Linaro connect Vancouver 2018

How to build a C++ processing tool using the Clang libraries

Peter Smith

Goals

- Find out what a non-expert can realistically build with the clang libraries.
- Find the problem space where building your own tool is worthwhile.
- Describe a process for building a refactoring tool and a plugin.
- Outline the basic boilerplate that a clang based tool needs to follow.

Motivation

- Make precise edits based on the compiler's AST representation of the source.
    - In particular C++ code is difficult to parse and not well covered by other tools.
- Introduce your own domain specific errors and warnings.
    - Complicated code-base with implicit rules that can be checked statically.
- Code generation based on C/C++ source
    - Serialization of data structures.
    - Foreign Function Interface (FFI).

What are the Clang Libraries?

- Clang the compiler is built from a composition of libraries.
- Libraries can be used independently to build C/C++ processing tools.
- Plugins can be built to run at compilation time.
- Compilation database is used to pass through command line options that might affect source code.

Clang APIs

- Clang provides a C and a C++ API
  - The C API libclang is stable but incomplete.
  - The C++ API libtooling is complete but not stable.
- External tools can choose which API to use.
- Plugins must use the C++ API.
- We will be using the C++ API
  - I want to use ASTMatchers, a domain specific language for querying the AST.
  - I want to be able to build a plugin.

Clang AST

- Made up of several class hierarchies with no common base class
  - `Type`, `Decl`, `DeclContext` and `Stmt`.
- Each node has dedicated traversal functions to access children.
- Clang makes heavy use of visitors to traverse the AST.
- ASTContext class holds additional information such as source locations.
- For a refactoring tool we can learn what we need to know as we go along
  - Make snippets of C code and see what the AST looks like.
  - Use the ASTMatchers domain specific language to find the AST nodes we want.
  - Search the source code and doxygen for the C++ code we need.

Getting Clang

- Multiple ways to do this
  - In tree or out of tree builds?
  - Monorepo or individual repos?
  - Git or svn?
- The example here uses an in tree build with the git monorepo.
- In tree means using Clang's build system.

Step by step guide

- Follow the instructions in the LLVM GettingStarted guide to clone the monorepo.
- Make a new directory in the clang-tools-extra directory for the tool.
- Modify the CMakeLists.txt to include the directory
  - add_subdirectory(my-tool-dir-name)
- Change to the directory and Add a CMakeLists.txt file.
- The example below is for a standalone tool. We'll cover the plugin later.

```
set(LLVM_LINK_COMPONENTS support)
add_clang_executable(clang-yvr18 ClangYVR18.cpp)
target_link_libraries(clang-yvr18 PRIVATE clangAST
                      clangASTMatchers clangBasic
                      clangFrontend clangRewrite clangTooling)
```

Building the tool

- Clang uses cmake, some extra options are needed
  - We need to turn on clang-tools-extra builds.
  - A static debug build is very large and will likely kill your machine with parallel links.
  - `ninja all` or `ninja name-of-your-tool` will build it.
  - Examples aren't strictly necessary but it is where the existing plugins live.

```
cmake -GNinja \
      -DCMAKE_EXPORT_COMPILE_COMMANDS=1\
      -DLLVM_ENABLE_ASSERTIONS=On\
      -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra"\
      -DCMAKE_BUILD_TYPE=Debug\
      -DBUILD_SHARED_LIBS=ON\
      -DLLVM_PARALLEL_LINK_JOBS=2\
      -DLLVM_BUILD_EXAMPLES=ON\
      -DLLVM_INCLUDE_EXAMPLES=ON\
      path/to/llvm-project/llvm
```

Process for writing a tool

1. Have a clear idea of what you want to do.
2. Write many minimal source examples to use as test cases.
3. Use the clang `-Xclang -ast-dump` command line option on your examples.
4. Work out the Clang AST Nodes that you need to match against.
5. Use the clang-query tool on your examples to develop an ASTMatcher.
6. Import the ASTMatcher into the C++ code for your plugin or tool.

Our use case

- Some functions take bool or integer parameters
    - `void function(int value, bool ignore_parameter);`
- Passing literal values can be unclear at the call site
    - `function(0, true);`
- A comment with the parameter value can be useful
    - `function( /* value */ 0, /* ignore_parameter */ true);`
- Can we write a tool to insert these comments for us?
    - Find all callsites using integer or bool literals as arguments.
    - Look up the parameters of the callee and insert the names as comments.

Test Cases

- Recommend a single test per source file.
- Writing test cases first lets us use the ast-dump feature to work out what AST nodes we need to search for.
- We can use clang-query to build our ASTMatcher.
- In general you won’t need a compilation database for the test cases
    - Add `--` to the end of the command line and clang tools to inform tools that there is no database.
    - `clang-query path/to/test.cpp --`

C Direct Function Call

```
void f(bool p1);

void test_f() {
  f(true);
}
```

```
|-FunctionDecl 0x1145210 <slide1.cpp:1:1, col:15> col:6 used f 'void (bool)'
| `-ParmVarDecl 0x1145148 <col:8, col:13> col:13 p1 'bool'
`-FunctionDecl 0x1145340 <line:3:1, line:5:1> line:3:6 test_f 'void ()'
  `-CompoundStmt 0x11454e0 <col:15, line:5:1>
    `-CallExpr 0x11454b0 <line:4:3, col:9> 'void'
      |-ImplicitCastExpr 0x1145498 <col:3> 'void (*)(bool)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x1145448 <col:3> 'void (bool)' lvalue Function 0x1145210 'f' 'void
(bool)'
      `-CXXBoolLiteralExpr 0x1145430 <col:5> 'bool' true
```

C Indirect function call

```
void test_fptr(void (f)(bool)) {
  f(true);
}
```

```
`-FunctionDecl 0x15d33e0 <slide3.cpp:2:1, line:4:1> line:2:6 test_fptr 'void (void (*)(bool))'
  |-ParmVarDecl 0x15d32e0 <col:16, col:29> col:22 used f 'void (*)(bool)':'void (*)(bool)'
  `-CompoundStmt 0x15d3548 <col:32, line:4:1>
    `-CallExpr 0x15d3518 <line:3:3, col:9> 'void'
      |-ImplicitCastExpr 0x15d3500 <col:3> 'void (*)(bool)':'void (*)(bool)' <LValueToRValue>
      | `-DeclRefExpr 0x15d34c0 <col:3> 'void (*)(bool)':'void (*)(bool)' lvalue ParmVar
0x15d32e0 'f' 'void (*)(bool)':'void (*)(bool)'
      `-CXXBoolLiteralExpr 0x15d34e8 <col:5> 'bool' true
```

Casts and anonymous parameters

```
void f(bool);

void test_f() {
  f(3);
}
```

```
|-FunctionDecl 0x12f7210 <slide5.cpp:1:1, col:12> col:6 used f 'void (bool)'
| `-ParmVarDecl 0x12f7148 <col:8> col:12 'bool'
`-FunctionDecl 0x12f7340 <line:3:1, line:5:1> line:3:6 test_f 'void ()'
  `-CompoundStmt 0x12f7508 <col:15, line:5:1>
    `-CallExpr 0x12f74c0 <line:4:3, col:6> 'void'
      |-ImplicitCastExpr 0x12f74a8 <col:3> 'void (*)(bool)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x12f7450 <col:3> 'void (bool)' lvalue Function 0x12f7210 'f' 'void
(bool)'
      `-ImplicitCastExpr 0x12f74f0 <col:5> 'bool' <IntegralToBoolean>
        `-IntegerLiteral 0x12f7430 <col:5> 'int' 3
```

C++ member function

```
struct S {
  bool f(bool p1);
};

void test_f() {
  S s;
  s.f(true);
}
```

```
| |-CXXRecordDecl 0x11f5248 <col:1, col:8> col:8 implicit struct S
| |-CXXMethodDecl 0x11f53d0 <line:2:3, col:17> col:8 used f 'bool (bool)'
| | `-ParmVarDecl 0x11f52e0 <col:10, col:15> col:15 p1 'bool'
...
   `-CXXMemberCallExpr 0x11f5bb0 <line:7:3, col:11> 'bool'
     |-MemberExpr 0x11f5b60 <col:3, col:5> '<bound member function type>' .f 0x11f53d0
     | `-DeclRefExpr 0x11f5b38 <col:3> 'S' lvalue Var 0x11f55c8 's' 'S'
     `-CXXBoolLiteralExpr 0x11f5b98 <col:7> 'bool' true
```

C++ Operator()

```
struct S {
  void operator()(bool p1) {}
};

void test_callable() {
  S s;
  s(true);
}
```

```
 | |-CXXRecordDecl 0x1637248 <col:1, col:8> col:8 implicit struct S
 | |-CXXMethodDecl 0x16373d0 <line:2:3, col:29> col:8 used operator() 'void (bool)'
 | | |-ParmVarDecl 0x16372e0 <col:19, col:24> col:24 p1 'bool'
 ...
    `-CXXOperatorCallExpr 0x1637bf0 <line:7:3, col:9> 'void'
       |-ImplicitCastExpr 0x1637bd8 <col:4, col:9> 'void (*)(bool)' <FunctionToPointerDecay>
       | `-DeclRefExpr 0x1637b88 <col:4, col:9> 'void (bool)' lvalue CXXMethod 0x16373d0
 'operator()' 'void (bool)'
       |-DeclRefExpr 0x1637b48 <col:3> 'S' lvalue Var 0x16375d8 's' 'S'
       `-CXXBoolLiteralExpr 0x1637b70 <col:5> 'bool' true
```

Observations on the AST so far

- Calls represented by **CallExpr**, **CXXMemberCallExpr**, **CXXOperatorCallExpr**
  - **CXXOperatorCallExpr** has the **DeclRefExpr** of the object as a child but **CXXMemberCallexpr** does not.
- Bool literals represented by **CXXBoolLiteralExpr**.
- Integer literals represented by **IntegerLiteral**.
- Function parameters identified by **ParmVarDecl**
  - May have an empty name.
- Indirect call distinguished by lvalue not being a **Function**.
- We will have to handle implicit casts.

Ways to call a function in C++

```
void test_lambda() {
  auto lambda = [](bool p1, bool p2){
  };
  lambda(true, false);
}
```

```
void variadic(int p1, ...);
void test_variadic() {
  variadic(2, true, false);
}
```

```
template <typename T> class TemplateClass
{
public:
  void test_fn(T p1) {}
};

void test_template_dependent() {
  TemplateClass<bool> tc;
  tc.test_fn(true);
}
```

```
#include <functional>
void test_std_function(std::function<void(bool)> f) {
  f(true);
}
```

And more

```
struct S {
  void boolfn(bool p1, bool p2);
};
struct S2 {
  void boolfn(bool a1, bool a2);
};
template <typename T> void callT(T t) {
  t.boolfn(true, false);
}
void test_template_dependent_call() {
  S s; S2 s2;
  callT(s); callT(s2);
}
```

```
struct S {
  static void foo(bool p1, bool p2) {}
};
struct S2 {
  static void foo(bool a1, bool a2) {}
};
template <typename T> void f2() {
  T::foo(true, false);
}
void test_static_from_template() {
  f2<S>();
  f2<S2>();
}
```

```
template <typename... Ts> void variadic_template_fn(Ts ... args);
void test_variadic_template() {
  variadic_template_fn(true, false);
}
```

Observations on AST with templates

- Option `-Xclang -ast-print` will show the C++ code instantiated
- Will see the instantiation and the original template in the AST
- Leads to some complications in finding parameter names
  - Couldn’t work out how to distinguish cases with ASTMatcher.
  - Decided that since cases would be rare, it wasn’t worth the effort to build custom solution.

```
template <typename... Ts> void variadic_template_fn(Ts ... args);

void test_variadic_template() {
  variadic_template_fn(true, false);
}
```

```
template <typename... Ts> void variadic_template_fn(Ts ... args);
template<> void variadic_template_fn<<bool, bool>>(bool args, bool args);
void test_variadic_template() {
  variadic_template_fn(true, false);
}
```

Clang AST Matchers

- A domain specific language to express patterns in the AST
  - Example: `recordDecl(hasName("MyClass")).bind("id")`
- Made up of many smaller matchers chained together in functional style.
  - **Node Matchers** such as recordDecl() match types of ASTNode and can be bound.
  - **Narrowing Matchers** such as hasName("name") match attributes of of ASTNodes.
  - **Traversal Matchers** match nodes reachable from NodeMatcher.
- Develop matcher in C++ or use an interactive tool `clang-query`.
- Code can be called back when the matcher hits the pattern
  - Example: `struct MyClass {};`
- AST Nodes can be bound and can be retrieved in the callback.
  - Example: We can refer to the `RecordDecl` AST Node via the string "id".

clang-query

- Interactive tool for development of ASTMatchers
  - `clang-query source.cpp --`
  - Useful to set output to AST dump fragment with `set output dump`
- From the clang tutorial page, the process to develop the matcher is:
  - Find the outermost class in Clang's AST you want to match.
  - Look at the **AST Matcher Reference** for matchers that either match the node you're interested in or narrow down attributes on the node.
  - Create your outer match expression. Verify that it works as expected.
  - Examine the matchers for what the next inner node you want to match is.
  - Repeat until the matcher is finished.

Developing our matcher

- Our goal is to find call sites with **bool** or **integer** literals as parameters so that we can insert a comment with the parameter name
    - Need to find all function calls with bool or integer literals.
    - Need to find the callee so that we can find the parameters.
- Do not want to match ambiguous cases
    - Callee is dependent on a template parameter.
    - Function pointers.
- Following the process `callExpr()` will be the outermost class
    - In clang-query we can use `match callExpr()` to confirm it matches all the calls in our tests.
- We now need to narrow the matcher to only match calls we are interested in
    - `hasAnyArgument()` can be used to find literal parameters.
    - `allOf() anyOf()` and `unless()` can be used to combine matchers.
    - `ignoreParenCasts()` can help us with implicit casts.

First attempt

```
clang-query> set output dump
clang-query> match callExpr(anyOf(hasAnyArgument(ignoringParenCasts(cxxBoolLiteral()),
                                  hasAnyArgument(ignoringParenCasts(integerLiteral()))))
Binding for "root":
CXXMemberCallExpr 0x13eccd0 <p10.cpp:12:3, col:21> 'void'
|-MemberExpr 0x13ecc80 <col:3, col:5> '<bound member function type>' .inline_func 0x13bef70
| `-DeclRefExpr 0x13ecc58 <col:3> 'class C' lvalue Var 0x13bf468 'c' 'class C'
`-CXXBoolLiteralExpr 0x13eccb8 <col:17> '_Bool' true
```

- Close but not good enough
  - Matches all calls with a bool or integer literal passed as a parameter.
  - Also matches calls to function pointers, variadic functions and all template function calls.
  - No reference to the callee that we can bind.

Refined ASTMatcher

```
clang-query> set output dump
clang-query> match
    callExpr(
        allOf(callee(functionDecl(unless(isVariadic()))),
              unless(cxxMemberCallExpr(
                  on(hasType(substTemplateTypeParmType())))),
              anyOf(hasAnyArgument(ignoringParenCasts(cxxBoolLiteral())),
                    hasAnyArgument(ignoringParenCasts(integerLiteral())))))
```

- Close enough to switch to C++
  - Excludes function pointers and variadic functions.
  - Excludes C++ member calls where the member is dependent on a template parameter.
  - Has an inner matcher against the callee that we can bind.

Building the C++ tool

- Use Clang's common command line options parsing library.
- Implement interface code to set up the ASTMatcher and callback
  - `ASTFrontEndAction`, `ASTConsumer`, `MatchFinder::MatchCallback`
- The `ASTMatcher` we developed in clang-query can be reused.
- Add bind calls for the caller and callee.
- For each caller find the parameters associated with the literal arguments and insert comment.
- Use Clang's `Rewriter` to make our modifications to the source file.
- Print the modified source file after translation unit finished processing.

Main function

```
// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...\n");

int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  // YVR18FrontEndAction is our implementation of ASTFrontEndAction
  return Tool.run(newFrontendActionFactory<YVR18FrontendAction>().get());
}
```

ASTFrontendAction

```
// We implement the ASTFrontEndAction interface to run our matcher on a
// source file.
class YVR18FrontendAction : public ASTFrontendAction {
public:
  // Output the edit buffer for this translation unit
  void EndSourceFileAction() override {
    YVR18Rewriter.getEditBuffer(YVR18Rewriter.getSourceMgr().getMainFileID())
        .write(llvm::outs());
  }
  // Returns our ASTConsumer implementation per translation unit.
  std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
                                                 StringRef file) override {
    YVR18Rewriter.setSourceMgr(CI.getSourceManager(), CI.getLangOpts());
    return llvm::make_unique<YVR18ASTConsumer>(YVR18Rewriter);
  }
private:
  Rewriter YVR18Rewriter;
};
```

ASTConsumer

```
class YVR18ASTConsumer : public ASTConsumer {
public:
  YVR18ASTConsumer(Rewriter &R) : LAC(R) {
    // We use almost the same syntax as the ASTMatcher prototyped in
    // clang-query. The changes are the .bind(string) additions so that we
    // can access these once the match has occurred.
    StatementMatcher CallSiteMatcher =
        callExpr(
            allOf(callee(functionDecl(unless(isVariadic())).bind("callee")),
                  unless(cxxMemberCallExpr(
                      on(hasType(substTemplateTypeParmType())))),
                  anyOf(hasAnyArgument(ignoringParenCasts(cxxBoolLiteral())),
                        hasAnyArgument(ignoringParenCasts(integerLiteral())))))
            .bind("functions");
    // LAC is our callback that will run when the ASTMatcher finds the pattern above.
    Matcher.addMatcher(CallSiteMatcher, &LAC);
  }
  // Implement the call back so that we can run our Matcher on the source file.
  void HandleTranslationUnit(ASTContext &Context) override {
    Matcher.matchAST(Context);
  }
private:
  MatchFinder Matcher;
  LiteralArgCommenter LAC;
};
```

Extracting the callee parameters

```
class LiteralArgCommenter : public MatchFinder::MatchCallback {
public:
  LiteralArgCommenter(Rewriter &YVR18Rewriter) : YVR18Rewriter(YVR18Rewriter) {}

  // This callback will be executed whenever the Matcher in YVR18ASTConsumer
  // matches.
  virtual void run(const MatchFinder::MatchResult &Result) {
    // ASTContext allows us to find the source location.
    ASTContext *Context = Result.Context;
    // Record the callees parameters. We can access the callee via the
    // .bind("callee") from the ASTMatcher. We will match these with
    // the callers arguments later.
    std::vector<ParmVarDecl *> Params;
    const FunctionDecl *CD =
        Result.Nodes.getNodeAs<clang::FunctionDecl>("callee");
    for (FunctionDecl::param_const_iterator PI = CD->param_begin(),
                                            PE = CD->param_end();

         PI != PE; ++PI) {
      Params.push_back(*PI);
    }
    // Continues on next slide!
```

Doing the rewriting

```
const CallExpr *E = Result.Nodes.getNodeAs<clang::CallExpr>("functions");
    size_t Count = 0;

    if (E && CD && !Params.empty()) {
      auto I = E->arg_begin();
      if (isa<CXXOperatorCallExpr>(E))
        // The first parameter is the object itself, skip over it.
        ++I;
      // For each argument match it with the callee parameter
      // If it is an integer or boolean literal then insert a comment
      // into the edit buffer.
      for (auto End = E->arg_end(); I != End; ++I, ++Count) {
        ParmVarDecl *PD = Params[Count];
        FullSourceLoc ParmLocation = Context->getFullLoc(PD->getBeginLoc());
        const Expr *AE = (*I)->IgnoreParenCasts();
        if (auto *IntArg = dyn_cast<IntegerLiteral>(AE)) {
          FullSourceLoc ArgLoc = Context->getFullLoc(IntArg->getBeginLoc());
          if (ParmLocation.isValid() && !PD->getDeclName().isEmpty() &&
              EditedLocations.insert(ArgLoc).second)
            // Will insert our text immediately before the Argument
            YVR18Rewriter.InsertText(
                ArgLoc,
                (Twine(" /* ") + PD->getDeclName().getAsString() + " */ ")
                    .str());
        }
        // Boolean case is almost identical, use CXXBoolLiteralExpr
```

Tool in action

```
clang-yvr18 test1.cpp --
void f(bool p1);
void test_f() {
  f( /* p1 */ true);
}
clang-yvr18 test2.cpp --
struct S {
  void operator()(bool p1) {}
};
void test_callable() {
  S s;
  s( /* p1 */ true);
}
```

```
clang-yvr18 test3.cpp --
template <typename T>
void template_fn(T p1) {}

void test_template_fn() {
  template_fn( /* p1 */ true);
}

clang-yvr18 test4.cpp --
template <typename... Ts>
void variadic_template_fn(Ts ... args);
void test_variadic_template() {
  variadic_template_fn( /* args */ true,
                        /* args */ false);
}
```

Clang plugin

- Instead of rewriting, give an error message if there is bool or int literal used without the callee having a parameter name.
- Most of the code in the executable tool can be reused
  - `CMakeLists.txt` changes to build a loadable module.
  - Remove main function and command line options.
  - Remove the Rewriter.
  - Replace the `ASTFrontEndAction` with a `PluginASTAction`.
  - Add a function to register our plugin.
  - Alter the callback to issue a diagnostic.

Build system changes

- Make a new directory for the source file. The existing plugins live in the clang/examples directory.
- Add add_subdirectory(<new directory name>) to the CMakeLists.txt file.
- In the new directory add a CMakeLists.txt file with something like add_llvm_loadable_module(ClangYVR18Plugin ClangYVR18Plugin.cpp PLUGIN_TOOL clang)
- To build use ninja ClangYVR18Plugin.so

Source file changes

```
// Implement PluginASTAction rather than ASTFrontendAction
class YVR18PluginAction : public PluginASTAction {
public:
  // Our plugin can alter behavior based on the command line options
  bool ParseArgs(const CompilerInstance &CI,
                 const std::vector<std::string> &args) override {
    return true;
  }
  // Returns our ASTConsumer per translation unit.
  std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
                                                  StringRef file) override {
    return llvm::make_unique<YVR18ASTConsumer>();
  }
};
static FrontendPluginRegistry::Add<YVR18PluginAction> X("YVR18Plugin",
                                                         "YVR18 Plugin example");
```

Issuing a diagnostic

```
DiagnosticsEngine &Diagnostics = Context->getDiagnostics();
unsigned DiagID =
    Diagnostics.getCustomDiagID(
              DiagnosticsEngine::Error,
              "Bool Literal passed as argument to parameter without name");
Diagnostics.Report(ArgLoc, DiagID);
```

Using the plugin

- Need command line options to load and run the plugin
- `-Xclang -load -Xclang <plugin name>.so -Xclang -plugin -Xclang <plugin name>`

```
clang ../yvr18/p11.cpp -Xclang -load -Xclang lib/ClangYVR18Plugin.so -Xclang -plugin -Xclang
YVR18Plugin -c
../yvr18/p11.cpp:7:26: error: Bool Literal passed as argument to parameter
      without name
  prototype_without_name(false);
                         ^
1 error generated.
```

Conclusions

- Writing your own clang tool is feasible with about a week of effort.
- There is a common structure to most tools, look at examples and existing tutorials.
- Expect to have to fix minor problems due to API and build system changes.
- Can save a lot of effort by deciding not to handle difficult corner cases.
- Think about whether you can achieve your refactoring using simpler tools
    - spatch for C code?
- Think hard about maintenance and deployment of a plugin
    - Interface is not stable so a plugin for Clang 6.0 may not work on Clang 7.0
    - Do you control the compilers your contributors use?
    - Do you have a CI build that you could enable the plugin on?
- Most publicly visible deployments have been in large complex projects
    - Chromium, Mozilla, Libre Office

References

- I am eternally grateful for the clang tutorials available online and in the clang documentation
    - Understanding the Clang AST by Jonas Devlieghere.
    - AST Matchers and Clang refactoring tools by Eli Bendersky.
    - Compilation Databases for Clang based tools by Eli Bendersky.
    - Don't write a Clang Plugin Chromium website.
    - C++ Static Analysis using Clang by Ehsan Akhgari.
    - LibASTMatchers Tutorial Clang documentation.
    - Clang Plugins Clang documentation.
    - LibASTMatchers reference Clang documentation.