the LIT tests presented
  - Try adding your own category of checkers (not inserted into “misc”)

References

- Introduction to the Clang AST - https://clang.llvm.org/docs/IntroductionToTheClangAST.html
- Matching the Clang AST - https://clang.llvm.org/docs/LibASTMatchers.html
- AST Matcher Reference - https://clang.llvm.org/docs/LibASTMatchersReference.html
- Stephen Kelly's blog - https://devblogs.microsoft.com/cppblog/author/stkellyms/, https://steveire.wordpress.com/
- Tutorial source - https://github.com/vabridgers/LLVM-Virtual-Tutorial-2020.git
- The complete compile_commands.json reference - https://sarcasm.github.io/notes/dev/compilation-database.html
- See http://clang.llvm.org/extra/clang-tidy, list of checks here https://clang.llvm.org/extra/clang-tidy/checks/list.html

Thank you for attending!