Large Scale Refactoring in C++ with clang-tidy

Tristan Stevens

Agenda

- Me and my experience
- Introduction to clang-tidy and the AST
- Basics of clang-tidy
- Using clang-query
- More complex matching
  - Type matching
- Rewriting
- Testing

About Me

- 4th year student at McGill
- Interested in robotics, C++, compilers, software design
- Originally from Seattle
- 3rd internship at Coveo!

Experience with clang-tidy

- 4-month internship with Kevin Lalumiere at Coveo on the Index Infrastructure team
- Main project – rewriting in house versions of std::unique_ptr and std::shared_ptr to std versions
  - 40,000 lines of automatic code changes
  - Challenging compared to what we could find online
    - Extracting types
    - Inferred types

Clang-tidy

- C++ “linter” tool
- Probably already seen it for things like programming errors, style violations, bugs that can be shown with static analysis
- Parses the AST to match certain conditions
- Has the ability to bind on certain nodes then rewrite the bound nodes
- Don’t need to recompile clang-tidy to add new rules
  - since clang 14 (2022)


CMakeLists.txt
main.cpp
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int* ptr = new int(5); // allocating memory without releasing it
6 return 0;
7 }
Problems
File 3
Project Errors
main.cpp ~/Code/clang-tidy-ex 3 problems
Local variable 'ptr' is only assigned but never accessed :5
Allocated memory is leaked :5
The value is never used :5

Why refactor with clang-tidy?

- Compared to find and replace:
- Specific
  - Bind to nodes with very specific criteria
    - Specific children
    - Specific parents
    - Specific types
  - Extract information from other parts of the code
    - Types
- Deal with inferred use cases
  - Implicit conversions
  - Pointer access vs non pointer access
- No collateral
  - If matchers are specific enough you won’t change anything else
- Relatively fast

The AST

- Tree based representation of compiled code
- Prebuilt matchers here https://clang.llvm.org/docs/LibASTMatchersReference.html
- Node for EVERYTHING
  - `returnStmt, integerLiteral, explicitCastExpr, cxxTemporaryObjectExpr, genericSelectionExpr, etc.`

Building a clang-tidy check

- 3 parts
  - Traversal
    - Matches a node and binds to it
    - Uses matchers
  - Transformer
    - Performs an action to said node
    - Uses stencils
  - Message
    - Warning message in IDE/console
    - Uses stencils
- Additional code to register a check

Basic clang-tidy check

```
auto createSimpleRule()
{
    return makeRule(declRefExpr(to(functionDecl(hasName("MkX")))),
        changeTo(cat("MakeX")),
        cat("MkX has been renamed MakeX"));
}
```

matchers

transformers

output message

Registering a check

```cpp
// Boilerplate

class SimpleCheck : public TransformerClangTidyCheck
{
public:
    SimpleCheck(StringRef Name, ClangTidyContext* Context)
     : TransformerClangTidyCheck(createSimpleRule(), Name, Context)
    {
    }
};

class SimpleCheckModule : public ClangTidyModule
{
public:
    void addCheckFactories(ClangTidyCheckFactories& CheckFactories) override
    {
        CheckFactories.registerCheck<SimpleCheck>("simplecheck-module");
    }
};

}  // namespace

namespace clang::tidy {

// Register the module using this statically initialized variable.
static ClangTidyModuleRegistry::Add<::SimpleCheckModule> SimpleCheckModuleInit(
    "simplecheck-module",
    "Adds 'simplecheckcheck' checks.");

// This anchor is used to force the linker to link in the generated object file and thus register the module.
volatile int SimpleCheckModuleAnchorSource = 0;
```

See an example at the bottom of this file https://github.com/coveooss/clang-tidy-plugin-examples/blob/main/src/replace_smartptr_by_std/ReplaceSmartPtrBySharedCheck.cpp

clang-query

- Command line tool to let you run matchers
- Loads up a single file with the `compile_commands.json` file and parses it as clang-tidy would
  - Outputted by cmake, turn on with a flag
- All the same matchers are available as clang-tidy
  - Can’t build your own matchers and run them in clang-query (you probably don’t need to do this anyways)
- Let's you easily see what matchers are available
  - Hit tab
- Headers and macro outputs are included code that is matched

Settings for clang-query/clang-tidy checks

- Make sure clang-tidy version and compiler version are the same
- Matching mode:
  - IgnoreUnlessSpelledInSource
    - Recommended traversal mode in docs, easier to use
      - Defaults to AsIs
    - Will be what you see written in the source code
    - Easier to write matchers as there will be less intermediaries
  - AsIs
    - If you need it, you need it
    - `ImplicitCastExpr`, Implicit constructions
- Setting matching mode
  - Clang-query: `set traversal IgnoreUnlessSpelledInSource`
  - In clang-tidy checks: `traverse(clang::TK_AsIs, matcher)`

Settings for clang-query

- Enable output
  - dump
    - Prints the ast tree
    - Useful for building more strict matchers
  - diag
    - Default, just prints out the text for matched nodes and bound nodes

	AsIs	IgnoreUnlessSpelledInSource

AST dump of `func1`:
 	
`struct B {`
`  B(int);`
`};`
 	
`B func1() { return 42; }`
	C++98 dialect:
	
	`FunctionDecl`
	`` `-CompoundStmt``
	`` `-ReturnStmt``
	`` `-ExprWithCleanups``
	`` `-CXXConstructExpr``
	`` `-MaterializeTemporaryExpr``
	`` `-ImplicitCastExpr``
	`` `-ImplicitCastExpr``
	`` `-CXXConstructExpr``
	`` `-IntegerLiteral 'int' 42``
	
	C++11, C++14 dialect:
	
	`FunctionDecl`
	`` `-CompoundStmt``
	`` `-ReturnStmt``
	`` `-ExprWithCleanups``
	`` `-CXXConstructExpr``
	`` `-MaterializeTemporaryExpr``
	`` `-ImplicitCastExpr``
	`` `-CXXConstructExpr``
	`` `-IntegerLiteral 'int' 42``
	
	C++17, C++20 dialect:
	
	`FunctionDecl`
	`` `-CompoundStmt``
	`` `-ReturnStmt``
	`` `-ImplicitCastExpr``
	`` `-CXXConstructExpr``
	`` `-IntegerLiteral 'int' 42``
		All dialects:
		
		`FunctionDecl`
		`` `-CompoundStmt``
		`` `-ReturnStmt``
		`` `-IntegerLiteral 'int' 42``

More examples here
https://clang.llvm.org/docs/LibASTMatchersReference.html

clang-query demo

- AsIs vs IgnoreUnlessSpelledInSource
- Dump vs diag

Matching on the AST

- Three types of matchers
  - Node matchers
    - finding specific types of nodes
    - Main are `decl, stmt, type, typeloc`
  - Narrowing matchers
    - matching attributes on nodes
    - e.g. how many args, what type of args, what type of unary operator, value is equal to
  - Traversal matchers
    - For moving around the tree
    - `forEach, hasDescendant, hasAncestor, hasRHS, callee, hasTypeLoc, hasType`

Node Matchers

- Always start with a node matcher
- Can bind to node matchers
  - How most of the rewriting happens

Return type	Name	Parameters
Matcher<Attr>	attr	Matcher<Attr>...
Matcher<CXXBaseSpecifier>	cxxBaseSpecifier	Matcher<CXXBaseSpecifier>...
Matcher<CXXCtorInitializer>	cxxCtorInitializer	Matcher<CXXCtorInitializer>...
Matcher<Decl>	accessSpecDecl	Matcher<AccessSpecDecl>...
Matcher<Decl>	bindingDecl	Matcher<BindingDecl>...
Matcher<Decl>	blockDecl	Matcher<BlockDecl>...
Matcher<Decl>	classTemplateDecl	Matcher<ClassTemplateDecl>...
Matcher<Decl>	classTemplatePartialSpecializationDecl	Matcher<ClassTemplatePartialSpecializationDecl>...
Matcher<Decl>	classTemplateSpecializationDecl	Matcher<ClassTemplateSpecializationDecl>...
Matcher<Decl>	conceptDecl	Matcher<ConceptDecl>...
Matcher<Decl>	cxxConstructorDecl	Matcher<CXXConstructorDecl>...
Matcher<Decl>	cxxConversionDecl	Matcher<CXXConversionDecl>...
Matcher<Decl>	cxxDeductionGuideDecl	Matcher<CXXDeductionGuideDecl>...
Matcher<Decl>	cxxDestructorDecl	Matcher<CXXDestructorDecl>...
Matcher<Decl>	cxxMethodDecl	Matcher<CXXMethodDecl>...
Matcher<Decl>	cxxRecordDecl	Matcher<CXXRecordDecl>...
Matcher<Decl>	decl	Matcher<Decl>...
Matcher<Decl>	declaratorDecl	Matcher<DeclaratorDecl>...
Matcher<Decl>	decompositionDecl	Matcher<DecompositionDecl>...
Matcher<Decl>	enumConstantDecl	Matcher<EnumConstantDecl>...
Matcher<Decl>	enumDecl	Matcher<EnumDecl>...
Matcher<Decl>	fieldDecl	Matcher<FieldDecl>...
Matcher<Decl>	friendDecl	Matcher<FriendDecl>...

Nodes have supertypes

- `decl, expr, stmt, type, typeloc` are some of the major supertypes

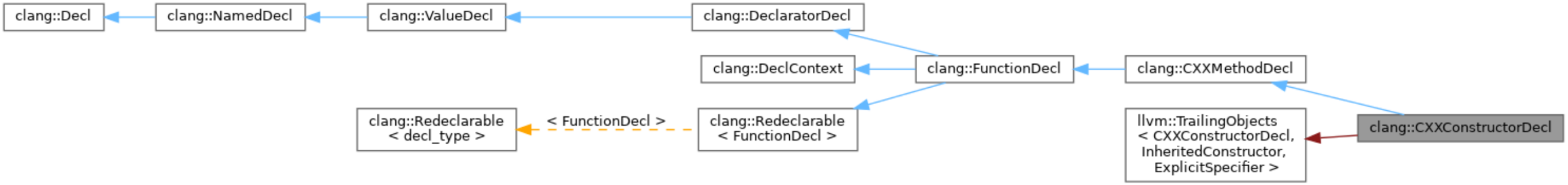
List of all members

clang::CXXConstructorDecl Class Reference final

Represents a C++ constructor within a class. More...

`#include "clang/AST/DeclCXX.h"`

Inheritance diagram for clang::CXXConstructorDecl:



Node matcher demo

```
decl()

varDecl()

binaryOperator()

functionDecl()
```

Binding Node Matchers

- Can bind to any node matcher
- In clang-query, this let's us see the subtree
- In clang-tidy, this is useful to get information from other parts of the tree, e.g. `TypeLoc`

Narrowing Matchers

- Let you narrow down your matchers
- Vary by node you are matching too
- More specific the node, more specific the matchers
- Can use as many as you want

Matcher<BinaryOperator>	hasAnyOperatorName	StringRef, ..., StringRef
Matcher<BinaryOperator>	hasOperatorName	std::string Name
Matcher<BinaryOperator>	isAssignmentOperator	
Matcher<BinaryOperator>	isComparisonOperator	
Matcher<CXXBaseSpecifier>	isPrivate	
Matcher<CXXBaseSpecifier>	isProtected	
Matcher<CXXBaseSpecifier>	isPublic	
Matcher<CXXBaseSpecifier>	isVirtual	
Matcher<CXXBoolLiteralExpr>	equals	bool Value
Matcher<CXXBoolLiteralExpr>	equals	const ValueT Value
Matcher<CXXBoolLiteralExpr>	equals	double Value
Matcher<CXXBoolLiteralExpr>	equals	unsigned Value
Matcher<CXXCatchStmt>	isCatchAll	
Matcher<CXXConstructExpr>	argumentCountAtLeast	unsigned N
Matcher<CXXConstructExpr>	argumentCountIs	unsigned N
Matcher<CXXConstructExpr>	isListInitialization	
Matcher<CXXConstructExpr>	requiresZeroInitialization	
Matcher<CXXConstructorDecl>	isCopyConstructor	
Matcher<CXXConstructorDecl>	isDefaultConstructor	
Matcher<CXXConstructorDecl>	isDelegatingConstructor	
Matcher<CXXConstructorDecl>	isExplicit	
Matcher<CXXConstructorDecl>	isInheritingConstructor	
Matcher<CXXConstructorDecl>	isMoveConstructor	
Matcher<CXXConversionDecl>	isExplicit	

Narrowing matchers demo

```
varDecl(hasName("a"))
```

```
functionDecl(hasName("foo"))
```

```
binaryOperator(hasAnyOperatorName("+"))
```

Example of a traversal matcher

```
binaryOperator(hasLHS(declRefExpr()), hasRHS(declRefExpr()))
```

```
functionDecl(hasDescendant(returnStmt(has(integerLiteral()))))
```

```
functionDecl(hasParameter(0, hasType(asString("int"))))
```

Getting Types

- Easy enough to match a simple type
- What about a templated type?
- Aliased templated type?

Templated Type Matching

- Consider `AutoPtr<int> stubInt2(nullptr)`
- `m varDecl()`
- `m varDecl(hasType(type().bind("type")))`

```
Binding for "a":
ElaboratedType 0x138fe9290 'AutoPtr<int>' sugar
`-TemplateSpecializationType 0x138fe9250 'AutoPtr<int>' sugar AutoPtr
  |-TemplateArgument type 'int'
  | `-BuiltinType 0x134050710 'int'
  `-RecordType 0x138d5e710 'CGL::AutoPtr<int>'
    `-ClassTemplateSpecialization 0x138d5e618 'AutoPtr'

/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:259:9:       "root" binds here
  259 |         AutoPtr<int>              stubInt2(nullptr);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
VarDecl 0x138fe9308 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:259:9, col:51> col:35 used stubInt2 'AutoPtr<int>':'CGL::AutoPtr<int>' callinit destroyed
`-CXXConstructExpr 0x138fe9ab8 <col:35, col:51> 'AutoPtr<int>':'CGL::AutoPtr<int>' 'void (int *)'
  `-ImplicitCastExpr 0x138fe9a70 <col:44> 'int *' <NullToPointer>
    `-CXXNullPtrLiteralExpr 0x138fe9370 <col:44> 'std::nullptr_t'
```

Two things to notice

- Elaborated type
  - Refers to a previously declared class
  - Need to match this node in order match the class
- Sugared type
  - Layers of abstractions we need to remove to get our original type to match
  - Aliases, templating, etc.

Matching the Desugared Type

- Very close:

```
Binding for "root":
ElaboratedType 0x1298f1290 'AutoPtr<int>' sugar
`-TemplateSpecializationType 0x1298f1250 'AutoPtr<int>' sugar AutoPtr
  |-TemplateArgument type 'int'
  | `-BuiltinType 0x12282f710 'int'
  `-RecordType 0x129506710 'CGL::AutoPtr<int>'
    `-ClassTemplateSpecialization 0x129506618 'AutoPtr'

Binding for "type":
RecordType 0x129506710 'CGL::AutoPtr<int>'
`-ClassTemplateSpecialization 0x129506618 'AutoPtr'

26569 matches.
clang-query> m elaboratedType(hasUnqualifiedDesugaredType(type().bind("type")))
```

Getting the Declaration

```
Binding for "root":
ElaboratedType 0x1298f1290 'AutoPtr<int>' sugar
`-TemplateSpecializationType 0x1298f1250 'AutoPtr<int>' sugar AutoPtr
  |-TemplateArgument type 'int'
  | `-BuiltinType 0x12282f710 'int'
  `-RecordType 0x129506710 'CGL::AutoPtr<int>'
    `-ClassTemplateSpecialization 0x129506618 'AutoPtr'

8606 matches.
clang-query> m elaboratedType(hasUnqualifiedDesugaredType(recordType(hasDeclaration(recordDecl().bind("a")))))
```

```
Binding for "a":
ClassTemplateSpecializationDecl 0x129506618 </Users/tristan/Code/clang-tidy-ex/CGLAutoPtrStub.h:5:1, line:78:1> line:5:31 class AutoPtr definition
|-DefinitionData empty standard_layout has_user_declared_ctor can_const_default_init
| |-DefaultConstructor exists non_trivial user_provided defaulted_is_constexpr
| |-CopyConstructor non_trivial user_declared needs_overload_resolution implicit_has_const_param
| |-MoveConstructor exists non_trivial user_declared
| |-CopyAssignment non_trivial user_declared needs_overload_resolution implicit_has_const_param
| |-MoveAssignment exists non_trivial user_declared
| `-Destructor non_trivial user_declared
|-TemplateArgument type 'int'
| `-BuiltinType 0x12282f710 'int'
|-CXXRecordDecl 0x1296b0fe0 <col:25, col:31> col:31 implicit class AutoPtr
|-AccessSpecDecl 0x1296b1090 <line:7:1, col:7> col:1 public
|-CXXConstructorDecl 0x1296b1238 <line:8:5, line:10:5> line:8:14 used AutoPtr 'void (int *)' implicit-inline
| |-ParmVarDecl 0x1296b1130 <col:22, col:39> col:28 pi_pType 'int *' cinit
| | `-ImplicitCastExpr 0x129735358 <col:39> 'int *' <NullToPointer>
| |   `-IntegerLiteral 0x128348ce0 <col:39> 'int' 0
```

All Together

```
Match #44:

/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:259:9:       "root" binds here
  259 |         AutoPtr<int>              stubInt2(nullptr);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
VarDecl 0x1298f1308 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:259:9, col:51> col:35 used stubInt2 'AutoPtr<int>':'CGL::AutoPtr<int>' callinit destroyed
`-CXXConstructExpr 0x1298f1ab8 <col:35, col:51> 'AutoPtr<int>':'CGL::AutoPtr<int>' 'void (int *)'
  `-ImplicitCastExpr 0x1298f1a70 <col:44> 'int *' <NullToPointer>
    `-CXXNullPtrLiteralExpr 0x1298f1370 <col:44> 'std::nullptr_t'

44 matches.
clang-query> m varDecl(hasType(elaboratedType(hasUnqualifiedDesugaredType(recordType(hasDeclaration(recordDecl(hasName("AutoPtr")))))))) 
```

Typedef

```
m varDecl(hasType(asString("TestDefInt")))
```

```
/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9:       "root" binds here
  153 |         TestDefInt iiiii(i.IsNull() ? i : ii);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
VarDecl 0x1398cafd0 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9, col:45> col:20 iiiii 'TestDefInt':'CGL::AutoPtr<int>' callinit destroyed
`-CXXConstructExpr 0x1398cb7a0 <col:20, col:45> 'TestDefInt':'CGL::AutoPtr<int>' 'void (AutoPtr<int> &)'
  `-ConditionalOperator 0x1398cb100 <col:26, col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue
    |-CXXMemberCallExpr 0x1398cb088 <col:26, col:35> 'bool'
    | `-MemberExpr 0x1398cb058 <col:26, col:28> '<bound member function type>' .IsNull 0x13824a860
    |   `-ImplicitCastExpr 0x1398cb0a8 <col:26> 'const CGL::AutoPtr<int>' lvalue <NoOp>
    |     `-DeclRefExpr 0x1398cb038 <col:26> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1398c8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    |-DeclRefExpr 0x1398cb0c0 <col:39> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1398c8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    `-DeclRefExpr 0x1398cb0e0 <col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1398c9560 'ii' 'TestDefInt':'CGL::AutoPtr<int>'

5 matches.
clang-query> m varDecl(hasType(asString("TestDefInt")))
```

What do we get with the elaborated type?

```
m varDecl(hasType(asString("TestDefInt")), hasType(elaboratedType().bind("type")))
```

```
Match #5:

/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9:       "root" binds here
  153 |         TestDefInt iiiii(i.IsNull() ? i : ii);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
VarDecl 0x1398cafd0 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9, col:45> col:20 iiiii 'TestDefInt':'CGL::AutoPtr<int>' callinit destroyed
`-CXXConstructExpr 0x1398cb7a0 <col:20, col:45> 'TestDefInt':'CGL::AutoPtr<int>' 'void (AutoPtr<int> &)'
  `-ConditionalOperator 0x1398cb100 <col:26, col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue
    |-CXXMemberCallExpr 0x1398cb088 <col:26, col:35> 'bool'
    | `-MemberExpr 0x1398cb058 <col:26, col:28> '<bound member function type>' .IsNull 0x13824a860
    |   `-ImplicitCastExpr 0x1398cb0a8 <col:26> 'const CGL::AutoPtr<int>' lvalue <NoOp>
    |     `-DeclRefExpr 0x1398cb038 <col:26> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1398c8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    |-DeclRefExpr 0x1398cb0c0 <col:39> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1398c8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    `-DeclRefExpr 0x1398cb0e0 <col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1398c9560 'ii' 'TestDefInt':'CGL::AutoPtr<int>'

Binding for "type":
ElaboratedType 0x1398c8d00 'TestDefInt' sugar
`-TypedefType 0x1398c8cd0 '(anonymous namespace)::TestDefInt' sugar
  |-Typedef 0x1396f6808 'TestDefInt'
  `-ElaboratedType 0x1396f6770 'CGL::AutoPtr<int>' sugar
    `-TemplateSpecializationType 0x1396f6730 'AutoPtr<int>' sugar AutoPtr
      |-TemplateArgument type 'int'
      | `-BuiltinType 0x13285d910 'int'
      `-RecordType 0x1396f6710 'CGL::AutoPtr<int>'
        `-ClassTemplateSpecialization 0x1396f6618 'AutoPtr'

5 matches.
clang-query> m varDecl(hasType(asString("TestDefInt")), hasType(elaboratedType().bind("type")))
```

Elaborated Desugared Type?

```
m varDecl(hasType(asString("TestDefInt")),
hasType(elaboratedType(hasUnqualifiedDesugaredType(type().bind("type")))))
```

```
Match #5:

/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9:       "root" binds here
  153 |         TestDefInt iiiii(i.IsNull() ? i : ii);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
VarDecl 0x1410bafd0 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9, col:45> col:20 iiiii 'TestDefInt':'CGL::AutoPtr<int>' callinit destroyed
`-CXXConstructExpr 0x1410bb7a0 <col:20, col:45> 'TestDefInt':'CGL::AutoPtr<int>' 'void (AutoPtr<int> &)'
  `-ConditionalOperator 0x1410bb100 <col:26, col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue
    |-CXXMemberCallExpr 0x1410bb088 <col:26, col:35> 'bool'
    | `-MemberExpr 0x1410bb058 <col:26, col:28> '<bound member function type>' .IsNull 0x14030a860
    |   `-ImplicitCastExpr 0x1410bb0a8 <col:26> 'const CGL::AutoPtr<int>' lvalue <NoOp>
    |     `-DeclRefExpr 0x1410bb038 <col:26> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1410b8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    |-DeclRefExpr 0x1410bb0c0 <col:39> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1410b8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    `-DeclRefExpr 0x1410bb0e0 <col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1410b9560 'ii' 'TestDefInt':'CGL::AutoPtr<int>'

Binding for "type":
RecordType 0x140ee6710 'CGL::AutoPtr<int>'
`-ClassTemplateSpecialization 0x140ee6618 'AutoPtr'

5 matches.
clang-query> m varDecl(hasType(asString("TestDefInt")), hasType(elaboratedType(hasUnqualifiedDesugaredType(type().bind("type")))))
```

Same as before, we can grab make sure it's AutoPtr the same way.

```
m varDecl(hasType(asString("TestDefInt")),
hasType(elaboratedType(hasUnqualifiedDesugaredType(recordType(hasDeclaration(recordD
ecl(hasName("AutoPtr")))))))))
```

```
/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9:       "root" binds here
  153 |         TestDefInt iiiii(i.IsNull() ? i : ii);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
VarDecl 0x1410bafd0 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsAutoPtr.cpp:153:9, col:45> col:20 iiiii 'TestDefInt':'CGL::AutoPtr<int>' callinit destroyed
`-CXXConstructExpr 0x1410bb7a0 <col:20, col:45> 'TestDefInt':'CGL::AutoPtr<int>' 'void (AutoPtr<int> &)'
  `-ConditionalOperator 0x1410bb100 <col:26, col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue
    |-CXXMemberCallExpr 0x1410bb088 <col:26, col:35> 'bool'
    | `-MemberExpr 0x1410bb058 <col:26, col:28> '<bound member function type>' .IsNull 0x14030a860
    |   `-ImplicitCastExpr 0x1410bb0a8 <col:26> 'const CGL::AutoPtr<int>' lvalue <NoOp>
    |     `-DeclRefExpr 0x1410bb038 <col:26> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1410b8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    |-DeclRefExpr 0x1410bb0c0 <col:39> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1410b8d48 'i' 'TestDefInt':'CGL::AutoPtr<int>'
    `-DeclRefExpr 0x1410bb0e0 <col:43> 'TestDefInt':'CGL::AutoPtr<int>' lvalue Var 0x1410b9560 'ii' 'TestDefInt':'CGL::AutoPtr<int>'

5 matches.
clang-query> m varDecl(hasType(asString("TestDefInt")), hasType(elaboratedType(hasUnqualifiedDesugaredType(recordType(hasDeclaration(recordDecl(hasName("AutoPtr")))))))))
```

Rewriting with clang-tidy

- In early demos green squiggles underneath, bound nodes represent what we have “bound” to
- We use the `changeTo` stencil to change whatever the root node of our matcher is
- We can get information from other nodes by binding to them then “printing” the output of the node

Example rule

```
1   std::vector<RewriteRuleWith<std::string>> createRulesForMethodsWithNoArg()
2   {
3       const std::vector<std::pair<const char*, const char*>> oldAndNewMethods = {
4           {"::CGLFile::Path::FullPath", "u16string"}
5       };
6
7       std::vector<RewriteRuleWith<std::string>> result;
8       for (const auto& [oldMethod, newMethod] : oldAndNewMethods) {
9           const std::string object = "object";
10          result.emplace_back(
11              makeRule(traverse(clang::TK_IgnoreUnlessSpelledInSource, cxxMemberCallExpr(on(expr().bind(object)),
12                                                                       callee(cxxMethodDecl(hasName(oldMethod))))),
13                       changeTo(cat(access(object, cat(newMethod, "()")))),
14                       createDeprecatedMessage(oldMethod, newMethod)));
15      }
16      return result;
17  }
```

Replacing node where we do the method call

Capturing the object we call the method on

Matching the method name we call

Example matching from the rule

This is the object we're accessing that we bound

```
Match #3:

/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsPath.cpp:50:5:       "obj" binds here
   50 |     pathRawPtr->FullPath();
      |     ^~~~~~~~~~
Binding for "obj":
DeclRefExpr 0x1317cd708 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsPath.cpp:50:5> 'const Path *' lvalue Var 0x131779358 'pathRawPtr' 'const Path *'

/Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsPath.cpp:50:5:       "root" binds here
   50 |     pathRawPtr->FullPath();
      |     ^~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
CXXMemberCallExpr 0x1317cd770 </Users/tristan/Code/clang-tidy-ex/TestCallsReplacementsPath.cpp:50:5, col:26> 'const std::u16string':'const std::u16string'
|-MemberExpr 0x1317cd740 <col:5, col:17> '<bound member function type>' ->FullPath 0x1316d58b0
| `-ImplicitCastExpr 0x1317cd728 <col:5> 'const Path *' <LValueToRValue>
|   `-DeclRefExpr 0x1317cd708 <col:5> 'const Path *' lvalue Var 0x131779358 'pathRawPtr' 'const Path *'
`-CXXDefaultArgExpr 0x1317cd798 <<invalid sloc>> 'bool'

3 matches.
clang-query> m cxxMemberCallExpr(on(expr().bind("obj")), callee(cxxMethodDecl(hasName("::CGLFile::Path::FullPath"))))
```

This is the root we're replacing

Rewriting Rule Example

```
changeTo(cat(access(object, cat(newMethod, "()"))))
```

- Accesses bound object
- Writes text from variable
- Assume object is bound to an object called myObject

In our case
“u16string”

If pointer:

```
myObject->u16string();
```

Else:

```
myObject.u16string();
```

Stencils

- How we print information and get information from nodes
- Simplest: `cat("text")`
  - Concatenating print function
- `node(`*`node`*`)`
  - Gets a whole node and semicolon if it has it
  - `access(`*`node`*`)`
    - Prints the name of the node and how to access it
- `name(`*`node`*`)`
  - Gets the node name, if the node has a name. Works for nodes like `NamedDecl`, `TypeLoc`
- `enclose(`*`range1, range2`*`)`
  - Lets you copy text from ranges which you can get by doing `before(`*`node`*`)` or `after(`*`node`*`)`
- See all here https://clang.llvm.org/doxygen/RangeSelector_8h.html

Testing Framework

- Wanted to make sure we weren’t regressing when building new rules (conflicts, overriding, etc.)
- Solution: test file, expected output file
- Run clang-tidy on test file, diff expected, restore original test file
  - Written into a script

Takeaways from building matchers

- Use rules with very specific matchings, even if they have the same output
  - Let's you disable/re-enable rules in case you start getting build errors
  - Makes sure your rules don’t overlap as it will only apply one

```
auto createReplaceSmartPtrBySharedRule()
{
    return applyFirst({rewriteImplicitConversionSPPointerReset(),
                       rewriteImplicitConversionSPPointer(),
                       rewriteImplicitConversion(),
                       rewriteSwapToLowerCase(),
                       rewriteOperatorsToGet(),
                       ::CoveoClangTidyPluginsUtils::ReplaceMethodCallWithNoParamBuilder()
                           .withOldMethodName("GetPtr")
                           .withNewMethodName("get")
                           .withParentClass(className)
                           .build(),
                       replaceStarEqNullPtrWithReset(),
                       rewriteNewToCGLMakeSmart(),
                       replaceEqNullPtrWithReset(),
                       rewriteRawAssignmentToReset(),
                       rewriteImplicitConversionDependent()});
}
```

Takeaways from building matchers

- Have very specific matchings that no longer apply once the rule is run (matchers are idempotent)
  - In case you have overlapping rules, you can just run all the rules twice and it will handle both cases
  - E.g. in our case creating a pointer inside a pointer

Takeaways from building matchers

- Create a check that displays warnings when the user does regressive behavior
  - Can use existing matchers with no-op and add to standard IDE config
  - Important if you’re doing removal in stages

```
auto createFindRawPointerAssigmentOperatorCallExprRule()
{
    const std::string varExpr = "vexpr";
    return makeRule(
        traverse(
            clang::TK_AsIs,
            cxxOperatorCallExpr(
                isAssignmentOperator(),
                hasLHS(expr(matchAliasType(className))),
                hasRHS(expr(unless(cxxNullPtrLiteralExpr()), unless(integerLiteral()), unless(hasDescendant(cxxNullPtrLiteralExpr()))),
                .bind(varExpr)),
        noopEdit(node(varExpr)),
        cat(msgCheck));
}
```

Using an iterative process to transition from current to target class

- **Problem**: How can we migrate to a new API that is very similar while minimizing the risk we break anything?

- **Answer**: Do it in small steps.

1. Clean up use cases of current object

- Will probably be some places where current API is doing something not allowed in new API
  - E.g `shared_ptr` of `this`
- Needs to be fixed manually
- First step – is this even possible with new API?

2. Add target API to current API

- Should make it so all methods called on objects match methods called on target API
- Add in methods if they don’t exist, build out of existing methods
- E.g. `Add(i)` to `add(i)`

3. Build clang-tidy rules to migrate API

- Rewrite method names
- Rewrite constructors
- Redo operators
- Remove implicit calls
- Use utilities like `std::make_xxx`
- etc.

4. Alias API

- Alias current API to target API
- When you build it will point out all the places where transition wasn’t completed
  - Might need to revert and go build new rules

```
5   namespace CGL {
6
7   template<class Type> using ArrayAutoPtr = std::unique_ptr<Type[]>;
8
9   }
```

5. Remove aliasing

- Go and rewrite all places where current class name is used to target class name
- Deal with typedefs as well

Questions? and Links

- Our large-scale refactoring rules: https://github.com/coveooss/clang-tidy-plugin-examples
- AST Matcher reference https://clang.llvm.org/docs/LibASTMatchersReference.html
- Clang transformer tutorial https://intel.github.io/llvm-docs/clang/ClangTransformerTutorial.html
- Clang classes https://clang.llvm.org/doxygen/index.html (probably better to just google the nodes you're interested in, but you'll end up here)