Abstract (Last updated June 10, 2019)

Abstract: In this talk, Michael Shah (“Mike”) will be presenting an introduction to Clang, the C/C++ Language Frontend Compiler. A discussion of what Clang is, who is using it, and why you might be interested in using Clang will be presented during the first part of the talk. The second part of the talk will show interactive examples of setting up a clang plugin. A distinction will be made of when to use libclang, clang plugins, and libtooling. Mike will also be presenting some steps on how to proceed further and what resources are available for working with Clang in the future.

Materials:

- Please bring a laptop with Clang/LLVM 8.0 setup if you want to follow along
- Otherwise materials will be posted to www.mshah.io

Resources:

- Downloading and setting up LLVM (make sure to include Clang!): http://llvm.org/docs/GettingStarted.html#checkout

Contact: mshah.475@gmail.com
Twitter: @MichaelShah

Terminology (Open in a second browser if you like)

- Clang - The C language family frontend
- AST - Abstract Syntax Tree

Introduction to Clang (Tutorial)

Mike Shah, Ph.D.
@MichaelShah | mshah.io

June 10, 2019

30-45 Minutes for talk (plenty of time for questions)

Demo Time! Right from the start!

- So you know what to pay attention to!
  - In case you (or maybe I) walked into the wrong room by accident!
  - (Or if you are deciding to commit to an hour long talk online in the *distant future*)
- For those attending this talk live
  - Take a moment to introduce yourself to someone next to you .

- demo1.sh - PrintFunctions

Who Am I?

by Mike Shah

- Currently an assistant teaching professor at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- www.mshah.io

This is an introduction to Clang

We have some specific goals

1. Figure out what is Clang
2. Understand how to obtain Clang
   a. (This can be a major bottleneck for students)
3. Navigate some of the tools and ‘lingo’ in Clang
4. Understand how to work with clang’s library architecture to build your own tools

Goals for Tomorrow

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future

Goals for Tomorrow

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future
- **As always--**Be able to run through these slides again with confidence and excitement!

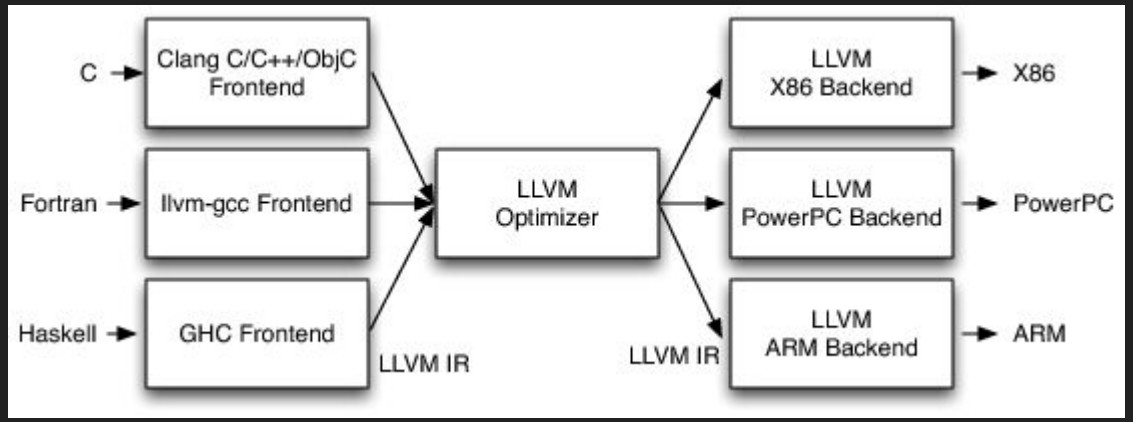
Slides and code are at the following location

www.mshah.io

What is Clang

A quick review on compilers

- Clang is the ‘frontend’ responsible for lexing and parsing an input language (e.g. ‘C++’) to make sure the source is valid
- Typically this involves building up a data structure (Abstract Syntax Tree) which represents the source, and then can be translated to the intermediate representation (IR)
  - Then a tool like LLVM takes over (for optimization and code generation in the backend)



Clang - https://clang.llvm.org/index.html (1/3)

- Clang is a frontend for several C-family languages
  - C and C++ being the most widely known
    - Supports C++11/14/17
      - (and already some C++20 features: https://clang.llvm.org/cxx_status.html)
  - (Objective C/C++, OpenCL, CUDA< and RenderScript are the other C-style languages actively developed)

Language	Year started	Created by (at)	Comments
E	1997	Mark S. Miller, Dan Bornstein (Electric Communities)	Designed with secure computing in mind, accomplished chiefly by strict adherence to the object-oriented computing model.
C	1969-1973	Dennis Ritchie (Bell Labs)	Was an enhancement of Ken Thompson's B language.
Ratfor	1974	Brian Kernighan (Bell Labs)	A hybrid of C and Fortran, implemented as a preprocessor for environments without easy access to C compilers.
C shell/tcsh	1978	Bill Joy (UC Berkeley)	Scripting language and standard Unix shell.
C++	1979	Bjarne Stroustrup (Bell Labs)	Named as "C with Classes" and renamed C++ in 1983; it began as a reimplementation of static object orientation in the tradition of Simula 67, and through standardization and wide use has grown to encompass generic programming as well as its original object-oriented roots.

Clang - https://clang.llvm.org/index.html (2/3)

- Clang is built to be a fast compiler competing with GCC, Intel, MSVC, etc.
- Clang provides helpful diagnostic warnings and errors
  - examples: https://clang.llvm.org/diagnostics.html
  - From personal experience, this is where Clang *appeared* to me to really gain ground.
    - (Often I would tell students to try both gcc and clang when debugging and clang gave better error messages--gcc now gives pretty good results too)

```
$ clang -fsyntax-only format-strings.c
format-strings.c:91:13: warning: '.*' specified field precision
  printf("%.*d");
```

Clang - https://clang.llvm.org/index.html (3/3)

- Clang is [illegible] SVC, etc.
- Clang pr[illegible]
  - exam[illegible]
  - From [illegible] ground.
    - [illegible] nd clang

What is it that makes Clang so great that programmers are paying attention to it?

```
$ clang -[illegible]
format-st[illegible] ecision
  printf("%.*d");
```

Clang’s Secret Sauce

What makes Clang so special?

- Well at least what made it popular to start
  - If you have a previous codebase in GCC, you can largely just replace “gcc” with “clang” and it should just work.
    - (I do not believe there is 100% compatibility, but it is close! (especially if you ignore gcc extensions))
  - So at the very least--we have two production grade open source compilers competing
    - (along with others like MSVC and Intel)
  - But there is more than popularity!
    - http://clang.llvm.org/features.html

sources: http://clang.llvm.org/features.html

What makes Clang so special -- the big features to me

- Clang compared to other compilers has a library based architecture
  - This means you only need to understand a few components of the infrastructure before diving in!
  - It also means clang can be used as a library for not just parsing C++ and generating intermediate code
    - This library can be used to generate documentation, work in IDEs, perform static analysis, etc.

End-User Features:

- Fast compiles and low memory use
- Expressive diagnostics
- GCC compatibility

Utility and Applications:

- Library based architecture
- Support diverse clients
- Integration with IDEs
- Use the LLVM 'BSD' License

Internal Design and Implementation:

- A real-world, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++
- Conformance with C/C++/ObjC and their variants

sources: http://clang.llvm.org/features.html

What makes Clang so special -- the big features to me

- Clang compared to other compilers has a library b
  - This m few co diving i
  - It also r for not interme
    - T documentation, work in IDEs, perform static analysis, etc.

Okay let’s get started!

End-User Features:

- Fast compiles and low memory use
- A real-world, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++
- Conformance with C/C++/ObjC and their variants

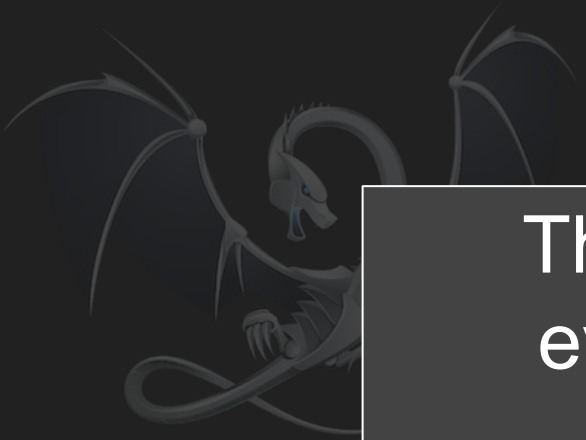
sources: http://clang.llvm.org/features.html

How to get Clang

(And all the tools)

I am actually going to run through this section very quick!

Use it as a reference for how to setup and run examples from this slide deck



The LLVM/Clang projects evolves at a good pace.

That is why you will want to know how to build from source to get the latest changes.

Where the instructions always will be

- http://clang.llvm.org/get_started.html

Getting Started: Building and Running Clang

This page gives you the shortest path to checking out Clang and demos a few options. This should get you up and running with the minimum of muss and fuss. If you like what you see, please consider getting involved with the Clang community. If you run into problems, please file bugs in LLVM Bugzilla.

Downloading Clang (1/2)

- (Note: If you have previously setup LLVM with clang, you are done--the following directions will grab clang as well though)
- For this talk, I am using and have tested the code with LLVM 8.0
- This tutorial is for an x86 based Ubuntu 18.04 machine
  - A similar process should work on Mac
    - (Windows users may need some different tools, I have not built LLVM on windows)
- Tools you will need
  - svn
  - Cmake
  - Make
  - A C compiler (Mine is GNU 5.4.0)

Downloading Clang (2/2)

- (Note: If you have previously setup LLVM with clang, you are done--the following directions will grab clang as well though)
- For this talk, I am using and have tested the code with LLVM 8.0
- This tutorial is for an x86 based Ubuntu 18.04 machine
  - A similar process should work on Mac
    - (Windows users may need some different tools, I have not built LLVM on windows)
- Tools you will need
  - svn
  - Cmake
  - Make
  - A C compiler (Mine is GNU 5.4.0)

In short--here is a copy and paste to install clang on your terminal
Official instructions were followed at the time http://clang.llvm.org/get_started.html (Contains up to date instructions)

```
1.  mkdir 6_10_19_clang # Wherever you want to store things
2.  cd 6_10_19_clang
3.  git clone https://github.com/llvm/llvm-project.git
4.  cd llvm-project
5.  mkdir build # (in-tree build is not supported)
6.  cd build
7.  cmake -DLLVM_ENABLE_PROJECTS=clang -G "Unix Makefiles"
    ../llvm
8.  make # or 'make -j 8' for a faster build
        i.  # Note: I had to use plain old 'make' at some point, because I
            was running out of memory for my builds on my machine when
            running in parallel.
```

In short--here is a copy and paste to install clang on your terminal
Official instructions were followed at the time http://clang.llvm.org/get_started.html (Contains up to date instructions)

1. mkdir ... things
2. cd 6_1...
3. git cl... it
4. cd llv...
5. mkdir ...
6. cd bui...
7. cmake ... files" ../llv...
8. make #...
   - # Note: I had to use plain old 'make' at some point, because I was running out of memory for my builds on my machine when running in parallel.
9.

It took about an hour to get setup on my i7 gaming laptop

How will we know it worked?

- Check your `build/bin` directory
- It should look something like this
- Note that for the examples, `clang++`, and other tools are referenced from here!
  - **I will create a folder called clang_examples in the bin directory for simplicity sake**
  - If your system already has clang++ installed from a package manager, it may have a different version!

```
clang                      llv
clang++                    llv
clang-8                    llv
clang-apply-replacements   llv
clang-change-namespace     llv
clang-check                llv
clang-cl                   llv
clang-cpp                  llv
clangd                     llv
clang-diff                 llv
clangd-indexer             llv
clang-doc                  llv
clang-extdef-mapping       llv
clang-format               llv
clang-import-test          llv
clang-include-fixer        llv
clang-interpreter          llv
clang-move                 llv
clang-offload-bundler      llv
clang-query                llv
clang-refactor             llv
clang-rename               llv
clang-reorder-fields       llv
clang-tblgen               llv
clang-tidy                 llv
count                      llv
dexp                       llv
```

How will we know it worked?

- You might additionally run `clang++` to make sure you are using the same version
  - You should see version 8.0.0 for this example

```
mike:bin$ ./clang++ -v
clang version 8.0.0 (tags/RELEASE_800/final 362351)
Target: x86_64-unknown-linux-gnu
Thread model: posix
```

Assumption: We all have a working clang at this point

Running Clang Tools

Clang Tooling Support

- As mentioned, the real power of clang, is the library architecture and the tools built around it.
- The tooling can allow us to build tools that make programming easier for us.
- One class of these tools are ‘syntax checkers’

Syntax Checkers

Goal of Syntax Checkers: Improve the style and code health in large-scale systems

Pros: Readability, maintainability (uniformity), Simplicity
Cons: Additional step in build stage

Building our own tools

- Clang gives us many built in tools already, but we really want to be building our own for our needs.
  - (We will look at a few clang tools in a moment)
- There are a few options
  - LibClang
  - Clang Plugins
  - LibTooling
- https://clang.llvm.org/docs/Tooling.html

Building our own tools - LibClang

- In Short--the first choice for working with clang to build cool tools.

LibClang is a stable high level C interface to clang. When in doubt LibClang is probably the interface you want to use. Consider the other interfaces only when you have a good reason not to use LibClang.

Canonical examples of when to use LibClang:

- Xcode
- Clang Python Bindings

Use LibClang when you...:

- want to interface with clang from other languages than C++
- need a stable interface that takes care to be backwards compatible
- want powerful high-level abstractions, like iterating through an AST with a cursor, and don't want to learn all the nitty gritty details of Clang's AST.

Do not use LibClang when you...:

- want full control over the Clang AST

Building our own tools - Clang Plugins

- In Short--lets us build tools that integrate with our build environment, we run them with the compiler.

Clang Plugins

Clang Plugins allow you to run additional actions on the AST as part of a compilation. Plugins are dynamic libraries that are loaded at
they're easy to integrate into your build environment.

Canonical examples of when to use Clang Plugins:

- special lint-style warnings or errors for your project
- creating additional build artifacts from a single compile step

Use Clang Plugins when you…:

- need your tool to rerun if any of the dependencies change
- want your tool to make or break a build
- need full control over the Clang AST

Do not use Clang Plugins when you…:

- want to run tools outside of your build environment
- want full control on how Clang is set up, including mapping of in-memory virtual files
- need to run over a specific subset of files in your project which is not necessarily related to any changes which would trigger rebuil

Building our own tools - LibTooling

- In Short--helps us build standalone tools

LibTooling ¶

LibTooling is a C++ interface aimed at writing standalone tools, as well as integrating into services that run clang tools. Canonical examples of w
LibTooling:

- a simple syntax checker
- refactoring tools

Use LibTooling when you…:

- want to run tools over a single file, or a specific subset of files, independently of the build system
- want full control over the Clang AST
- want to share code with Clang Plugins

Do not use LibTooling when you…:

- want to run as part of the build triggered by dependency changes
- want a stable interface so you don't need to change your code when the AST API changes
- want high level abstractions like cursors and code completion out of the box
- do not want to write your tools in C++

Building our own tools - LibTooling (1/2)

- There are some popular tools built on top of LibTooling listed on the help page
  - clang-check - A syntax checking tool
  - clang-fixit - An automatic compile-time error repair tool
  - clang-format - A code refactoring tool

Building our own tools - LibTooling (2/2)

- There are some popular tools built on top of LibTooling listed on the help page
  - clang-check - A syntax checking tool <-- Let’s check out clang-check
  - clang-fixit - An automatic compile-time error repair tool
  - clang-format - A code refactoring tool

clang-check example

- Can run: `./../clang-check hello.cpp`
- The following syntax error will be reported.
  - Note the nice colored output, and a cursor showing the missing semicolon

```
1  #include <iostream>
2
3  void printHello(){
4      std::cout << "hello\n"
5  }
6
7  int main(){
8
9      printHello();
10
11     return 0;
12 }
```

```
mike:clang_examples$ ./../clang-check hello.cpp
warning: unknown warning option '-Wno-maybe-uninitialized'; did you mean
      '-Wno-uninitialized'? [-Wunknown-warning-option]
/home/mike/Desktop/LLVM_6_3_19/build/bin/clang_examples/hello.cpp:4:27: error:
      expected ';' after expression
    std::cout << "hello\n"
                          ^
                          ;
1 warning and 1 error generated.
Error while processing /home/mike/Desktop/LLVM_6_3_19/build/bin/clang_examples/hello.
cpp.
```

clang-check -help

- Several handy options

```
clang-check options:

  -analyze                    - Run static analysis engine
  -ast-dump                   - Build ASTs and then debug dump them
  -ast-dump-filter=<string>   - Use with -ast-dump or -ast-print to dump/print only AST declaration nodes having a certain
 substring in a qualified name. Use -ast-list to list all filterable declaration node names.
  -ast-list                   - Build ASTs and print the list of declaration node qualified names
  -ast-print                  - Build ASTs and then pretty-print them
  -extra-arg=<string>         - Additional argument to append to the compiler command line
  -extra-arg-before=<string> - Additional argument to prepend to the compiler command line
  -fix-what-you-can           - Apply fix-it advice even in the presence of unfixable errors
  -fixit                      - Apply fix-it advice to the input source
  -p=<string>                 - Build path
```

```
ww mike:clang_examples$ ./../clang-check -ast-print hello.cpp
```

clang-check -ast-print

- Can actually get the whole pretty printed version of our file
  - (Including the appended '-E' information from the preprocessor.)

```
        explicit basic_iostream(basic_streambuf<wchar_t, std::char_traits<wchar_t> > *__sb);
        virtual ~basic_iostream<wchar_t>();
    protected:
        basic_iostream();
        basic_iostream(const std::basic_iostream<wchar_t> &) = delete
        basic_iostream(std::basic_iostream<wchar_t> &&__rhs);
        std::basic_iostream<wchar_t> &operator=(const std::basic_iostream<wchar_t> &) = delete
        std::basic_iostream<wchar_t> &operator=(std::basic_iostream<wchar_t> &&__rhs);
        void swap(std::basic_iostream<wchar_t> &__rhs);
    };
}
namespace std {
    extern std::istream cin;
    extern std::ostream cout;
    extern std::ostream cerr;
    extern std::ostream clog;
    extern std::wistream wcin;
    extern std::wostream wcout;
    extern std::wostream wcerr;
    extern std::wostream wclog;
    static ios_base::Init __ioinit;
}
void printHello() {
    std::cout << "hello\n";
}
int main() {
    printHello();
    return 0;
}
```

```
mike:clang_examples$ ./../clang-check -ast-list hello.cpp
```

clang-check -ast-list

- Could instead list every individual token
  - We can start to see the frontend of our compiler working at this point.
- However--to do a more general purpose analysis, we probably want to be working on the Abstract Syntax Tree instead

```
__streambuf_type
__int_type
__ctype_type
__ct
__eof
__sb
__c
std::basic_istream
std::basic_iostream
std::basic_istream
std::basic_iostream
std
std::cin
std::cout
std::cerr
std::clog
std::wcin
std::wcout
std::wcerr
std::wclog
std::__ioinit
printHello
main
1 warning generated.
```

```
mike:clang_examples$ ./../clang-check -ast-list hello.cpp
```

clang-check -ast-list

- Could instead list every individual token
  - We can start to see the frontend of our compiler working at this point.
- However--to do a more general purpose analysis, we probably want to be working on the Abstract Syntax Tree instead

```
__streambuf_type
__int_type
__ctype_type
__ct
__eof
__sb
__c
std::basic_istream
std::basic_iostream
std::basic_istream
std::basic_iostream
std
std::cin
std::cout
std::cerr
std::clog
std::wcin
std::wcout
std::wcerr
std::wclog
std::__ioinit
printHello
main
1 warning generated.
```

AST

(Abstract Syntax Tree)

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- Here is our test program

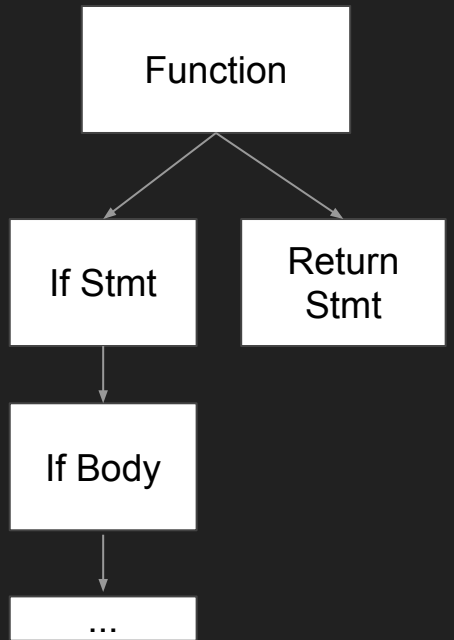
```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- The AST will roughly give us some ‘Tree-like’ form like such:




```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- Here is the AST from our test program
  - Let’s look at a few of the pieces

```
TranslationUnitDecl 0x5634527151e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x563452715ac0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x563452715780 '__int128'
|-TypedefDecl 0x563452715b28 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5634527157a0 'unsigned __int128'
|-TypedefDecl 0x563452715e58 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstant
String_tag'
| `-RecordType 0x563452715c00 '__NSConstantString_tag'
|     `-CXXRecord 0x563452715b78 '__NSConstantString_tag'
|-TypedefDecl 0x563452715ef0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x563452715eb0 'char *'
|   `-BuiltinType 0x563452715280 'char'
|-TypedefDecl 0x563452751578 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag
 [1]'
| `-ConstantArrayType 0x563452751520 '__va_list_tag [1]' 1
|   `-RecordType 0x563452715fd0 '__va_list_tag'
|     `-CXXRecord 0x563452715f40 '__va_list_tag'
|-FunctionDecl 0x563452751618 <hello.cpp:3:1, col:24> col:13 used printHello 'void ()' extern
`-FunctionDecl 0x563452751740 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x563452751970 <col:11, line:10:1>
    |-IfStmt 0x563452751928 <line:6:5, line:8:5>
    | |-ImplicitCastExpr 0x563452751828 <line:6:8> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x563452751808 <col:8> 'int' 1
    | `-CompoundStmt 0x563452751910 <col:10, line:8:5>
    |   `-CallExpr 0x5634527518f0 <line:7:9, col:20> 'void'
    |     `-ImplicitCastExpr 0x5634527518d8 <col:9> 'void (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr 0x563452751888 <col:9> 'void ()' lvalue Function 0x563452751618 'printHello'
 'void ()'
    `-ReturnStmt 0x563452751960 <line:9:5, col:12>
      `-IntegerLiteral 0x563452751940 <col:12> 'int' 0
```

```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- ‘Decl’
  - Some definition from the source code
    - e.g. a variable declaration

```
TranslationUnitDecl 0x5634527151e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x563452715ac0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x563452715780 '__int128'
|-TypedefDecl 0x563452715b28 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5634527157a0 'unsigned __int128'
|-TypedefDecl 0x563452715e58 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstant
String_tag'
| `-RecordType 0x563452715c00 '__NSConstantString_tag'
|   `-CXXRecord 0x563452715b78 '__NSConstantString_tag'
|-TypedefDecl 0x563452715ef0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x563452715eb0 'char *'
|   `-BuiltinType 0x563452715280 'char'
|-TypedefDecl 0x563452751578 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag
[1]'
| `-ConstantArrayType 0x563452751520 '__va_list_tag [1]' 1
|   `-RecordType 0x563452715fd0 '__va_list_tag'
|     `-CXXRecord 0x563452715f40 '__va_list_tag'
|-FunctionDecl 0x563452751618 <hello.cpp:3:1, col:24> col:13 used printHello 'void ()' extern
`-FunctionDecl 0x563452751740 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x563452751970 <col:11, line:10:1>
    |-IfStmt 0x563452751928 <line:6:5, line:8:5>
    | |-ImplicitCastExpr 0x563452751828 <line:6:8> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x563452751808 <col:8> 'int' 1
    | `-CompoundStmt 0x563452751910 <col:10, line:8:5>
    |   `-CallExpr 0x5634527518f0 <line:7:9, col:20> 'void'
    |     `-ImplicitCastExpr 0x5634527518d8 <col:9> 'void (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr 0x563452751888 <col:9> 'void ()' lvalue Function 0x563452751618 'printHello'
'void ()'
    `-ReturnStmt 0x563452751960 <line:9:5, col:12>
      `-IntegerLiteral 0x563452751940 <col:12> 'int' 0
```

```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- Stmt
  - Some Compound Statement binary operator, or other general operation (e.g. ReturnStmt)

```
TranslationUnitDecl 0x5634527151e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x563452715ac0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x563452715780 '__int128'
|-TypedefDecl 0x563452715b28 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5634527157a0 'unsigned __int128'
|-TypedefDecl 0x563452715e58 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstant
String_tag'
| `-RecordType 0x563452715c00 '__NSConstantString_tag'
|   `-CXXRecord 0x563452715b78 '__NSConstantString_tag'
|-TypedefDecl 0x563452715ef0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x563452715eb0 'char *'
|   `-BuiltinType 0x563452715280 'char'
|-TypedefDecl 0x563452751578 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag
 [1]'
| `-ConstantArrayType 0x563452751520 '__va_list_tag [1]' 1
|   `-RecordType 0x563452715fd0 '__va_list_tag'
|     `-CXXRecord 0x563452715f40 '__va_list_tag'
|-FunctionDecl 0x563452751618 <hello.cpp:3:1, col:24> col:13 used printHello 'void ()' extern
`-FunctionDecl 0x563452751740 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x563452751970 <col:11, line:10:1>
    |-IfStmt 0x563452751928 <line:6:5, line:8:5>
    | |-ImplicitCastExpr 0x563452751828 <line:6:8> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x563452751808 <col:8> 'int' 1
    | `-CompoundStmt 0x563452751910 <col:10, line:8:5>
    |   `-CallExpr 0x5634527518f0 <line:7:9, col:20> 'void'
    |     `-ImplicitCastExpr 0x5634527518d8 <col:9> 'void (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr 0x563452751888 <col:9> 'void ()' lvalue Function 0x563452751618 'printHello'
 'void ()'
    `-ReturnStmt 0x563452751960 <line:9:5, col:12>
      `-IntegerLiteral 0x563452751940 <col:12> 'int' 0
```

```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- Expr
  - Something that returns or evaluates to a value

```
TranslationUnitDecl 0x5634527151e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x563452715ac0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x563452715780 '__int128'
|-TypedefDecl 0x563452715b28 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5634527157a0 'unsigned __int128'
|-TypedefDecl 0x563452715e58 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstant
String_tag'
| `-RecordType 0x563452715c00 '__NSConstantString_tag'
|     `-CXXRecord 0x563452715b78 '__NSConstantString_tag'
|-TypedefDecl 0x563452715ef0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x563452715eb0 'char *'
|   `-BuiltinType 0x563452715280 'char'
|-TypedefDecl 0x563452751578 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag
 [1]'
| `-ConstantArrayType 0x563452751520 '__va_list_tag [1]' 1
|   `-RecordType 0x563452715fd0 '__va_list_tag'
|     `-CXXRecord 0x563452715f40 '__va_list_tag'
|-FunctionDecl 0x563452751618 <hello.cpp:3:1, col:24> col:13 used printHello 'void ()' extern
`-FunctionDecl 0x563452751740 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x563452751970 <col:11, line:10:1>
    |-IfStmt 0x563452751928 <line:6:5, line:8:5>
    | |-ImplicitCastExpr 0x563452751828 <line:6:8> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x563452751808 <col:8> 'int' 1
    | `-CompoundStmt 0x563452751910 <col:10, line:8:5>
    |   `-CallExpr 0x5634527518f0 <line:7:9, col:20> 'void'
    |     `-ImplicitCastExpr 0x5634527518d8 <col:9> 'void (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr 0x563452751888 <col:9> 'void ()' lvalue Function 0x563452751618 'printHello'
 'void ()'
    `-ReturnStmt 0x563452751960 <line:9:5, col:12>
      `-IntegerLiteral 0x563452751940 <col:12> 'int' 0
```

```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- IfStmt / ForStmt / DoStmt/ etc.
  - Basic control flow structures

```
TranslationUnitDecl 0x5634527151e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x563452715ac0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x563452715780 '__int128'
|-TypedefDecl 0x563452715b28 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5634527157a0 'unsigned __int128'
|-TypedefDecl 0x563452715e58 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstant
String_tag'
| `-RecordType 0x563452715c00 '__NSConstantString_tag'
|     `-CXXRecord 0x563452715b78 '__NSConstantString_tag'
|-TypedefDecl 0x563452715ef0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x563452715eb0 'char *'
|   `-BuiltinType 0x563452715280 'char'
|-TypedefDecl 0x563452751578 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag
 [1]'
| `-ConstantArrayType 0x563452751520 '__va_list_tag [1]' 1
|   `-RecordType 0x563452715fd0 '__va_list_tag'
|     `-CXXRecord 0x563452715f40 '__va_list_tag'
|-FunctionDecl 0x563452751618 <hello.cpp:3:1, col:24> col:13 used printHello 'void ()' extern
`-FunctionDecl 0x563452751740 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x563452751970 <col:11, line:10:1>
    |-IfStmt 0x563452751928 <line:6:5, line:8:5>
    | |-ImplicitCastExpr 0x563452751828 <line:6:8> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x563452751808 <col:8> 'int' 1
    | `-CompoundStmt 0x563452751910 <col:10, line:8:5>
    |   `-CallExpr 0x5634527518f0 <line:7:9, col:20> 'void'
    |     `-ImplicitCastExpr 0x5634527518d8 <col:9> 'void (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr 0x563452751888 <col:9> 'void ()' lvalue Function 0x563452751618 'printHello'
 'void ()'
    `-ReturnStmt 0x563452751960 <line:9:5, col:12>
      `-IntegerLiteral 0x563452751940 <col:12> 'int' 0
```

```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- You will notice there is also source code location information

```
TranslationUnitDecl 0x5634527151e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x563452715ac0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x563452715780 '__int128'
|-TypedefDecl 0x563452715b28 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5634527157a0 'unsigned __int128'
|-TypedefDecl 0x563452715e58 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstant
String_tag'
| `-RecordType 0x563452715c00 '__NSConstantString_tag'
|     `-CXXRecord 0x563452715b78 '__NSConstantString_tag'
|-TypedefDecl 0x563452715ef0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x563452715eb0 'char *'
|   `-BuiltinType 0x563452715280 'char'
|-TypedefDecl 0x563452751578 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag
 [1]'
| `-ConstantArrayType 0x563452751520 '__va_list_tag [1]' 1
|   `-RecordType 0x563452715fd0 '__va_list_tag'
|     `-CXXRecord 0x563452715f40 '__va_list_tag'
|-FunctionDecl 0x563452751618 <hello.cpp:3:1, col:24> col:13 used printHello 'void ()' extern
`-FunctionDecl 0x563452751740 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x563452751970 <col:11, line:10:1>
    |-IfStmt 0x563452751928 <line:6:5, line:8:5>
    | |-ImplicitCastExpr 0x563452751828 <line:6:8> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x563452751808 <col:8> 'int' 1
    | `-CompoundStmt 0x563452751910 <col:10, line:8:5>
    |   `-CallExpr 0x5634527518f0 <line:7:9, col:20> 'void'
    |     `-ImplicitCastExpr 0x5634527518d8 <col:9> 'void (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr 0x563452751888 <col:9> 'void ()' lvalue Function 0x563452751618 'printHello'
 'void ()'
    `-ReturnStmt 0x563452751960 <line:9:5, col:12>
      `-IntegerLiteral 0x563452751940 <col:12> 'int' 0
```

```
// Removed preprocessor to make analysis more
// doable in a slide
extern void printHello();

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

```
mike:clang_examples$ ./../clang -Xclang -ast-dump hello.cpp -fsyntax-only
```

Dumping the Abstract Syntax Tree with Clang

- You will
  source c
  informati

```
__int128_t '__int128'
__uint128_t 'unsigned __int128'
__NSConstantString '__NSConstant
__builtin_ms_va_list 'char *'
__builtin_va_list '__va_list_tag
intHello 'void ()' extern
Boolean>
ctionToPointerDecay>
nction 0x563452751618 'printHello'
```

So what we'll want to do is navigate the AST in 'clangs language'

High Level Goal: Find patterns in AST

```
// Removed prep
// doable in a
extern void pri

int main(){
    if(1){
        printHello();
    }
    return 0;
}
```

Clang Query

(Helpful tool for matching AST)

ClangQuery

- clang-query is a handy tool that can be used to find nodes in AST what we would want to match.
- Reference for AST matchers:
  - https://manu343726.github.io/2017-02-11-writing-ast-matchers-for-libclang/
  - https://clang.llvm.org/docs/LibASTMatchersReference.html
  - (Handy tutorial: https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools)

ClangQuery - Example

- In this example, we can query and find all of the patterns that ‘match’ FunctionDecl() (i.e. function declarations)

```
mike:clang_examples$ ./../clang-query hello.cpp --
clang-query> match functionDecl()

Match #1:

/home/mike/Desktop/LLVM_6_3_19/build/bin/clang_examples/hello.cpp:3:1: note: "root" binds here
extern void printHello();
^~~~~~~~~~~~~~~~~~~~~~~~

Match #2:

/home/mike/Desktop/LLVM_6_3_19/build/bin/clang_examples/hello.cpp:5:1: note: "root" binds here
int main(){
^~~~~~~~~~~
2 matches.
clang-query>
```

A look at a first clang tool

(Printing all of the functions--this time from clang)

First Example

- The *easiest* starting point is to modify the PrintFunctionNames example.
  - (A sure classic in all examples!)
- The source is located at:
  - your_home_location/source/llvm/tools/clang/examples/PrintFunctionNames
  - PrintFunctionNames.cpp
- Figuring out just how to run a clang plugin is probably the next step.

```
mples$ ./../clang -cc1 -load ./../../lib/PrintFunctionNames.so -plugin print-fns hello.cpp
```

Running libPrintFunctionNames.so

- (Within: build/bin/clang_examples)
- `./../clang -cc1 -load ./../../lib/PrintFunctionNames.so -plugin print-fns hello.cpp`
  - or
- `./../clang++ -cc1 -load ./../../lib/PrintFunctionNames.so -plugin print-fns hello.cpp`

(See llvm/tools/clang/examples/PrintFunctionNames/README.txt for more information on running plugin)

```
mples$ ./../clang -cc1 -load ./../../lib/PrintFunctionNames.so -plugin print-fns hello.cpp
```

Running libPrintFunctionNames.so

- (Within: build/bin/clang_examples)
- `./../clang -cc1 -load ./../../lib/PrintFunctionNames.so -plugin print-fns hello.cpp`
- This code is loading a plug-in that will run during the compilation phase and generate some information for us
  - (Or potentially find an error and cause compilation to fail)

(See llvm/tools/clang/examples/PrintFunctionNames/README.txt for more information on running plugin)

```
mples$ ./../clang -cc1 -load ./../../lib/PrintFunctionNames.so -plugin print-fns hello.cpp
```

Updating libPrintFunctionNames.so

- As we make more changes to the PrintFunctionNames example, we can rebuild them in the ‘build’ directory

```
mike:PrintFunctionNames$ pwd
/home/mike/Desktop/LLVM_6_3_19/build/tools/clang/examples/PrintFunctionNames
mike:PrintFunctionNames$ make
```

Pieces of PrintFunctionNames

- PrintFunctonNames inherits from something called a PluginASTAction
  - Essentially we are building a tool that can take in command line options
    - https://clang.llvm.org/docs/ClangPlugins.html

```
class PrintFunctionNamesAction : public PluginASTAction {
  std::set<std::string> ParsedTemplates;
protected:
  std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
                                                 llvm::StringRef) override {
    return llvm::make_unique<PrintFunctionsConsumer>(CI, ParsedTemplates);
  }
```

Pieces of PrintFunctionNames

- The ASTConsumer is the interface that lets us do ‘generic things’ to an AST.
  - https://clang.llvm.org/docs/RAVFrontendAction.html

```
class PrintFunctionNamesAction : public PluginASTAction {
  std::set<std::string> ParsedTemplates;
protected:
  std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
                                                 llvm::StringRef) override {
    return llvm::make_unique<PrintFunctionsConsumer>(CI, ParsedTemplates);
  }
```

Pieces of PrintFunctionNames

- The ASTConsumer is the interface that lets us do ‘generic things’ to an AST.
  - https://clang.llvm.org/docs/RAVFrontendAction.html

```
26 class PrintFunctionsConsumer : public ASTConsumer {
27   CompilerInstance &Instance;
28   std::set<std::string> ParsedTemplates;
29
30 public:
31   PrintFunctionsConsumer(CompilerInstance &Instance,
```

Pieces of PrintFunctionNames

- A Part of the ASTConsumer that we can override is to `HandleTopLevelDecl`
  - i.e. when we declare something, print out what we have declared.
  - And this is how we get our function names.

```
35  bool HandleTopLevelDecl(DeclGroupRef DG) override {
36    for (DeclGroupRef::iterator i = DG.begin(), e = DG.end(); i != e; ++i) {
37      const Decl *D = *i;
38      if (const NamedDecl *ND = dyn_cast<NamedDecl>(D))
39        llvm::errs() << "top-level-decl: \"" << ND->getNameAsString() << "\"\n";
40    }
41
42    return true;
43  }
```

Pieces of PrintFunctionNames

- The ASTConsumer allows us to handle lots of different cases
  - https://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html

virtual bool	**HandleTopLevelDecl** (**DeclGroupRef** D) HandleTopLevelDecl - Handle the specified top-level declaration. More...
virtual void	**HandleInlineFunctionDefinition** (**FunctionDecl** *D) This callback is invoked each time an inline (method or friend) function definition in a class is completed. More...
virtual void	**HandleInterestingDecl** (**DeclGroupRef** D) HandleInterestingDecl - Handle the specified interesting declaration. More...
virtual void	**HandleTranslationUnit** (**ASTContext** &Ctx) HandleTranslationUnit - This method is called when the ASTs for entire translation unit have been parsed. More
virtual void	**HandleTagDeclDefinition** (**TagDecl** *D) HandleTagDeclDefinition - This callback is invoked each time a **TagDecl** (e.g. More...
virtual void	**HandleTagDeclRequiredDefinition** (const **TagDecl** *D) This callback is invoked the first time each **TagDecl** is required to be complete. More...
virtual void	**HandleCXXImplicitFunctionInstantiation** (**FunctionDecl** *D) Invoked when a function is implicitly instantiated. More...
virtual void	**HandleTopLevelDeclInObjCContainer** (**DeclGroupRef** D) Handle the specified top-level declaration that occurred inside and ObjC container. More...

Pieces of PrintFunctionNames

- Within our ASTConsumer we can then start writing ASTMatchers to get better queries
  - Some examples found here: https://s3.amazonaws.com/connect.linaro.org/yvr18/presentations/yvr18-223.pdf
  - We could even rewrite portions of our code--this is where you can stylize things with rewriters
    - https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html

Diving in

(Interactive portion? :))

Several other examples

- Clang provides several other examples
  - AnnotateFunctions
  - clang-interpreter
  - analyzer-plugin
- Things to try:
  - Try running each plugin
  - Try modifying somet behavior
    - either adding onto the class to gather metrics
    - Or changing what part of the AST you are analyzing

More on Clang API

- The clang API is quite rich--so there may be more things we want to do
  - Driver - Handles ‘Arg’ or allows us to pass in arguments.
    - https://clang.llvm.org/docs/DriverInternals.html
  - Lexer - Retrieves tokens
    - https://clang.llvm.org/doxygen/classclang_1_1Lexer.html
  - Parse - begin to create the AST from source code
    - Example: https://shaharmike.com/cpp/libclang/
    - https://clang.llvm.org/doxygen/classclang_1_1Parser.html
  - CodeGen - create IR from clang
    - https://clang.llvm.org/doxygen/namespaceclang_1_1CodeGen.html
  - AST - contains actual AST elements
    - Sample+Video: https://clang.llvm.org/docs/IntroductionToTheClangAST.html

Going Further (Challenges/Project Ideas)

Time permitting:

- Easy
  - Count number of declarations when printing them out
  - Count number of a specific functions declared with a given prefix (e.g. “gl” or “cuda”)
  - Count number of lines of each function
- Medium
  - Automatically add in comments for functions
  - Integrate a clang tool with cmake that runs everytime you build your project.
- Hard/Interesting?
  - Refactor code to your particular coding style (without using a tool like clang-tidy)

Contributing to Clang

Getting involved with clang

https://clang.llvm.org/get_involved.html

Getting Involved with the Clang Project

Once you have checked out and built clang and played around with it, you might be wondering what you can do to make it better and contribute to its development. Alternatively, maybe you just want to follow the development of the project to see it progress.

Resources

Useful resources

- The Clang AST - https://www.youtube.com/watch?v=VqCkCDFLSsc
- Clang static analysis https://www.youtube.com/watch?v=UcxF6CVueDM
- Plug Yourself In: Learn How to Write a Clang Compiler Plugin (Stephan Bergman) https://www.youtube.com/watch?v=pdxlmM477KY
- Clang-useful https://www.youtube.com/watch?v=E6i8jmiy8MY

Useful resources

- A way to avoid the JSON Compile_Commands database?
  - http://my-classes.com/2014/07/01/working-with-ast-matcher-without-clangtool-compilation-database/
  - https://clang.llvm.org/docs/JSONCompilationDatabase.html
- Where can I find tutorials for Clang
  - https://stackoverflow.com/questions/13923474/where-can-i-find-tutorials-for-clang
- Modern source-to-source transformations with Clang and LibTooling
  - https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling
  - github: https://github.com/eliben/llvm-clang-samples/blob/master/src_clang/matchers_rewriter.cpp
- Get list of methods
  - https://stackoverflow.com/questions/21878260/get-list-of-methods-in-class-using-clang
- Tutorial on detecting unused functions
  - https://xaizek.github.io/2014-08-28/detecting-unused-functions/
- Understanding the clang ast
  - https://jonasdevlieghere.com/understanding-the-clang-ast/
- https://clang.llvm.org/docs/ClangTools.html
-

Conclusion

- Clang is an exciting project with a lot of power
- There exist several ways to implement and build tools for your needs.

Thank You!

@MichaelShah | www.mshah.io

Feedback Form TBD
(Whether you watched this talk now or in the future!)

- Need to browse - Clang tutorial
  - https://github.com/loarabia/Clang-tutorial/wiki/TutorialOrig
  -

To read later

- Lessons from Building Static Analysis Tools at Google
  - https://cacm.acm.org/magazines/2018/4/226371-lessons-from-building-static-analysis-tools-at-google/fulltext

Compiling our Example

Add something about llvm-config here and how handy it is

http://llvm.org/docs/CommandGuide/llvm-config.html

https://jonasdevlieghere.com/understanding-the-clang-ast/#buildingtheexamples

When we finally generate an executable, we should copy it to the build/bin directory of our llvm.

This way our library can find all of the headers and libraries it needs. This is similar to clang and other tools, and it is a nice way to keep all of the tools in one place!

TBD -- discuss how would you distribute your tool commercially