Austin LLVM Meetup

Vince Bridgers

7/22/2020

Meetup Agenda

- Today
  - Compare, contrast static analysis and Clang Tidy
  - How these tools are used
  - Demo of writing a simple Clang Tidy Checker, with a roadmap for your future use
- Sponsorship
  - Our space is sponsored by Capital Factory
  - Meetup fees are paid by llvm.org – thank you, Arnaud de Grandmaison!

- Note: Much of this material is based of previous work from Stephen Kelly, see his blog at https://devblogs.microsoft.com/cppblog/author/stkellyms/ for detailed presentations on clang-query usage. I'm building on some of his excellent content there. Some of Stephen's changes are not yet upstreamed, so I'm focusing on what llvm/clang does today, built from Tip of Tree on 7/18/2020.

Meetup Schedule, Space

- 4th Wednesday every month
- We've gone virtual!
- Materials posted at
  - https://github.com/vabridgers/Austin-LLVM-Meetup-Material.git.
- Next meeting in August – will cover a sample static analysis pass in the same guided tutorial sort of way. Any particular static analysis passes of interest not already covered by Clang? Post suggestions in the meetup comments. ☺
- Would anyone like to present "virtually" at our next meetup ? Please contact me through the Meetup page.

Overview

- Why use tools like Static and Text Analyzers?
- Some context for Static Analysis, Text Analysis
- How do these tools fit into a process flow?
- Clang-tidy, Clang-query
- Examples of text matchers using clang-query
- Example clang-tidy check – “soup to nuts”

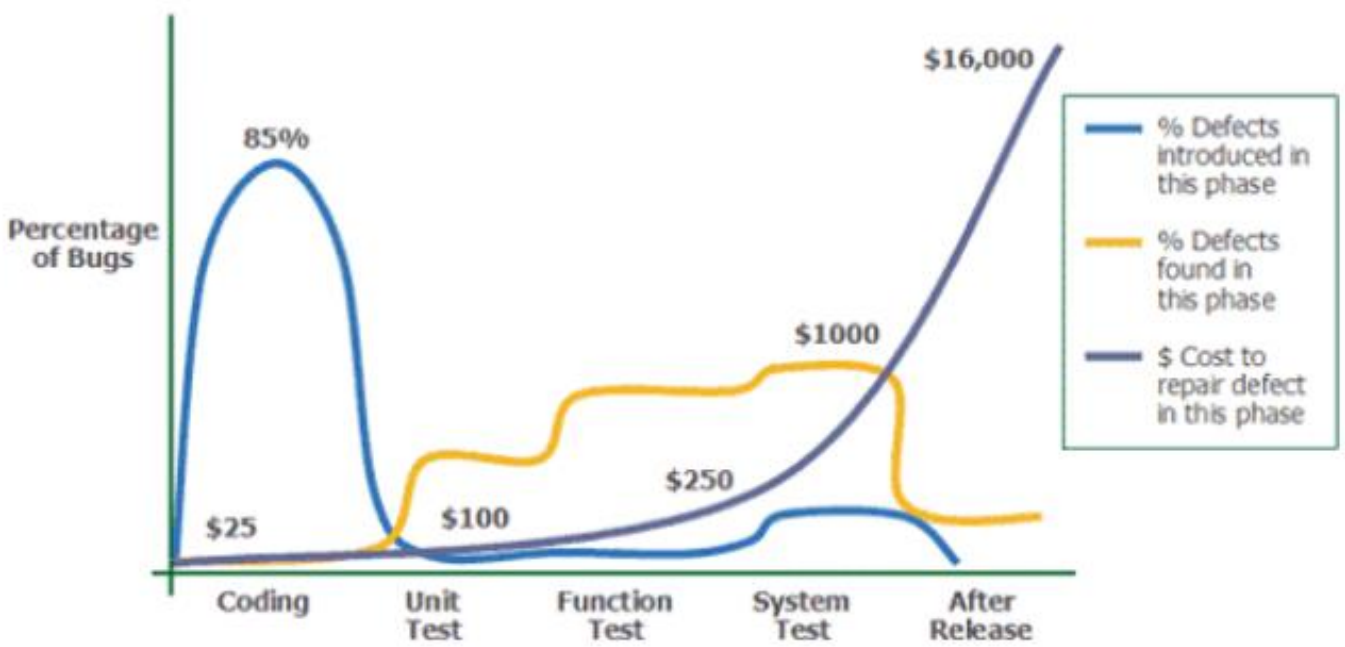
Demo - Environment, Tools

- Oracle VirtualBox VM, Ubuntu 20.04 on Win10
- 16G memory, 100GB storage, 4 processors
- Bidi sharing enabled, Guest Additions added
- sudo apt install gcc g++ cmake ninja-build python vim

Start “git clone” & build

- git clone https://github.com/llvm/llvm-project.git
  - mkdir llvm-install; mkdir llvm-build; cd llvm-build
- X86
  - cmake -DLLVM_ENABLE_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DLLVM_TARGETS_TO_BUILD="X86" -DCMAKE_INSTALL_PREFIX=<root>/llvm-install -G Ninja <root>/llvm-project/llvm
  - ninja; ninja install; ninja check-all

Why tools like SA and Tidy?: Cost of bugs



- Notice most bugs are introduced early in the development process.
- Most bugs are found in the middle of the development process
- The cost of fixing bugs grow exponentially the later they are found in the development process
- *Conclusion: The earlier the bugs found, and more bugs found earlier in the development process translates to less cost*

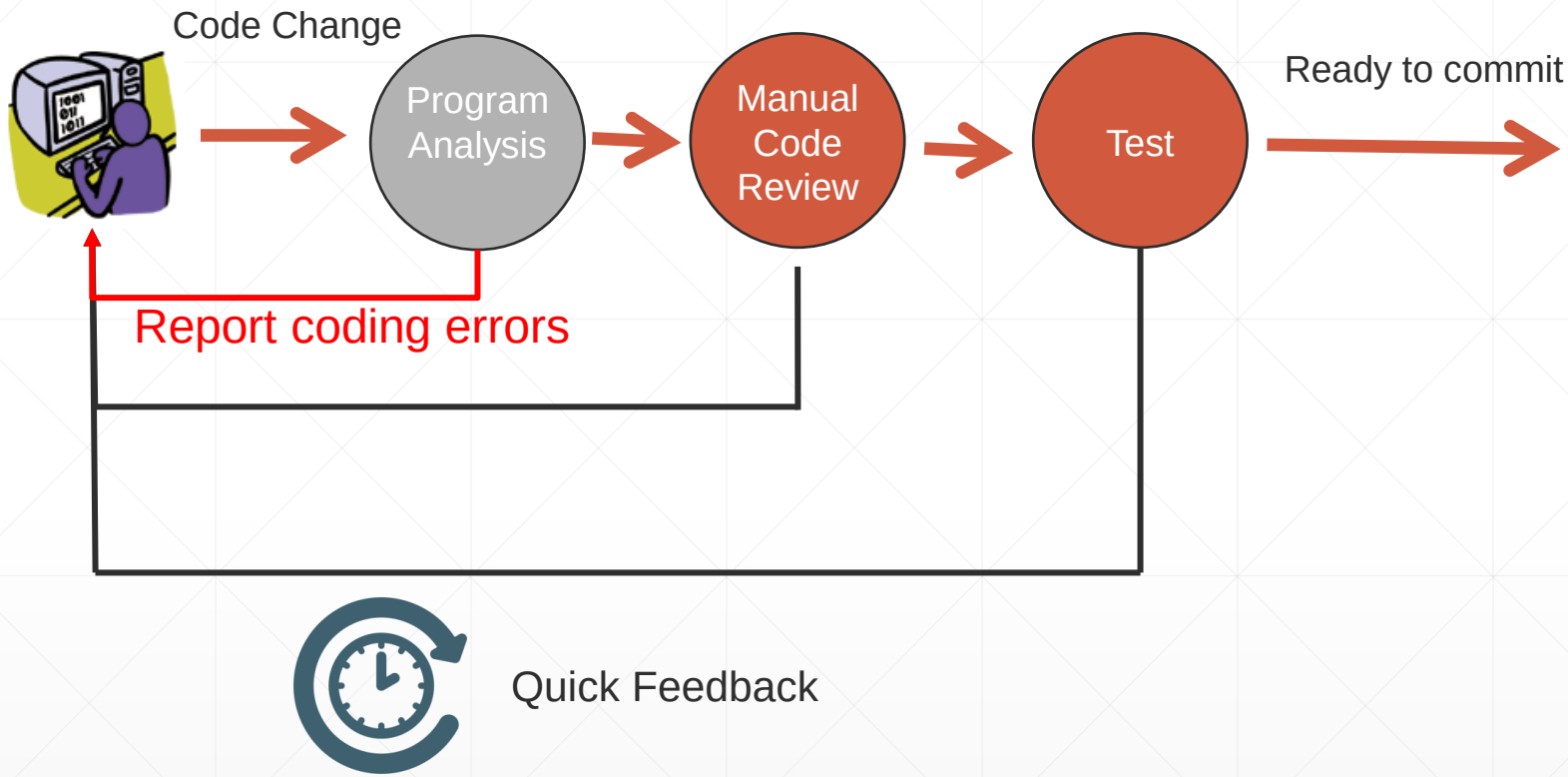
Finding Flaws in Source Code

- Code review
  - Code review is best used for subjective checks
  - Well defined flaws can be checked through automation
- Compiler diagnostics
- “Linting” checks, like Clang-tidy
- Static Analysis using Symbolic Execution – Examples include Clang SA, Coverity
  - Analysis Performed executing the code symbolically through simulation
  - Can address limitations of dynamic analysis
- Dynamic Analysis – Examples include UBSAN, TSAN, and ASAN
  - Analysis performed by instrumenting and running the code on a real target
  - Difficult to test the entire program, and all paths – dependent upon test cases
  - Memory resource limitations can be a problem for embedded systems

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/g++ 10, Clang, MSCV	Valgrind, gcc and clang sanitizers
False positives	No	Yes	Yes	Not likely, but possible
Inner Workings	Programmatic checks	Text/AST matching.	Symbolic Execution	Injection of Runtime checks
Compile and Runtime affects	None	Extra compile step	Extra compile step	Extra compile step, extended run times

Typical CI Loop with Automated Analysis

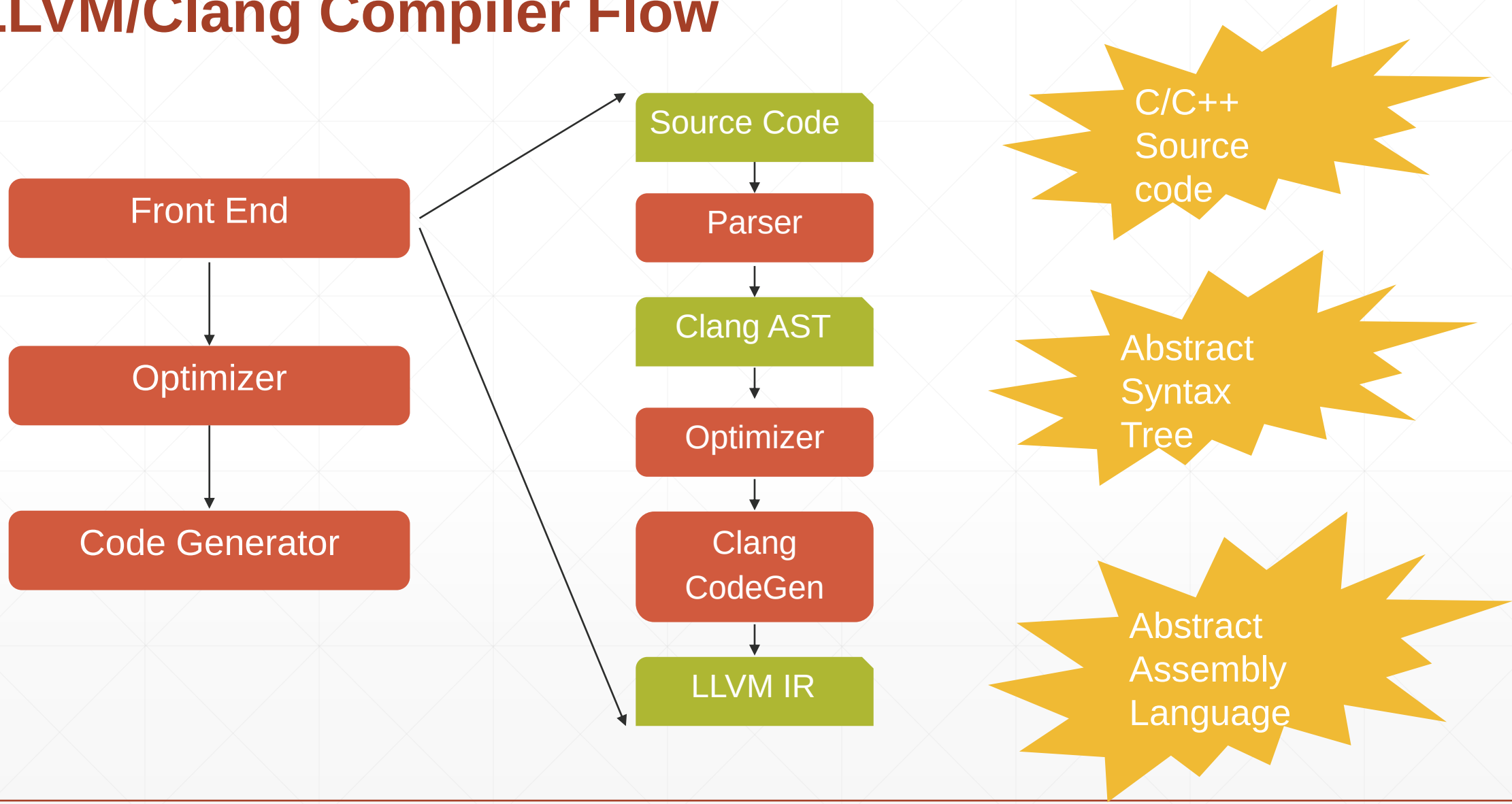


Static Analysis and Syntatic and Semantic Checks:
Find faults in code, not covered by test cases!
Can analyze properties of code that cannot be tested (coding style)!
Automates and offloads portions of manual code review
Tightens up CI loop for many issues

Clang-tidy & Static Analyzers – How?

- Clang-tidy uses AST Matchers
- Clang Static Analysis uses Symbolic Execution
- Both are useful, complement each other
- Both use the AST
- Contrast …

LLVM/Clang Compiler Flow



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

AST Matchers

```
#define ZERO 0
void test(int b)
{
  int a,c;
  double *d;
  switch (b){
    case 1:  a = b / 0; break;
    case 2:  a = b / ZERO; break;
    case 4:  c = b-4;
             a = b / c; break;
    };
}
```

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

Found

Found as all preprocessor statements are resolved.

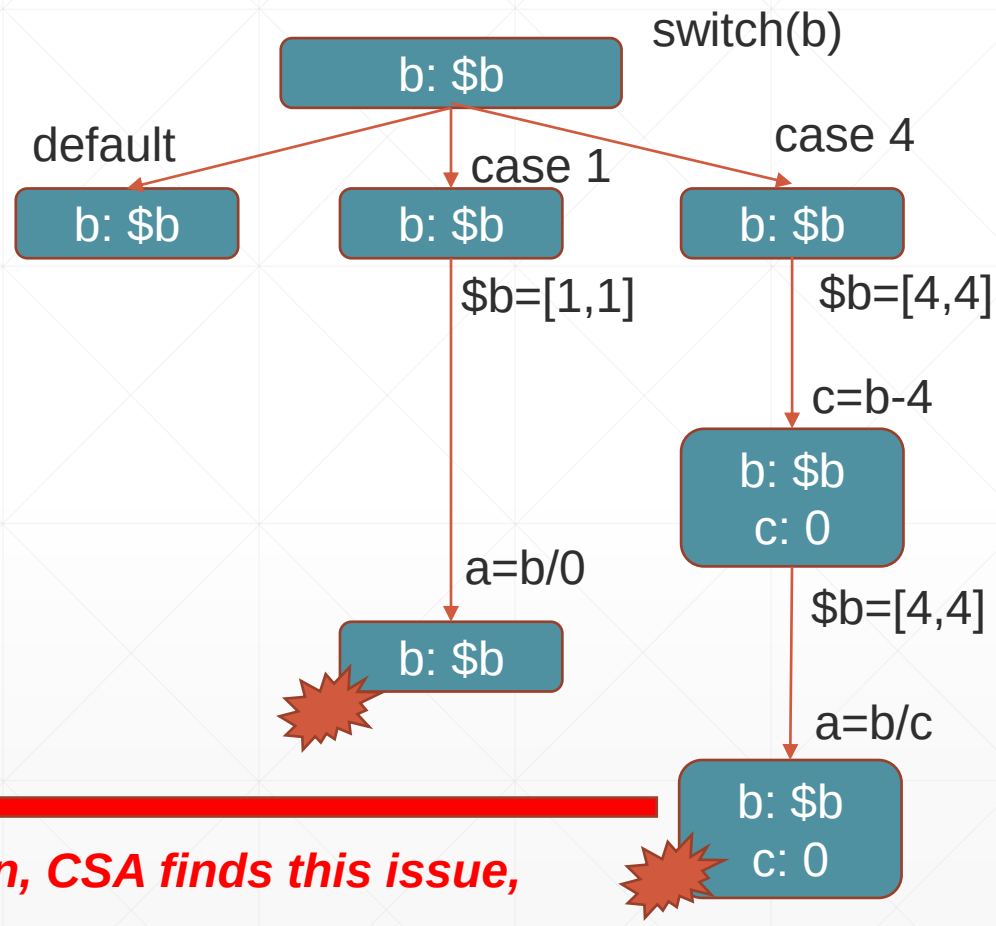
Not found by AST matcher as symbolic expressions are not evaluated.

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

Clang Static Analyzer – Symbolic Execution

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

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



By way of comparison, CSA finds this issue, AST matchers do not.

Clang-tidy

- Now with the 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

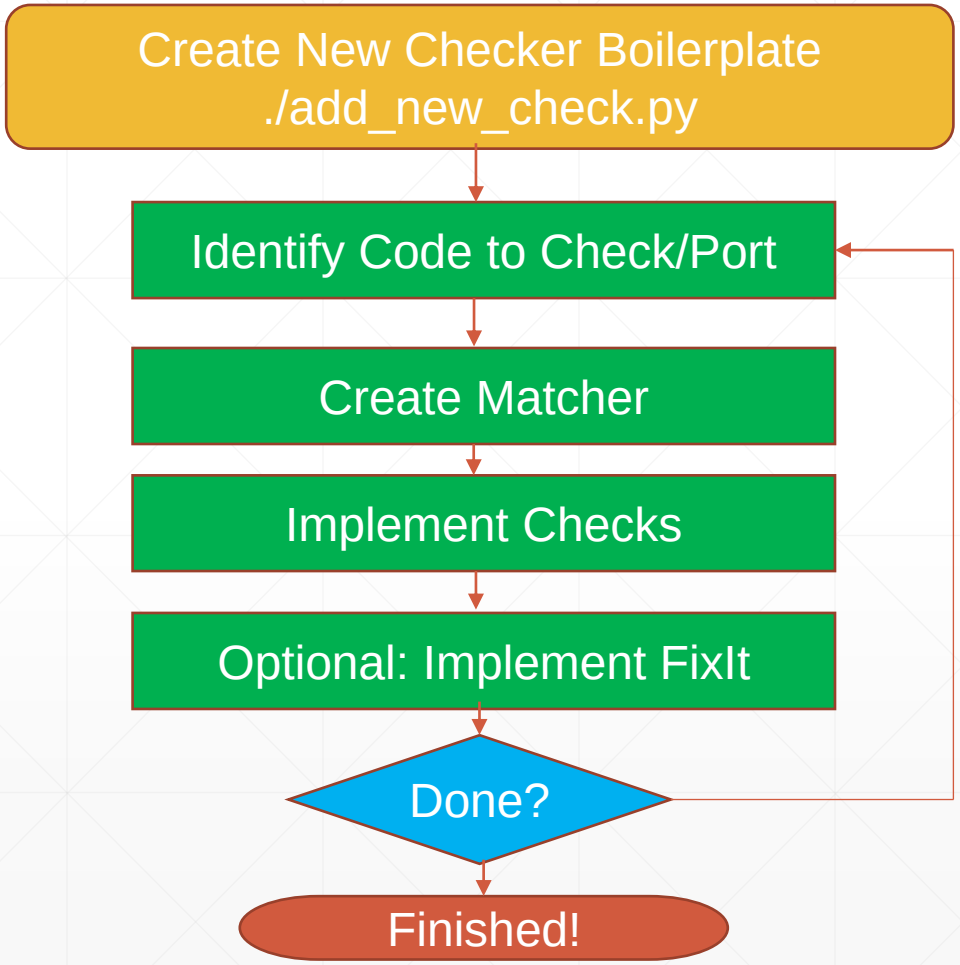
- Implement checks and changes that require semantic knowledge of the language
- Implement specialized checks for your organization
- Large scale refactoring
- Integration into your CI flow – Automated and repeatable

Clang-tidy Notes

- Not all checkers have “Fix”’s. See list of existing checkers to see …
- Why would not all checkers have fixes?
  - Some checks are not perfect, but “good enough”.
  - Provide warnings that a compiler has not implemented yet
  - Custom checks
- What’s the “--” at the end?
  - Says that we’re not using a compile_commands.json – more on that later.
- Can pass compiler commands to the compiler, example …
  - `clang-tidy --extra-arg="-DSWEET_DEFINE=1" --checks=-*,cppcoreguidelines-init-variables init.cpp  --`

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 though, right? ☺).
- 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 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 …
  - 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

- You'll want to match and change particular types of code
  - Functions, function calls, variables
  - Narrow items to port by name, type, return type, and parameters.
  - Matchers are a "Predicate Language". Parameters to matcher refine, or "narrow" the matches
- clang-query helps us to explore how to create particular matchers, e.g.,
  - functionDecl(), functionDecl(hasName("malloc"))
  - varDecl(), varDecl(matchesName("acme_"))
  - callExpr(), callExpr(callee(functionDecl(hasName("foo"))))
  - functionDecl(hasParameter(0, parmVarDecl(hasName("foo"))))
- These are readable but can be difficult to create. Clang-query helps us with creating these.

Extending clang-tidy …

- See https://clang.llvm.org/docs/LibASTMatchersReference.html
- Many existing matchers, and can be extended (subject for another day)
- There are 3 categories of matchers ….
  - Node Matchers
  - Narrowing Matchers
  - Traversal Matchers
- If you're overwhelmed so far – no worries ! This *is* difficult. Hang in there, we'll go through some simple examples to get started on your particular interests in this area…

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.
- We'll then push ahead with our tutorial to get to the "end to end" conclusion

Step 1: Replace “malloc”

- Most of the difficult work is done – we have a basic matcher expression we can use.
- From our exploration, you have references, perhaps some ideas of your own!
  - 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

clang::FunctionDecl

```
int someFunc(bool b, float f)
    ↑
    getLocation()
getBeginLoc()        getEndLoc()
```

- 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 …”

- Our manager has realized that we can work magic, so is asking for more. He’s promised to buy you a Starbuck’s latte if you can do this, instead of the company coffee ☺
- 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”

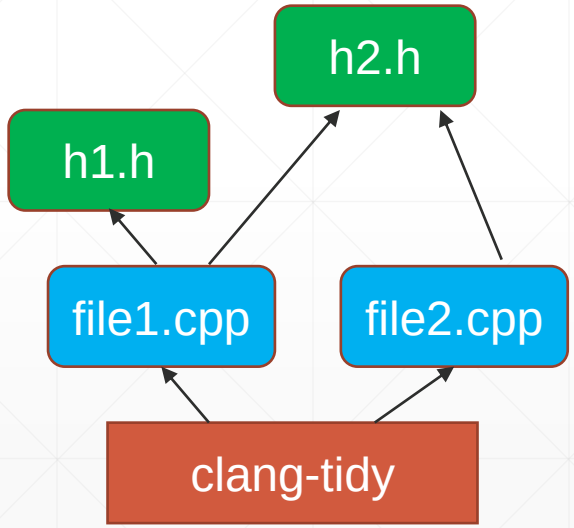

```
void ChangeMallocCheck::check(const MatchFinder::MatchResult &Result) {
  SmallString<64> NewArgument;
  const CallExpr *callExpr = Result.Nodes.getNodeAs<CallExpr>("malloc");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_zalloc() instead of malloc()")
      << 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 …

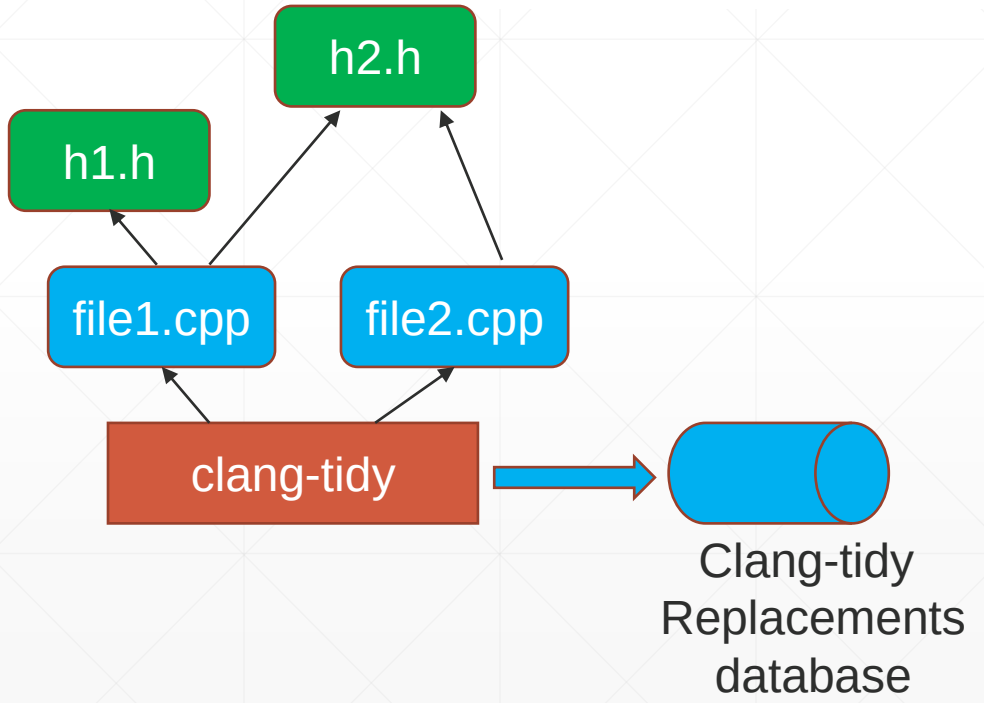
Clang-tidy for Projects

- 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!



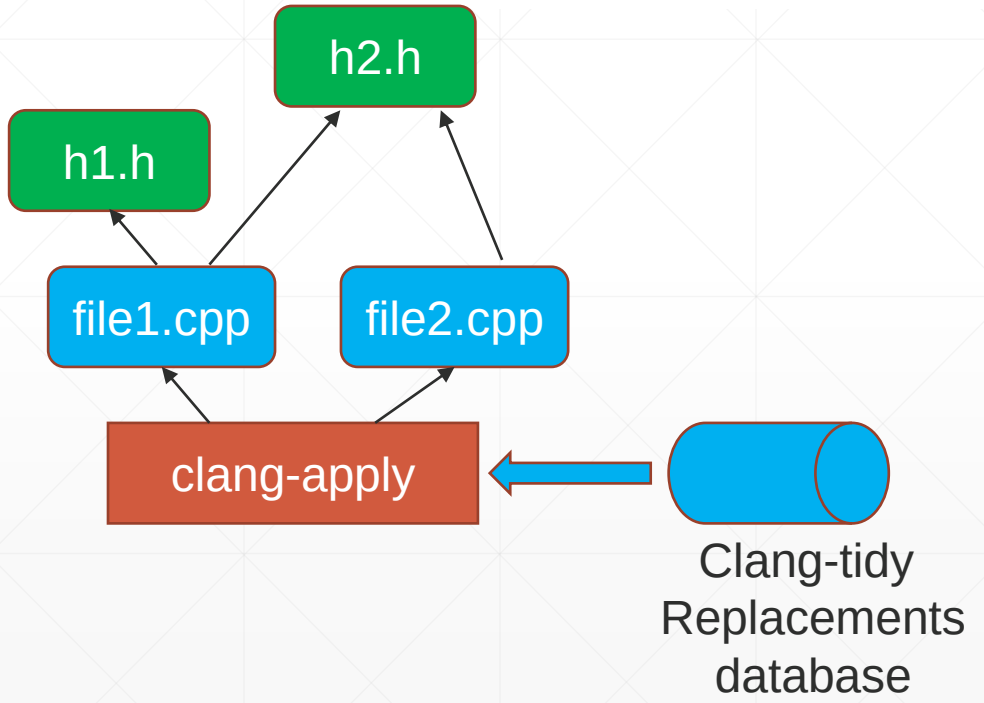
Clang-tidy for Projects

- 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.

Top: CMakeLists.txt

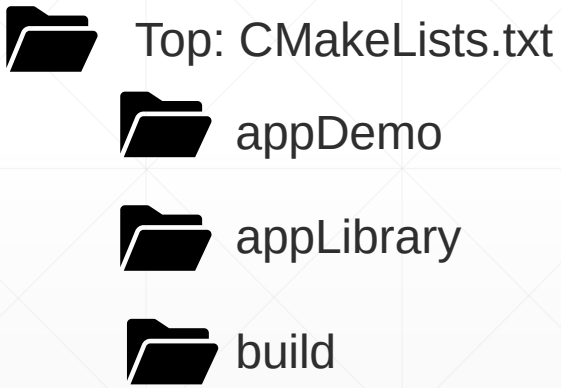
appDemo

appLibrary

build

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.
- Preferably 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, C++ topics
- Lot’s to explore!
  - Resources in the references
  - Try clang-query, using larger source examples. Explore AST nodes, get creative with AST matcher expressions.
  - Improve the LIT tests presented
  - Try adding your own category of checkers (not inserted into “misc”)
- Please post your questions, suggested topics in the meetup chat

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
- Austin LLVM Meetup Material - https://github.com/vabridgers/Austin-LLVM-Meetup-Material
- Stephen Kelly’s blog - https://devblogs.microsoft.com/cppblog/author/stkellyms/

Links

- https://www.meetup.com/Austin-LLVM-Meetup/
- https://www.capitalfactory.com/ Capital Factory
- https://docs.google.com/document/d/1bqms0I4u9-sNdHWfK5c76-iu1Q7DvZTHXIxmq74-8aA/edit LLVM Meetup suggestions
- https://berlincodeofconduct.org/ Berlin Code of Conduct
- https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf

Thank you for attending!