Create your own refactoring tool with clang

Richard Thomson
Senior Software Engineer
Fusion-io
@LegalizeAdulthd
http://LegalizeAdulthood.wordpress.com
legalize@xmission.com

Outline

- Statement of the problem
- Getting started with clang
- Examining `remove-cstr-calls`
- Bootstrapping remove-void-args
- Understanding clang's AST
- The compilation database
- Exploring matched function definitions
- Exploring a realistic source file
- Exploring matched function declarations
- Replacing matched function declarations and definitions
- Handling typedef statements
- Member functions
- Fields
- Uninitialized variable
- Initialized variable
- Constructors
- Cast operator expression

The Problem

- Dearth of refactoring tools for C/C++
- Existing tools tightly coupled to IDEs
- C/C++ code bases are often old
- Old code bases need refactoring the most!
- Tool adoption requires:
  - Easily invoked from workflow
  - Accurate
  - Never produce incorrect code

Some Existing Refactoring Tools

- Visual Studio add-ons
  - Visual Assist/X by Whole Tomato
  - CodeRush by DevExpress
- CLI
  - clang-modernize
  - remove-cstr-calls
  - clang-tidy
- Eclipse
  - CDT
- Email me about others! legalize@xmission.com

```
// dusty_deck.cpp
int foo(void) { return 0; }
void bar(void) { }
struct gronk {
  gronk(void) { }
  ~gronk(void) { }
  void foo(void) { }
  void (*f)(void);
  void (gronk::*p)(void);
};
```

The Problem

```
// dusty_deck.cpp
int foo(void) {
void bar(void)
struct gronk {
  gronk(void) {
  ~gronk(void) { }
  void foo(void) { }
};
```

Ugh. Someone's been bringing their dusty old C style habits into our C++ code.

Let's get rid of these!

They serve no purpose in C++.

The Problem

```
void (*f)(void);
void (gronk::*p)(void);
typedef void (fn)(void);
typedef void (gronk::*pm)(void);
void ff(void (*f)(void)) { }
f = (void (*)(void)) 0;
f = static_cast<void (*)(void)>(0);
f = reinterpret_cast<void (*)(void)>(0);
extern void (*)(void) get_fn_ptr();
// you can think of more...
```

Expressions Also Participate

Getting Started With Clang

- Download LLVM 3.4
- Download Clang 3.4
- Download Clang "Extra Tools" 3.4
- Unpack source into correct location
- Configure build with Cmake
- Build:
  - 293 projects in VS
  - 90+ minutes of 8 cores @ 100% CPU later...
  - We get a 9 GB build tree
  - Srsly?

- Yes, for now.
- Windows package doesn't include the necessary libraries or headers.
- All packages are missing some useful tools.
- In Clang 3.4, some useful tools require libedit as a prerequisite.

Srsly?

- libedit requirement eliminated.
- Windows package build can be enhanced.
- Additional tools added to package (patch forthcoming)
- Goal is to have prebuilt binary packages for most environments to lower the bar for refactoring tool development.

Will Be Better in Clang 3.5

Modify an Existing Tool to Learn

- We need to remove some `(void)` stuff
- `remove-cstr-calls` removes redundant calls to `c_str()`
- We can use this tool as a starting point

```
// remove-cstr-calls <cmake-output-dir> <file1> <file2> ...
//
// Where <cmake-output-dir> is a CMake build directory in
// which a file named compile_commands.json exists.
//
// <file1> ... specify the paths of files in the Cmake
// source tree. This path is looked up in the compile
// command database.

// this:
void f1(const std::string &s) {
  f1(s.c_str());
}

// becomes this:
void f1(const std::string &s) {
  f1(s);
}
```

remove-cstr-calls

```
// remove-cstr-calls <cmake-output-dir> <file1> <file2> ...
//
// Where <cmake-output-dir> is a CMake build directory in
// which a file named compile_commands.json exists.
//
// <file1> ... specify the paths of files in the Cmake
// source tree. This path is looked up in the compile
// command database.

// this:
void f1(const std::string &s) {
  f1(s.c_str());
}

// becomes this:
void f1(const std::string &s) {
  f1(s);
}
```

remove-cstr-calls

```
// remove-cstr-calls <cmake-output-dir> <file1> <file2> ...
//
// Where <cmake-output-dir> is a CMake build directory in
// which a file named compile_commands.json exists.
//
// <file1> ... specify the paths of files in the Cmake
// source tree. This path is looked up in the compile
// command database.

// this:
void f1(const std::string &s) {
  f1(s.c_str());
}

// becomes this:
void f1(const std::string &s) {
  f1(s);
}
```

remove-cstr-calls

```
cl::opt<std::string> BuildPath(
  cl::Positional,
  cl::desc("<build-path>"));

cl::list<std::string> SourcePaths(
  cl::Positional,
  cl::desc("<source0> [... <sourceN>]"),
  cl::OneOrMore);
```

Setting Up Command-Line Args

```
cl::opt<std::string> BuildPath(
    cl::Positional,
    cl::desc("<build-path>"));

cl::list<std::string>
    cl::Positional,
    cl::desc("<source0> [... <sourceN>]"),
    cl::OneOrMore);
```

LLVM Support library provides command-line argument classes in cl namespace

Setting Up Command-Line Args

```
cl::opt<std::string> BuildPath(
  cl::Positional,
  cl::desc("<build-path>"));
```

```
cl::list<std::string> SourcePaths(
  cl::Positional,
  cl::desc("<source0>
  cl::OneOrMore);
```

The first positional argument stored in a string gives us the build path.

Setting Up Command-Line Args

```
cl::opt<std::
  cl::Positi
  cl::desc("<
```

The second positional argument stored in a list of string gives us one or more source files to refactor

```
cl::list<std::string> SourcePaths(
  cl::Positional,
  cl::desc("<source0> [... <sourceN>]"),
  cl::OneOrMore);
```

Setting Up Command-Line Args

```
int main(int argc, const char **argv) {
  llvm::sys::PrintStackTraceOnErrorSignal();
  llvm::OwningPtr<CompilationDatabase> Compilations(
    tooling::FixedCompilationDatabase::loadFromCommandLine(
      argc, argv));
  cl::ParseCommandLineOptions(argc, argv);
  if (!Compilations) {
    std::string ErrorMessage;
    Compilations.reset(
      CompilationDatabase::loadFromDirectory(
        BuildPath, ErrorMessage));
    if (!Compilations)
      llvm::report_fatal_error(ErrorMessage);
    }
  tooling::RefactoringTool Tool(*Compilations, SourcePaths);
```

main(): Startup

```
int main(int argc, const char **argv) {
  llvm::sys::PrintStackTraceOnErrorSignal();
  llvm::OwningPtr<CompilationDatabase> Compilations(
    tooling::FixedCompilationDatabase::loadFromCommandLine(
      argc, argv));
  cl::ParseCommandLineOptio
  if (!Compilations) {
    std::string ErrorMessag
    Compilations.reset(
      CompilationDatabase::loadFromDirectory(
        BuildPath, ErrorMessage));
    if (!Compilations)
      llvm::report_fatal_error(ErrorMessage);
    }
  tooling::RefactoringTool Tool(*Compilations, SourcePaths);
```

LLVM Support library utility for printing out a stack trace diagnostic when an unhandled signal(2) occurs.

main(): Startup

```
int main(int argc, const char **argv) {
  llvm::sys::PrintStackTraceOnErrorSignal();
  llvm::OwningPtr<CompilationDatabase> Compilations(
    tooling::FixedCompilationDatabase::loadFromCommandLine(
      argc, argv));
  cl::ParseCommandLineOptions(argc, argv);
  if (!Compilations) {
    std::string ErrorMessage;
    Compilations.reset(
      CompilationDatabase::loac
        BuildPath, ErrorMessag
    if (!Compilations)
      llvm::report_fatal_error
    }
  tooling::RefactoringTool Tool
```

Let's us build a compilation database directly from the command-line.

More on the compilation database later!

main(): Startup

```
int main(int argc, const char **argv) {
  llvm::sys::PrintStackTraceOnErrorSignal();
  llvm::OwningPtr<CompilationDatabase> Compilations(
    tooling::FixedCompilationDatabase::loadFromCommandLine(
      argc, argv));
  cl::ParseCommandLineOptions(argc, argv);
  if (!Compilations) {
    std::string ErrorMessage;
    Compilations.reset(
      CompilationDatabase::loa
        BuildPath, ErrorMessage
    if (!Compilations)
      llvm::report_fatal_error(ErrorMessage);
    }
  tooling::RefactoringTool Tool(*Compilations, SourcePaths);
```

Get the command-line options parsed.

main(): Startup

Locate the compilation database using the given directory.

```
int main(int argc, const char **argv) {
  llvm::sys::PrintStackTraceOnErrorSignal();
  OwningPtr<CompilationDatabase> Compilations(
    tooling::FixedCompilationDatabase::loadFromCommandLine(
      argc, argv));
  cl::ParseCommandLineOptions(argc, argv);
  if (!Compilations) {
    std::string ErrorMessage;
    Compilations.reset(
      CompilationDatabase::loadFromDirectory(
        BuildPath, ErrorMessage));
    if (!Compilations)
      llvm::report_fatal_error(ErrorMessage);
    }
  tooling::RefactoringTool Tool(*Compilations, SourcePaths);
```

main(): Startup

No, really, we need this thing to continue!

```
int main(int argc, const char **argv) {
  llvm::sys::PrintStackTraceOnErrorSignal();
  OwningPtr<CompilationDatabase> Compilations(
    tooling::FixedCompilationDatabase::loadFromCommandLine(
      argc, argv));
  cl::ParseCommandLineOptions(argc, argv);
  if (!Compilations) {
    std::string ErrorMessage;
    Compilations.reset(
      CompilationDatabase::loadFromDirectory(
        BuildPath, ErrorMessage));
    if (!Compilations)
      llvm::report_fatal_error(ErrorMessage);
    }
  tooling::RefactoringTool Tool(*Compilations, SourcePaths);
```

main(): Startup

Get our refactoring tool instance created.

RefactoringTool is a ClangTool that knows how to parse source files into an AST, match nodes in the AST and create a list of source file text replacements.

We build it from the compilation database and the source files to refactor.

```
    CompilationDatabase::loadFromDirectory(
      BuildPath, ErrorMessage));
  if (!Compilations)
    llvm::report_fatal_error(ErrorMessage);
  }
tooling::RefactoringTool Tool(*Compilations, SourcePaths);
```

main(): Startup

```
ast_matchers::MatchFinder Finder;
FixCStrCall Callback(&Tool.getReplacements());
Finder.addMatcher(/* ... */);
Finder.addMatcher(/* ... */);
return Tool.runAndSave(
  newFrontendActionFactory(&Finder));
```

main(): AST Matching

```
ast_matchers::MatchFinder Finder;
FixCStrCall Callback(&Tool.getReplacements());
Finder.addMatcher(/* ... */);
Finder.addMatcher(/* ... */);
return Tool.runAndSave(
  ne
```

Create an instance of MatchFinder. MatchFinder provides an implementation of ASTConsumer to consume the AST created by the compiler.

The AST is matched in pre-order traversal, applying matchers in the order in which they are added to the finder.

main(): AST Matching

```
ast_matchers::MatchFinder Finder;
FixCStrCall Callback(&Tool.getReplacements());
Finder.addMatcher(/* ... */);
Finder.addMatcher(/* ... */);
return Tool.runAndSave(
  ne
```

Create an instance of our refactoring code. We pass it the address of the tool's source file replacements list so it can add replacements as it processes matches.

main(): AST Matching

```
ast_matchers::MatchFinder Finder;
FixCStrCall Callback(&Tool.getReplacements());
Finder.addMatcher(/* ... */);
Finder.addMatcher(/* ... */);
return Tool.runAndSave(
  newFrontendActionFactory(&Finder));
```

Add AST matchers to the MatchFinder.

Matchers are built up using a "builder" style interface.
This lets us express matchers using a fluent API.

main(): AST Matching

```
ast_matchers::MatchFinder Finder;
FixCStrCall Callback(&Tool.getReplacements());
Finder.addMatcher(/* ... */);
Finder.addMatcher(/* ... */);
return Tool.runAndSave(
  newFrontendActionFactory(&Finder));
```

Connect the MatchFinder to the front end of the compiler and pass this front end to the RefactoringTool to process source files, match AST nodes, build replacement lists and then modify the source files from the replacement lists.

main(): AST Matching

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

1st Matcher std::string(s.c_str())

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
```

Matches constructor call expressions,
including implicit constructor expressions.

```
      1,
      defaultArgExpr()))),
  &Callback);
```

1st Matcher std::string(s.c_str())

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
      1,
      defaultArgExpr()))),
  &Callback);
```

Matches a method declaration whose name is the name of the c'tor for std::string.

1st Matcher std::string(s.c_str())

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
    hasArgument(
      1,
      defaultArgExpr()))),
  &Callback);
```

The c'tor call takes two arguments.

1st Matcher std::string(s.c_str())

```
Finder.addMa
  constructF
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

The first argument is a call to the std::string::c_str() method.

1st Matcher std::string(s.c_str())

```
Finder.ad
  construc
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

Bind to a member function call expression matching the list of passed matchers.

1st Matcher std::string(s.c_str())

```
Finde
  cor
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

Bind the member function call expression to "call", so we can use this as the text to be replaced.

1st Matcher std::string(s.c_str())

Bind the member expression to "member", so we can determine if the member is invoked by value or by pointer.

```
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

1st Matcher std::string(s.c_str())

The method being called is a declaration matching the name for std::string::c_str()

```
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

1st Matcher std::string(s.c_str())

c_str() is invoked on some expression, which we bind to "arg".

```
Fi
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

1st Matcher std::string(s.c_str())

Fi

Matches if the callee matches the inner matcher.

```
  hasDeclaration(
    methodDecl(hasName(StringConstructor))),
  argumentCountIs(2),
  hasArgument(
    0,
    id("call", memberCallExpr(
        callee(id("member", memberExpr())),
        callee(methodDecl(hasName(StringCStrMethod))),
        on(id("arg", expr()))))),
  hasArgument(
    1,
    defaultArgExpr())),
&Callback);
```

1st Matcher std::string(s.c_str())

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(
      m
    argumentCountIs(2),
    hasArgument(
      0,
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr()))))),
    hasArgument(
      1,
      defaultArgExpr())),
  &Callback);
```

The second argument is a default argument.

1st Matcher std::string(s.c_str())

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(
      methodDecl(hasName(StringConstructor))),
    argumentCountIs(2),
    hasArgument(
      id("call", memberCallExpr(
          callee(id("member", memberExpr())),
          callee(methodDecl(hasName(StringCStrMethod))),
          on(id("arg", expr())))))),
    hasArgument(
      1,
      defaultArgExpr()))),
  &Callback);
```

Connect the matcher to our refactoring callback.

1st Matcher std::string(s.c_str())

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(methodDecl(anyOf(
      hasName("::llvm::StringRef::StringRef"),
      hasName("::llvm::Twine::Twine")))),
    argumentCountIs(1),
    hasArgument(
      0,
      id("call", memberCallExpr(
        callee(id("member", memberExpr())),
        callee(methodDecl(hasName(StringCStrMethod))),
        on(id("arg", expr())))))),
  &Callback);
```

2nd Matcher LLVM String Classes

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(methodDecl(anyOf(
      hasName("::llvm::StringRef::StringRef"),
      hasName("::llvm::Twine::Twine")))),
    argumentCountIs(1),
    has
      0,
      id("call", memberCallExpr(
        callee(id("member", memberExpr())),
        callee(methodDecl(hasName(StringCStrMethod))),
        on(id("arg", expr()))))))),
  &Callback);
```

Matches if any of the child matchers match.

2nd Matcher LLVM String Classes

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(methodDecl(anyOf(
      hasName("::llvm::StringRef::StringRef"),
      hasName("::llvm::Twine::Twine")))),
    argumentCountIs(1),
    hasArgument(
      0,
      id("call", memberCallExpr(
                                                    cMethod))),
  &Callback);
```

LLVM StringRef and Twine classes should be constructed from std::string directly instead of from std:string::c_str().

2nd Matcher LLVM String Classes

```
Finder.addMatcher(
  constructExpr(
    hasDeclaration(methodDecl(anyOf(
      hasName("::llvm::StringRef::StringRef"),
      hasName("::llvm::Twine::Twine")))),
    argumentCountIs(1),
    hasArgument(
      0,
      id("call", memberCallExpr(
                                                  Method))),
        on(id("arg", expr()))))),
  &Callback);
```

These c'tors take a single argument.

2nd Matcher LLVM String Classes

```
const char *StringConstructor =
  "::std::basic_string<"
      "char, "
      "std::char_traits<char>, "
      "std::allocator<char> "
  ">::basic_string";

const char *StringCStrMethod =
  "::std::basic_string<"
    "char, "
    "std::char_traits<char>, "
    "std::allocator<char> "
  ">::c_str";
```

std::string Method Names

```
const char *StringConstructor =
  "::std::basic_string<"
      "char, "
      "std::char_traits<char>, "
      "std::allocator<char> "
  ">::basic_string";

const char *StringCStrMet
  "::std::basic_string<"
    "char, "
    "std::char_traits<char>, "
    "std::allocator<char> "
  ">::c_str";
```

The *real* name of std::string.

std::string Method Names

```
const char *StringConstructor =
  "::std::basic_string<"
      "char, "
      "std::char_traits<char>, "
      "std::allocator<char> "
  ">::basic_string";

const char *StringCStrMet
  "::std::basic_string<"
    "char, "
    "std::char_traits<char>, "
    "std::allocator<char> "
  ">::c_str";
```

The name of the constructor.

std::string Method Names

```
const char *StringConstructor =
  "::std::basic_string<"
      "char, "
      "std::char_traits<char>, "
      "std::allocator<char> "
  ">::basic_string";

const char *StringCStrMet
  "::std::basic_string<"
    "char, "
    "std::char_traits<char>, "
    "std::allocator<char> "
  ">::c_str";
```

The name of the c_str method.

std::string Method Names

1. Copy llvm/tools/clang/tools/extra/remove-cstr-calls directory to extra/remove-void-args
2. Rename RemoveCStrCalls.cpp to RemoveVoidArgs.cpp
3. Edit remove-void-args/CMakeLists.txt:
    1. Change remove-cstr-calls to remove-void-args
    2. Change RemoveCStrCalls.cpp to RemoveVoidArgs.cpp
4. Edit extra/CMakeLists.txt and add the line add_subdirectory(remove-void-args)
5. Test build

Bootstrapping an LLVM Tree Build

```
// test.cpp
int foo(void) {
    return 0;
}

int bar() {
    return 0;
}

int feezle(int i) {
    return 0;
}
```

Some Simple Test Cases

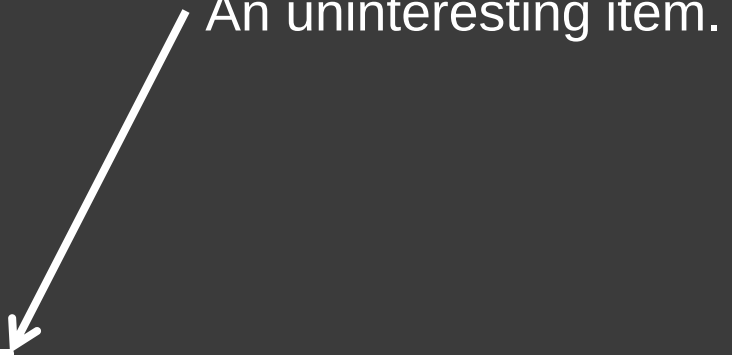
```
// test.cpp
int foo(void) {
    return 0;
}

int bar() {
    return 0;
}

int feezle(int i) {
    return 0;
}
```

Some Simple Test Cases

```
// test.cpp
int foo(void) {
    return 0;
}

int bar() {
    return 0;
}

int feezle(int i) {
    return 0;
}
```

Some Simple Test Cases

```
// test.cpp
int foo(void) {
    return 0;
}

int bar() {
    return 0;
}

int feezle(int i) {
    return 0;
}
```



Some Simple Test Cases

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <line:1:15, line:3:1>
|   `-ReturnStmt 0x469cf0 <line:2:5, col:12>
|     `-IntegerLiteral 0x469cd0 <col:12> 'int' 0
|-FunctionDecl 0x469d40 <line:5:1, line:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <line:6:5, col:12>
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <col:19, line:11:1>
    `-ReturnStmt 0x469f28 <line:10:5, col:12>
      `-IntegerLiteral 0x469f08 <col:12> 'int' 0
```

Dumping the AST

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <lin
|   `-ReturnStmt 0x469cf0 <lin
|     `-IntegerLiteral 0x469cd
|-FunctionDecl 0x469d40 <line:
| `-CompoundStmt 0x469de0 <lin
|   `-ReturnStmt 0x469dd0 <lin
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <col:19, line:11:1>
    `-ReturnStmt 0x469f28 <line:10:5, col:12>
      `-IntegerLiteral 0x469f08 <col:12> 'int' 0
```

You can dump the AST from the command line! That is so cool!

...and it even works on Windows!

Dumping the AST

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <lin
|   `-ReturnStmt 0x469cf0 <lin
|     `-IntegerLiteral 0x469cd
|-FunctionDecl 0x469d40 <line:5:1, line:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <line:6:5, col:12>
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <col:19, line:11:1>
    `-ReturnStmt 0x469f28 <line:10:5, col:12>
      `-IntegerLiteral 0x469f08 <col:12> 'int' 0
```

The mother of all nodes is a translation unit declaration.

Dumping the AST

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <line:1:15, line:3:1>
|   `-ReturnStmt 0x469cf0 <line:2:5, col:12>
|     `-IntegerLiteral 0x469cd0 <col:12> 'int' 0
|-FunctionDecl 0x469d40 <line:5:1, line:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <line:6:5, col:12>
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <co
    `-ReturnStmt 0x469f28 <li
      `-IntegerLiteral 0x469
```

Functions appear as a FunctionDecl node, followed by a CompoundStmt for the function body.
This one is for `int foo(void)`

Dumping the AST

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <line:1:15, line:3:1>
|   `-ReturnStmt 0x469cf0 <line:2:5, col:12>
|     `-IntegerLiteral 0x469cd0 <col:12> 'int' 0
|-FunctionDecl 0x469d40 <line:5:1, line:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <li
|     `-IntegerLiteral 0x469c
`-FunctionDecl 0x469e90 <line
  |-ParmVarDecl 0x469e10 <lin
  `-CompoundStmt 0x469f38 <co
    `-ReturnStmt 0x469f28 <li
      `-IntegerLiteral 0x469f
```

Every node is associated with a source range spanning the entire source text parsed into the node.
This source range is in test.cpp from line 1, character 1 to line 3, character 1.

Dumping the AST

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <line:1:15, line:3:1>
|   `-ReturnStmt 0x469cf0 <line:2:5, col:12>
|     `-IntegerLiteral 0x469cd0 <col:12> 'int' 0
|-FunctionDecl 0x469d40 <line:5:1, line:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <line:6:5, col:12>
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <c
    `-ReturnStmt 0x469f28 <l
      `-IntegerLiteral 0x469f
```

The FunctionDecl node for `int bar()`

Dumping the AST

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
TranslationUnitDecl 0x469850 <<invalid sloc>>
|-TypedefDecl 0x469b40 <<invalid sloc>> __builtin_va_list 'char *'
|-CXXRecordDecl 0x469b70 <<built-in>:28:1, col:7> class type_info
|-FunctionDecl 0x469c60 <test.cpp:1:1, line:3:1> foo 'int (void)'
| `-CompoundStmt 0x469d00 <line:1:15, line:3:1>
|   `-ReturnStmt 0x469cf0 <line:2:5, col:12>
|     `-IntegerLiteral 0x469cd0 <col:12> 'int' 0
|-FunctionDecl 0x469d40 <line:5:1, line:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <line:6:5, col:12>
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <co
    `-ReturnStmt 0x469f28 <li
      `-IntegerLiteral 0x469f
```

The FunctionDecl node for `int feezle(int)`

Dumping the AST

Dumping the AST

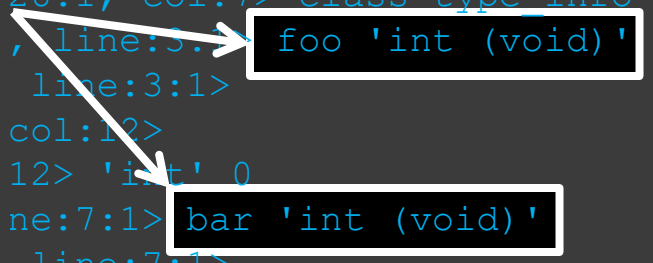
```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
                                                d sloc>>
                                                >> __builtin_va_list 'char *'
                                                28:1, col:7> class type_info
                                                , line:3:1> foo 'int (void)'
                                                 line:3:1>
                                                col:12>
                                                12> 'int' 0
                                                ne:7:1> bar 'int (void)'
| `-CompoundStmt 0x469de0 <line:5:11, line:7:1>
|   `-ReturnStmt 0x469dd0 <line:6:5, col:12>
|     `-IntegerLiteral 0x469db0 <col:12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <col:19, line:11:1>
    `-ReturnStmt 0x469f28 <line:10:5, col:12>
      `-IntegerLiteral 0x469f08 <col:12> 'int' 0
```

Both FunctionDecl nodes for foo and bar printed out the (void) signature.

What gives?

Dumping the AST

clang prints out a summary of the node after the source location. When clang prints a summary of a FunctionDecl, it prints what it thinks is a canonical function signature and uses (void) for all functions with no arguments.



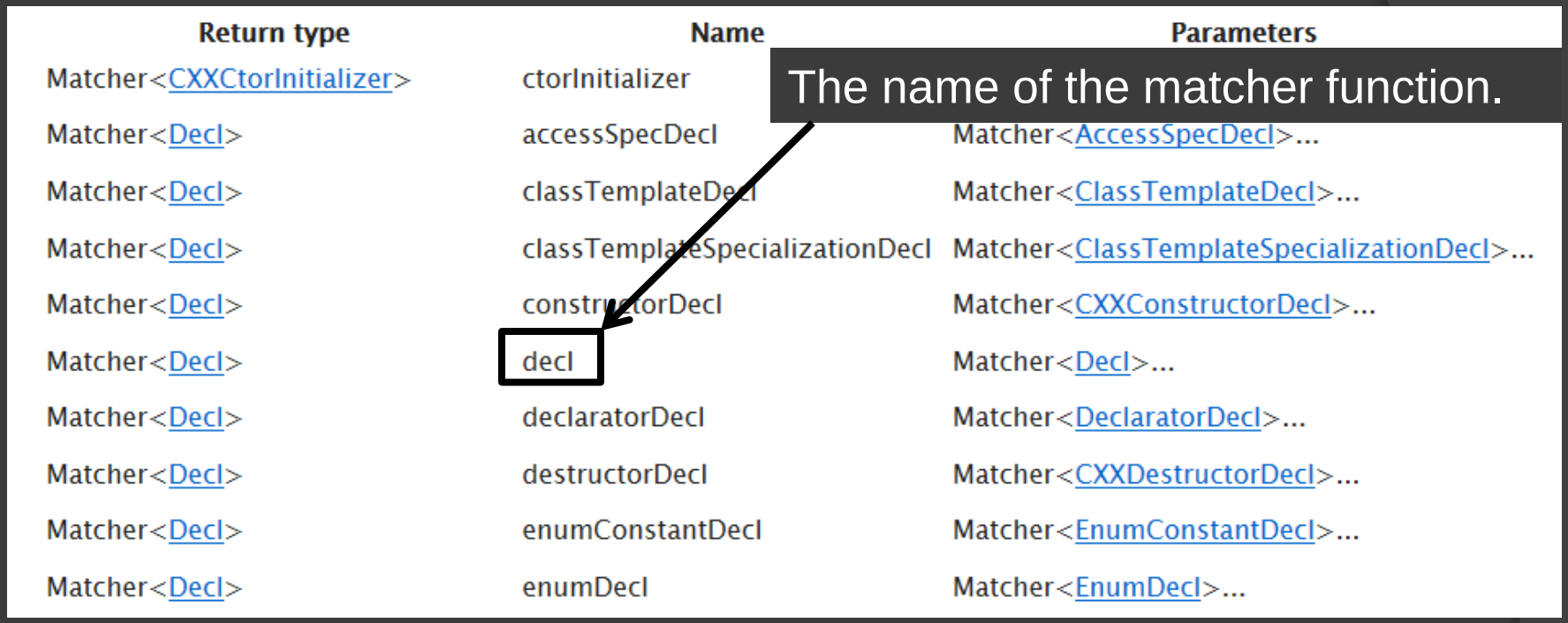

```
> clang -Xclang -ast-dump -fsyntax-only test.cpp
                                                d sloc>>
                                >>  __builtin_va_list 'char *'
                                28:1, col:7> class type_info
                                , line:3:1> foo 'int (void)'
                                 line:3:1>
                                col:12>
                                12> 'int' 0
                                ne:7:1> bar 'int (void)'
                                 line:7:1>
                                col:12>
                                12> 'int' 0
`-FunctionDecl 0x469e90 <line:9:1, line:11:1> feezle 'int (int)'
  |-ParmVarDecl 0x469e10 <line:9:12, col:16> i 'int'
  `-CompoundStmt 0x469f38 <col:19, line:11:1>
    `-ReturnStmt 0x469f28 <line:10:5, col:12>
      `-IntegerLiteral 0x469f08 <col:12> 'int' 0
```

Three Kinds of Matchers

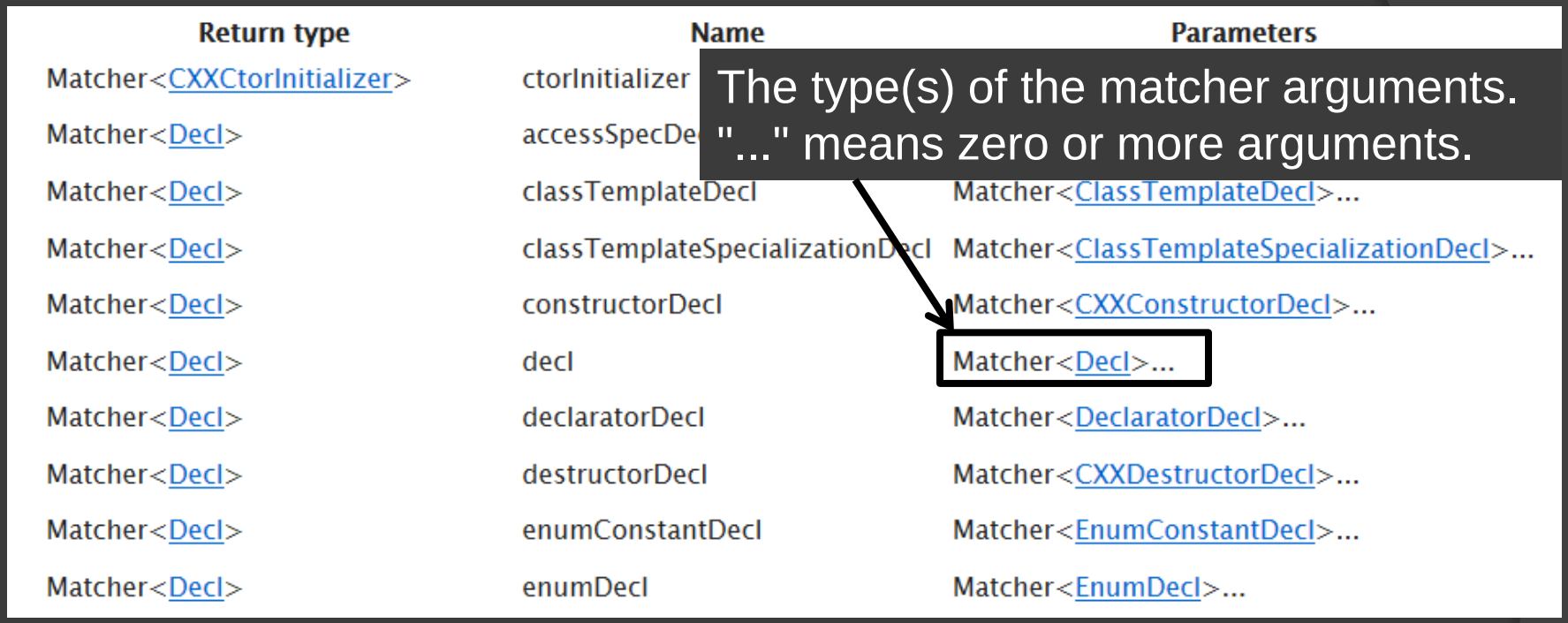
- *Node* matchers match a specific node type
  - constructorDecl, fieldDecl, varDecl, etc.
- *Narrowing* matchers match attributes on nodes
  - argumentCountIs, isConst, isVirtual, etc.
- *Traversal* matchers follow node relationships
  - hasAncestor, hasDescendent, pointee, callee

Return type	Name	Parameters
Matcher<CXXCtorInitializer>	ctorInitializer	Matcher<CXXCtorInitializer>...
Matcher<Decl>	accessSpecDecl	Matcher<AccessSpecDecl>...
Matcher<Decl>	classTemplateDecl	Matcher<ClassTemplateDecl>...
Matcher<Decl>	classTemplateSpecializationDecl	Matcher<ClassTemplateSpecializationDecl>...
Matcher<Decl>	constructorDecl	Matcher<CXXConstructorDecl>...
Matcher<Decl>	decl	Matcher<Decl>...
Matcher<Decl>	declaratorDecl	Matcher<DeclaratorDecl>...
Matcher<Decl>	destructorDecl	Matcher<CXXDestructorDecl>...
Matcher<Decl>	enumConstantDecl	Matcher<EnumConstantDecl>...
Matcher<Decl>	enumDecl	Matcher<EnumDecl>...

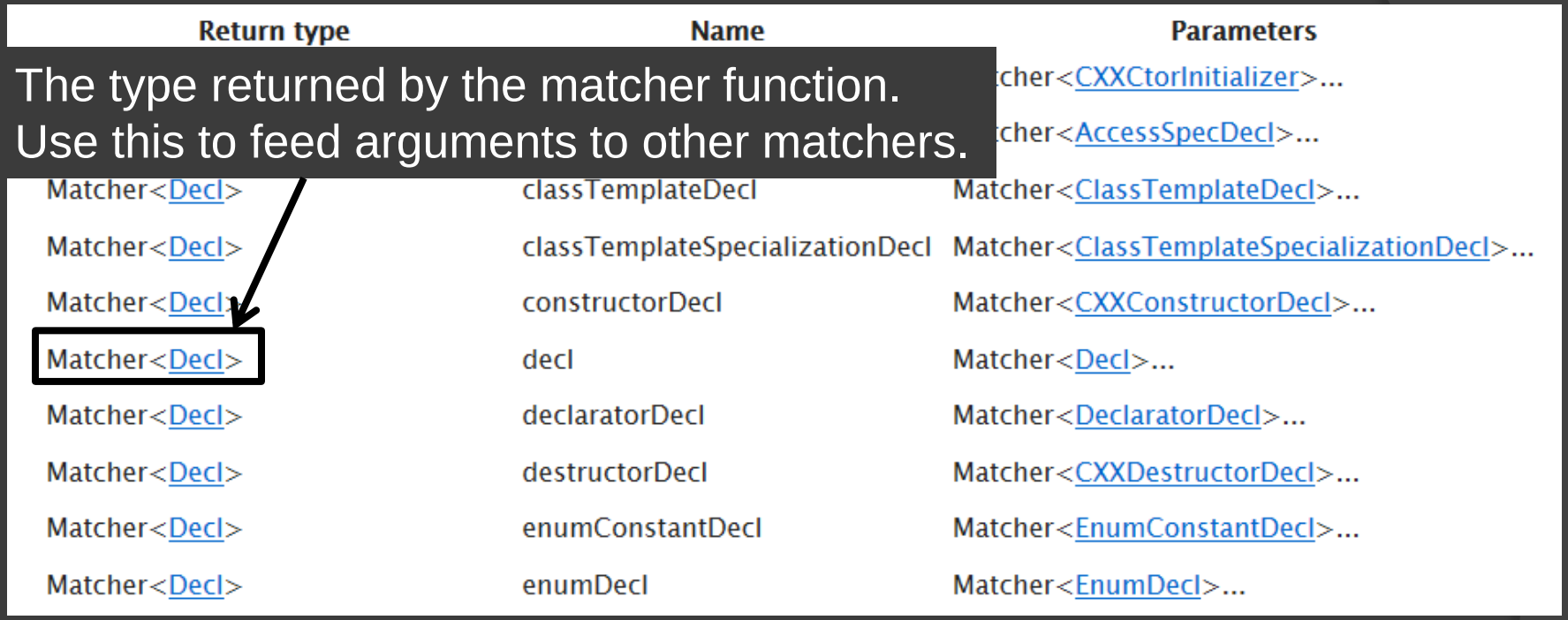
Traversing the Matcher Reference



Traversing the Matcher Reference

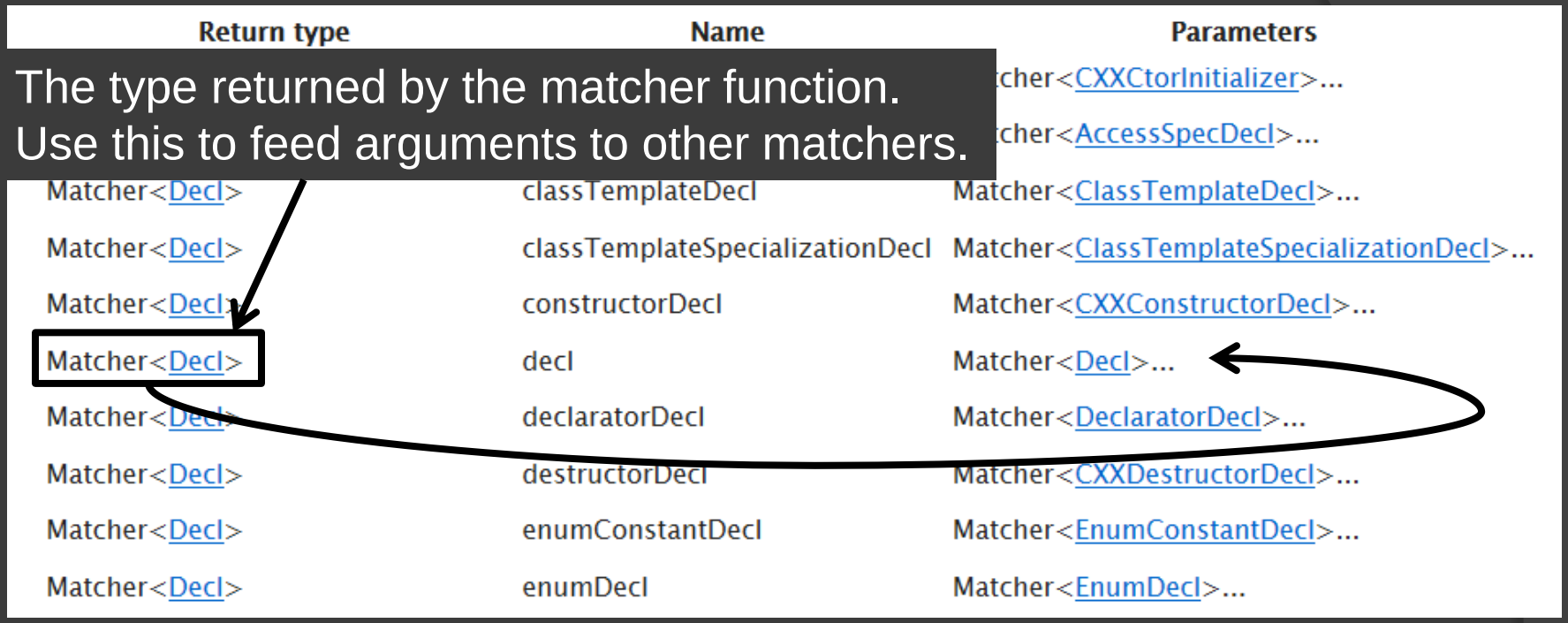


Traversing the Matcher Reference



Return type	Name	Parameters
		cher<CXXCtorInitializer>...
		cher<AccessSpecDecl>...
Matcher<Decl>	classTemplateDecl	Matcher<ClassTemplateDecl>...
Matcher<Decl>	classTemplateSpecializationDecl	Matcher<ClassTemplateSpecializationDecl>...
Matcher<Decl>	constructorDecl	Matcher<CXXConstructorDecl>...
Matcher<Decl>	decl	Matcher<Decl>...
Matcher<Decl>	declaratorDecl	Matcher<DeclaratorDecl>...
Matcher<Decl>	destructorDecl	Matcher<CXXDestructorDecl>...
Matcher<Decl>	enumConstantDecl	Matcher<EnumConstantDecl>...
Matcher<Decl>	enumDecl	Matcher<EnumDecl>...

Traversing the Node Matchers



Apply this process repeatedly to navigate acceptable matcher arguments and build larger matcher expressions.

Traversing the Node Matchers

- Each matcher has doxygen documentation linked from the AST Matcher Reference page
- ...it doesn't hurt to consult the source; almost everything about matchers is implemented in the header

Understanding Matchers in Detail

Compilation Database

- When refactoring C++ code, we need to take into account the entire preprocessor context
- This comes from the compiler command line:
  - -D defines symbols
  - -I modifies include search path
  - etc.
- The compilation database holds command lines for every source file we're processing.

Compilation Database

- JSON array of objects containing:
  - "directory" - The directory containing the source file
  - "command" - The command used to compile the file
  - "file" - The source filename
- CMake can generate these, yay!
  - ...but not on Windows, boo! (CMake patch in progress?)
- But you can easily create one in an editor or from a script

```
[
  {
    "directory":
"D:/Code/clang/llvm/tools/clang/tools/e
xtra/remove-void-args",
    "command": "CL.exe /c
/I\"D:/Code/clang/tools/clang/tools/ext
ra/remove-void-args\"
\"D:/Code/clang/llvm/tools/clang/tools/
extra/remove-void-args/test.cpp\"",
    "file": "test.cpp"
  }
]
```

Example Compilation Database

```
[
  {
    "directory":
"D:/Code/clang/llvm/tools/clang/tools/e
xtra/remove-void-args",
    "command": "CL.exe /c
/I\"D:/Code/clang/tools/clang/tools/ext
ra/remove-void-args\"
\                          tools/clang/tools/
e                         s/test.cpp\"",
                          ,"
  }
]
```

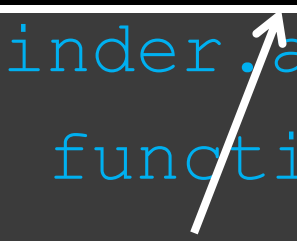
Make sure you use /'s as path separators, even on Windows, because \ is a JSON string escape. Alternatively, you could (ugh) double up all the \'s.

Example Compilation Database

```
// RemoveVoidArgs.cpp
FixVoidArg Callback(
  &Tool.getReplacements());
Finder.addMatcher(
  functionDecl(parameterCountIs(0))
  .bind("fn"),
  &Callback);
```

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
FixVoidArg Callback(
   &Tool.getReplacements());
Finder.addMatcher(
   functionDecl(parameterCountIs(0))
   &Callback);
```



Instantiate our refactoring callback

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
FixVoidArg
  &Tool.getReplacements());
```

Match FunctionDecls with no parameters and bind to "fn"

```
Finder.addMatcher(
  functionDecl(parameterCountIs(0))
  .bind("fn"),
  &Callback);
```

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg : public
    ast_matchers::MatchFinder::MatchCallback {
 public:
  FixVoidArg(tooling::Replacements *Replace)
      : Replace(Replace) {}

  // ...

 private:
  // ...
  tooling::Replacements *Replace;
};
```

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg : public
    ast_matchers::MatchFinder::MatchCallback {
 public:
  FixVoidArg(tooling::Replacements *Replace)
      : Replace(Replace) {}

  // ...

 private:
  // ...
  tooling::Replacements *Replace;
};
```

Some basic boiler plate needed by every refactoring tool: implement MatchCallback and keep a pointer to a replacement list.

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
    BoundNodes Nodes = Result.Nodes;
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
            std::string::size_type OpenBrace = Text.find_first_of('{');
            if (OpenBrace == std::string::npos) { return; }
            std::string::size_type EndOfDecl =
              Text.find_last_of(')', OpenBrace) + 1;
            std::string Decl = Text.substr(0, EndOfDecl);
            if (Decl.length() > 6
                && Decl.substr(Decl.length()-6) == "(void)") {
                std::cout << "Void Definition : "
                  << getLocation(SM, Function) << Decl << "\n";
            }
        }
    }
  }
};
```

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
    BoundNodes Nodes = Result.Nodes;
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
            std::
            if (O
            std::
              Tex
            std::
            if (Decl.length() > 6
                && Decl.substr(Decl.length()-6) == "(void)") {
                std::cout << "Void Definition : "
                  << getLocation(SM, Function) << Decl << "\n";
            }
        }
    }
  }
};
```

This method is invoked by the MatchFinder when our Matcher matches a node in the AST. It gives us the MatchResult for the matched node.

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
    BoundNodes Nodes = Result.Nodes;
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
            s
            i
            std::string::size_type EndOfDecl =
              Text.find_last_of(')', OpenBrace) + 1;
            std::string Decl = Text.substr(0, EndOfDecl);
            if (Decl.length() > 6
                && Decl.substr(Decl.length()-6) == "(void)") {
                std::cout << "Void Definition : "
                  << getLocation(SM, Function) << Decl << "\n";
            }
        }
    }
  }
};
```

Nodes bound to identifiers in the matching result set.

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
    BoundNodes Nodes = Result.Nodes;
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
            s
            i
            s
            std::string Decl = Text.substr(0, EndOfDecl);
            if (Decl.length() > 6
                && Decl.substr(Decl.length()-6) == "(void)") {
                std::cout << "Void Definition : "
                  << getLocation(SM, Function) << Decl << "\n";
            }
        }
    }
  }
};
```

The SourceManager is how we get source text associated with nodes in the AST.

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
    BoundNodes Nodes = Result.Nodes;
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
      std::string const Text = getText(*SM, *Function);
      if (Text.length() > 0) {
        std::string::size_type OpenBrace = Text.find_first_of('{');
        if (OpenBrace == std::string::npos) { return; }
        std::string::size_type EndOfDecl =
          Text.find_last_of(')', OpenBrace) + 1;
        std::string Decl = Text.substr(0, EndOfDecl);
        i
          std::cout << "Void Definition : "
            << getLocation(SM, Function) << Decl << "\n";
        }
      }
    }
  }
};
```

If we matched a FunctionDecl bound to "fn", then...

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
    BoundNodes Nodes = Result.Nodes;
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
      std::string const Text = getText(*SM, *Function);
      if (Text.length() > 0) {
        std::string::size_type OpenBrace = Text.find_first_of('{');
        if (OpenBrace == std::string::npos) { return; }
        std::string::size_type EndOfDecl =
          Text.find_last_of(')', OpenBrace) + 1;
        std::string Decl = Text.substr(0, EndOfDecl);
        i
          std::cout << "Void Definition : "
            << getLocation(SM, Function) << Decl << "\n";
        }
      }
    }
  }
};
```

Get the source text associated with the FunctionDecl

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVc
  virtual                                                    {
    BoundN
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
          std::string::size_type OpenBrace = Text.find_first_of('{');
          if (OpenBrace == std::string::npos) { return; }
          std::string::size_type EndOfDecl =
            Text.find_last_of(')', OpenBrace) + 1;
          std::string Decl = Text.substr(0, EndOfDecl);
          if (Decl.length() > 6
              && Decl.substr(Decl.length()-6) == "(void)") {
              std::cout << "Void Definition : "
                << getLocation(SM, Function) << Decl << "\n";
          }
        }
    }
  }
};
```

Heuristically, the function signature is everything up to the last) appearing before the first {

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual
    BoundN
    SourceManager const *SM = Result.SourceManager;
    if (FunctionDecl const *const Function =
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
            std::string::size_type OpenBrace = Text.find_first_of('{');
            if (OpenBrace == std::string::npos) { return; }
            std::string::size_type EndOfDecl =
              Text.find_last_of(')', OpenBrace) + 1;
            std::string Decl = Text.substr(0, EndOfDecl);
            if (Decl.length() > 6
                && Decl.substr(Decl.length()-6) == "(void)") {
                std::cout << "Void Definition : "
                  << getLocation(SM, Function) << Decl << "\n";
            }
        }
    }
  }
};
```

If the signature ends with (void), then print it.

Explore Decls in Simple test.cpp

```
// RemoveVoidArgs.cpp
class FixVoidArg { public:
  virtual
    BoundN
    Source
    if (Fu
          Nodes.getNodeAs<FunctionDecl>("fn")) {
        std::string const Text = getText(*SM, *Function);
        if (Text.length() > 0) {
            std::string::size_type OpenBrace = Text.find_first_of('{');
            if (OpenBrace == std::string::npos) { return; }
            std::string::size_type EndOfDecl =
              Text.find_last_of(')', OpenBrace) + 1;
            std::string Decl = Text.substr(0, EndOfDecl);
            if (Decl.length() > 6
                && Decl.substr(Decl.length()-6) == "(void)") {
                std::cout << "Void Definition : "
                  << getLocation(SM, Function) << Decl << "\n";
            }
        }
    }
  }
};
```

Helper method that gets the location of the matched node as a printable string.

Explore Decls in Simple test.cpp

```
// test.cpp
int foo(void) {
    return 0;
}

int bar() {
    return 0;
}

int feezle(int i) {
    return 0;
}
```

Some Simple Test Cases

```
> remove-void-args . test.cpp
Void Definition : test.cpp(1): int foo(void)
```

Explore Decls in Simple test.cpp

- Our test file is not realistic
- What happens if we include <cstdio>?

Explore Realistic Decls

```
// test.cpp
#include <cstdio>

int foo(void) {
    return 0;
}

int bar() {
    return 0;
}

int feezle(int i) {
    return 0;
}
```

Explore Realistic Decls

```
> remove-void-args . test.cpp
Void Definition : crtdefs.h(571): void __cdecl
_invalid_parameter_noinfo(void)
Void Definition : crtdefs.h(572): __declspec(noreturn)
    void __cdecl _invalid_parameter_noinfo_noreturn(void)
Void Definition : stdio.h(129): FILE * __cdecl __iob_func(void)
Void Definition : stdio.h(184): int __cdecl _fcloseall(void)
Void Definition : stdio.h(196): int __cdecl _fgetchar(void)
Void Definition : stdio.h(217): int __cdecl _flushall(void)
Void Definition : stdio.h(255): int __cdecl getchar(void)
Void Definition : stdio.h(256): int __cdecl _getmaxstdio(void)
Void Definition : stdio.h(289): int __cdecl _rmtmp(void)
Void Definition : stdio.h(302):
  unsigned int __cdecl _get_output_format(void)
Void Definition : stdio.h(372): int __cdecl _get_printf_count_output(void)
Void Definition : stdio.h(422): wint_t __cdecl _fgetwchar(void)
Void Definition : stdio.h(426): wint_t __cdecl getwchar(void)
Void Definition : test.cpp(3): int foo(void)
```

Explore Realistic Decls

```
> remove-void-args . test.cpp
Void Definition : crtdefs.h(571): void __cdecl
_invalid_parameter_noinfo(void)
Void Definition : crtdefs.h(572): __declspec(noreturn)
    void __cdecl _invalid_parameter_noinfo_noreturn(void)
Void Definition : stdio.h(129): FILE * __cdecl __iob_func(void)
Void Definition : stdio.h(184): int __cdecl _fcloseall(void)
Void Definition : stdio.h(196): int __cdecl _fgetchar(void)
Void Definition : stdio.h(217): int __cdecl _flushall(void)
Void Definition : stdio.h(255): int __cdecl getchar(void)
Void Definition : stdio.h(256): int __cdecl _getmaxstdio(void)
Void Definition : stdio.h(289): int __cdecl _rmtmp(void)
Void Definition : stdio.h(302):
  unsigned int __cdecl _get_output_format(void)
Void Definition : stdio.h(372): int __cdecl _get_printf_count_output(void)
Void Definition : stdio.h(422): wint_t __cdecl _fgetwchar(void)
Void Definition : stdio.h(426): wint_t __cdecl getwchar(void)
Void Definition : test.cpp(3): int foo(void)
```

All these functions are extern "C". No, thanks!

Explore Realistic Decls

```
virtual void run(/* ... */) {
  BoundNodes Nodes = Result.Nodes;
  SourceManager const *SM =
    Result.SourceManager;
  if (FunctionDecl const *const Function =
      Nodes.getNodeAs<FunctionDecl>("fn")) {
    if (Function->isExternC()) {
      return;
    }
    // ...
  }
}
```

Explore Realistic Decls

```
> remove-void-args . test.cpp
Void Definition : test.cpp(3): int foo(void)
```

Explore Realistic Decls

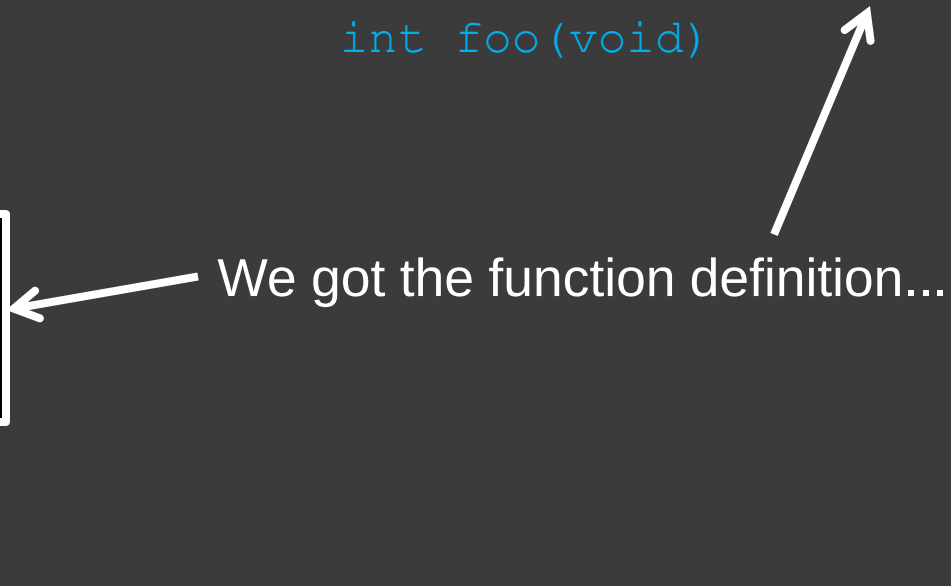
```
// test.cpp
#include <cstdio>

int foo(void);

int foo(void) {
    return 0;
}
int bar() {
    return 0;
}
int feezle(int i) {
    return 0;
}
```

```
> remove-void-args . test.cpp
Void Definition : test.cpp(5):
    int foo(void)
```

Identifying Function Declarations

```
// test.cpp
#include <cstdio>

int foo(void);

int foo(void) {
    return 0;
}
int bar() {
    return 0;
}
int feezle(int i) {
    return 0;
}
```

```
> remove-void-args . test.cpp
Void Definition : test.cpp(5):
    int foo(void)
```



Identifying Function Declarations

```
// test.cpp
#include <cstdio>

int foo(void);

int foo(void) {
    return 0;
}
int bar() {
    return 0;
}
int feezle(int i) {
    return 0;
}
```

```
> remove-void-args . test.cpp
Void Definition : test.cpp(5):
    int foo(void)
```

Identifying Function Declarations

```
std::string const Text = getText(*SM, *Function);
if (!Function->isThisDeclarationADefinition()) {
    if (Text.length() > 6
        && Text.substr(Text.length()-6) == "(void)") {
        std::cout << "Void Declaration: "
          << getLocation(SM, Function) << Text << "\n";
    }
} else if (Text.length() > 0) {
    std::string::size_type EndOfDecl =
      Text.find_last_of(')', Text.find_first_of('{')) + 1;
    std::string Decl = Text.substr(0, EndOfDecl);
    if (Decl.length() > 6
        && Decl.substr(Decl.length()-6) == "(void)") {
        std::cout << "Void Definition : "
          << getLocation(SM, Function) << Decl << "\n";
    }
}
```

Identifying Function Declarations

```
std::string const Text = getText(*SM, *Function);
if (!Function->isThisDeclarationADefinition()) {
    if (Text.length() > 6
        && Text.substr(Text.length()-6) == "(void)") {
        std::cout << "Void Declaration: "
          << getLocation(SM, Function) << Text << "\n";
    }
} else if (Text.length(
    std::string::size_t
      Text.find_last_of
    std::string Decl = T
    if (Decl.length() >
        && Decl.substr(
        std::cout << "Vo
          << getLocatio
    }
}
```

Tells us if this node is a definition or a declaration of a function.

For detailed work, get familiar with the methods on the AST node classes. The AST matcher reference links to doxygen pages for the node classes.

Identifying Function Declarations

```
> remove-void-args . test.cpp
Void Declaration: test.cpp(3): int foo(void)
Void Definition : test.cpp(5): int foo(void)
```

Identifying Function Declarations

```
if (!Function->isThisDeclarationADefinition()) {
  if (Text.length() > 6
      && Text.substr(Text.length()-6) == "(void)") {
    std::string const NoVoid = Text.substr(0, Text.length()-6) + "()";
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
} else if (Text.length() > 0) {
  std::string::size_type EndOfDecl =
    Text.find_last_of(')', Text.find_first_of('{')) + 1;
  std::string Decl = Text.substr(0, EndOfDecl);
  if (Decl.length() > 6 && Decl.substr(Decl.length()-6) == "(void)") {
    std::string NoVoid = Decl.substr(0, Decl.length()-6) + "()"
      + Text.substr(EndOfDecl);
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
}
```

Replacing Identified Source

```
if (!Function->isThisDeclarationADefinition()) {
  if (Text.length() > 6
      && Text.substr(Text.length()-6) == "(void)") {
    std::string const NoVoid = Text.substr(0, Text.length()-6) + "()";
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
} else if (Text.length() > 0) {
  std::string::size_type EndOfDecl =
    Text.find_last_of(')', Text.find_first_of('{')) + 1;
  std::string Decl = Text.substr(0, EndOfDecl);
  if (Decl.length() > 6 && Decl.substr(Decl.length()-6) == "(void)") {
    std::string NoVoid = Decl.substr(0, Decl.length()-6) + "()"
      + Text.substr(EndOfDecl);
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
}
```

Get the new text for this node:
the declaration minus the "(void)"

Replacing Identified Source

```
if (!Function->isThisDeclarationADefinition()) {
  if (Text.length() > 6
      && Text.substr(Text.length()-6) == "(void)") {
    std::string const NoVoid = Text.substr(0, Text.length()-6) + "()";
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
} else if (Text.length() > 0) {
  std::string::size_type EndOfDecl =
    Text.find_last_of(')', Text.find_first_of('{')) + 1;
  std::string Decl = Text.substr(0, EndOfDecl);
  if (Decl.length() > 6 && Decl.substr(Decl.length()-6) == "(void)") {
    std::string NoVoid = Decl.substr(0, Decl.length()-6) + "()"
      + Text.substr(EndOfDecl);
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
}
```

Build a Replacement for the Function node with the new text of NoVoid

Replacing Identified Source

```
if (!Function->isThisDeclarationADefinition()) {
  if (Text.length() > 6
      && Text.substr(Text.length()-6) == "(void)") {
    std::string const NoVoid = Text.substr(0, Text.length()-6) + "()";
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
} else if (Text.length() > 0) {
  std::string::size_type EndOfDecl =
    Text.find_last_of(')', Text.find_first_of('{')) + 1;
  std::string Decl = Text.substr(0, EndOfDecl);
  if (Decl.length() > 6 && Decl.substr(Decl.length()-6) == "(void)") {
    std::string NoVoid = Decl.substr(0, Decl.length()-6) + "()"
      + Text.substr(EndOfDecl);
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
}
```

Insert the replacement on the list

Replacing Identified Source

```
if (!Function->isThisDeclarationADefinition()) {
  if (Text.length() > 6
      && Text.substr(Text.length()-6) == "(void)") {
    std::string const NoVoid = Text.substr(0, Text.length()-6) + "()";
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
} else if (Text.length() > 0) {
  std::string::size_type EndOfDecl =
    Text.find_last_of(')', Text.find_first_of('{')) + 1;
  std::string Decl = Text.substr(0, EndOfDecl);
  if (Decl.length() > 6 && Decl.substr(Decl.length()-6) == "(void)") {
    std::string NoVoid = Decl.substr(0, Decl.length()-6) + "()"
      + Text.substr(EndOfDecl);
    Replace->insert(Replacement(*Result.SourceManager, Function, NoVoid));
  }
}
```

Do the same thing with a function definition.

Replacing Identified Source

```
> remove-void-args . test.cpp
> type test.cpp
#include <cstdio>
int foo();
int foo() {
    return 0;
}
int bar() {
    return 0;
}
int feezle(int i) {
    return 0;
}
```

Replacing Identified Source

- Lather
- Rinse
- Repeat

Shampoo Algorithm

Shampoo Algorithm

- Lather

  Write a matcher for the next AST construct you want to handle.

- Rinse

  Build replacements for the newly matched construct.

- Repeat

  Iterate until all language constructs are handled appropriately.

Discriminate Modifiable Files

- At some point you may end up matching nodes in the standard headers (<cstdio>) or third-party headers (<boost/scope_exit.hpp>)
- You will want to discriminate between "stable" files and modifiable files

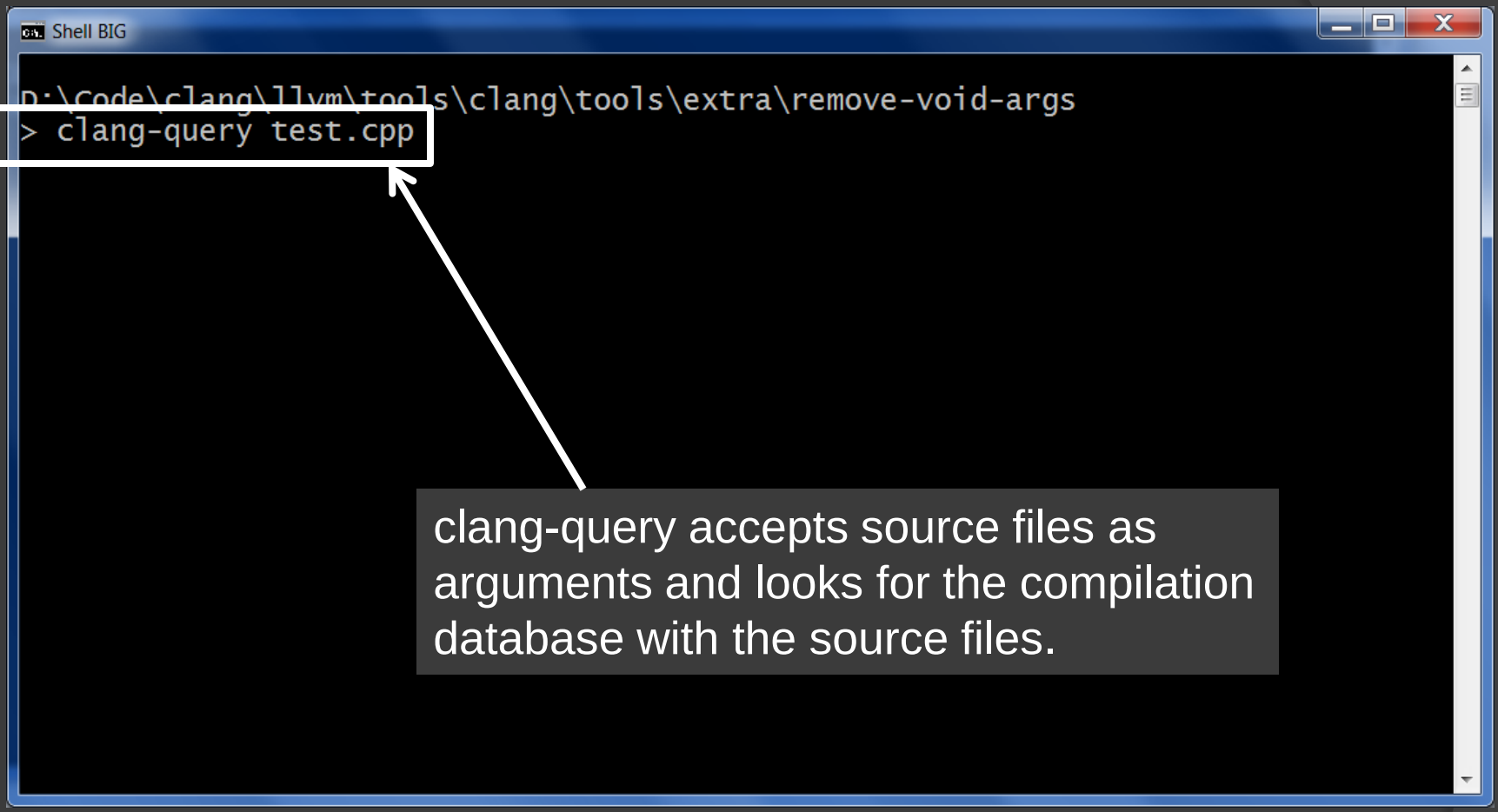
```
static bool modifiableFile(
    SourceManager const *SM, SourceLocation loc) {
  std::string fileName = SM->getFilename(loc).str();
  return fileName.length() >= 8 &&
    (fileName.substr(fileName.length()-8)
     == "test.cpp");
}
```

Discriminate Modifiable Files

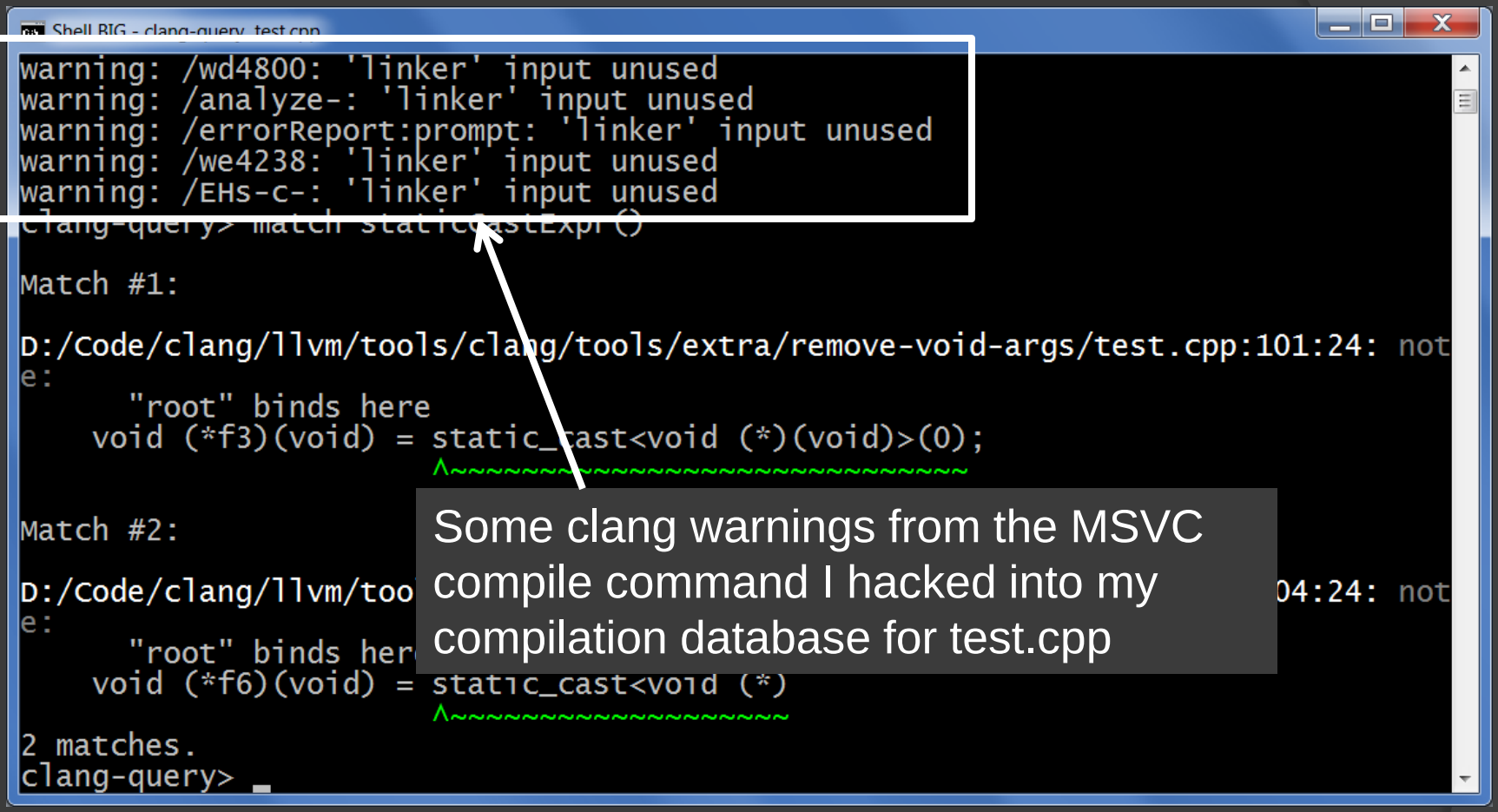
```
static bool modifiableFile(
    SourceManager const *SM, SourceLocation loc) {
  std::string fileName = SM->getFilename(loc).str();
  return fileName.length() >= 8 &&
    (fileName.substr(fileName.length()-8)
     == "test.cpp");
}
```

SourceManager knows how to find the file associated with a location.

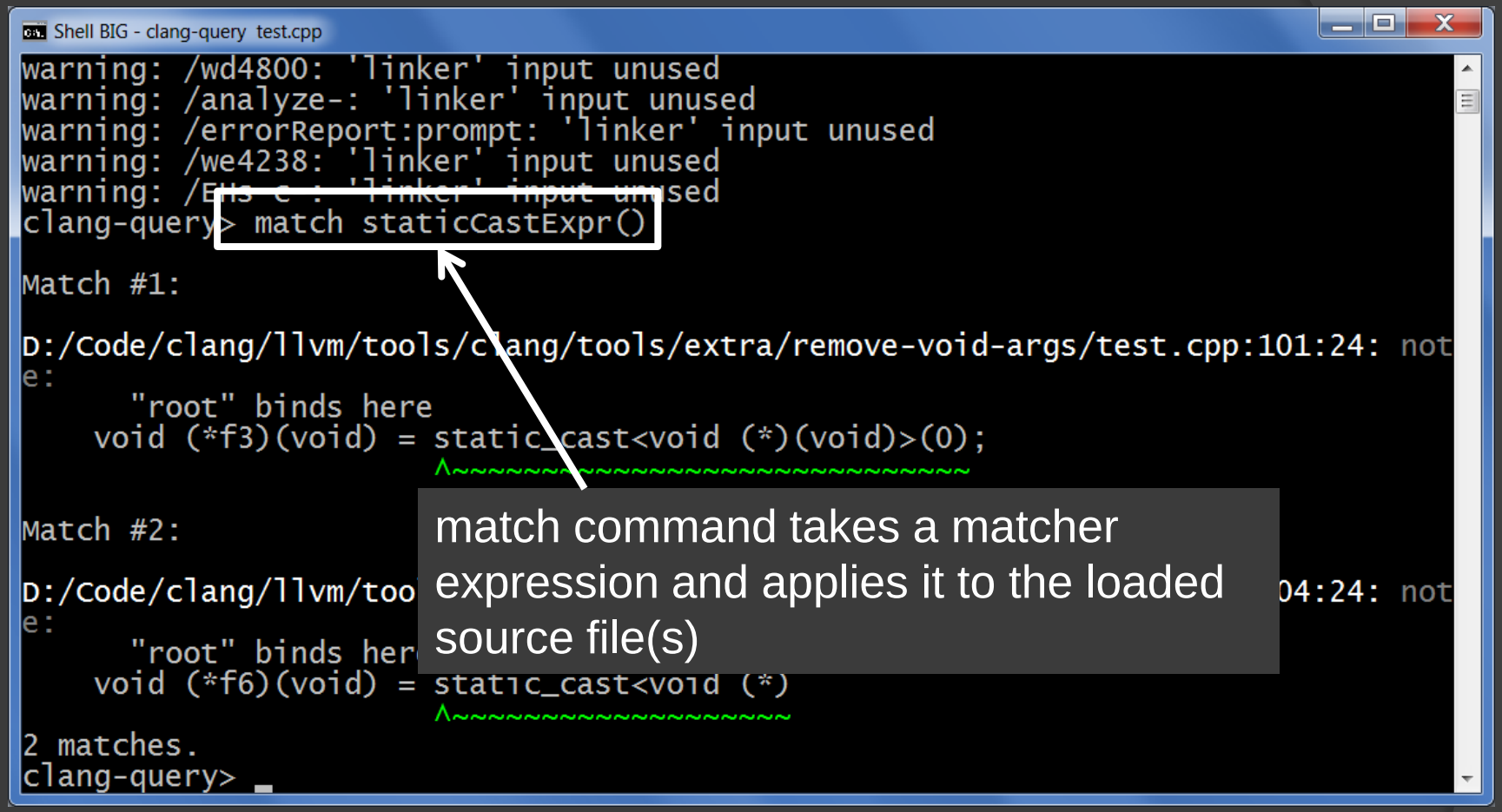
Discriminate Modifiable Files



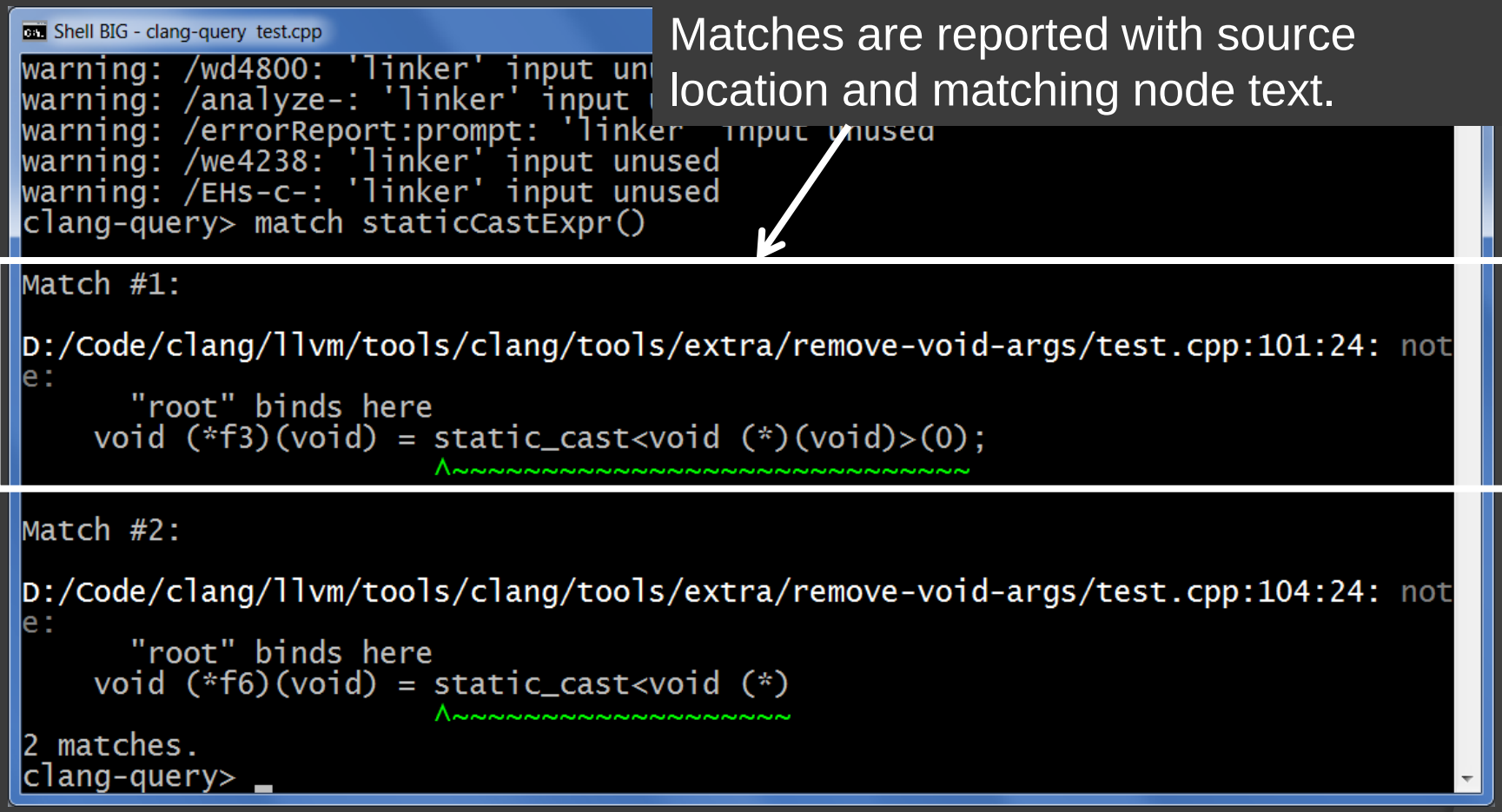
Interactive AST Query: clang-query



Interactive AST Query: clang-query



Interactive AST Query: clang-query



Interactive AST Query: clang-query

Resources

- Clang AST Matchers Reference http://clang.llvm.org/docs/LibASTMatchersReference.html
- C++ Refactoring Test Suite http://legalizeadulthood.wordpress.com/2010/02/02/c-refactoring-tools-test-suite-available/
- Some C/C++ Specific Refactorings: http://legalizeadulthood.wordpress.com/category/computers/programming/refactoring/

Room for Improvement

- Packaging is not uniform across platforms
- We shouldn't have to build clang ourselves
  - Need out of tree build recipes
- clang-query grammar needs documentation
- No tutorial for IDE integration
- Let's collaborate!

Recap

- The hard part is done for us by clang
- Get started by copying an existing tool
- Build a source file test suite
- Start with simple matches against the AST
- Build appropriate replacements
- Incrementally refine and extend matching
- Use clang-query to prototype matchers
- Let's collaborate to make this even easier!