Extending clang-tidy in the Present and the Future

Tools, Tips, Tricks and Traps

ACCU 2019
Stephen Kelly
steveire.wordpress.com
@steveire

Refactor with Clang Tooling

Tools, Tips, Tricks and Traps

ACCU 2019
Stephen Kelly
steveire.wordpress.com
@steveire

Stephen Kelly

- @steveire
- steveire.wordpress.com
- KDE
- Qt
- CMake
- Clang

Agenda

- What does clang-tidy do?
- Why refactor mechanically?
- How can we extend clang-tidy?
- What tools can help us?
- What problems will we encounter?

Non-Agenda

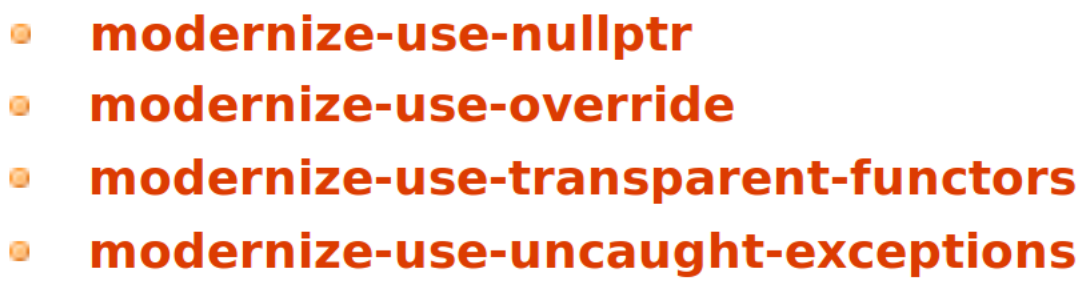
- Existing details of clang-tidy
- Compilation Databases

Take-aways

- Large refactorings possible
  - Bespoke needs
  - In your code
- Improving in near future
  - Better tooling
  - Better collaboration

Tools

clang-tidy Prior Art



- modernize-use-nullptr
- modernize-use-override
- modernize-use-transparent-functors
- modernize-use-uncaught-exceptions

- 241 existing checks
- https://clang.llvm.org/extra/clang-tidy
- Some library-specific
  - clang-tidy has no plugin system

clang-tidy - modernize-use-override

```
struct Base
{
    virtual void foo();
};

struct Derived : Base
{
    virtual void foo();
};
```

Demo

- 0001-clang-tidy-demo
- https://godbolt.org/z/NRo5Zi

Replace OldType with NewType

```
void foo()
{
    OldType someVar;
}
```

Replace OldType with NewType

```
void foo()
{
    OldType someVar = calledFunction();
}
```

Replace OldType with NewType

```
OldType foo()
{
    OldType someVar;
    return someVar;
}
```

Replace OldType with NewType

```
OldType foo()
{
    OtherType someVar;
    return someVar;
}
```

Replace OldType with NewType

```
OtherType foo()
{
    OldType someVar;
    return someVar;
}
```

Replace OldType with NewType

```
struct Bar {
    OldType mVar;
    void foo();
};
...
void Bar::foo()
{
    mVar = someFunction();
}
```

Motivation

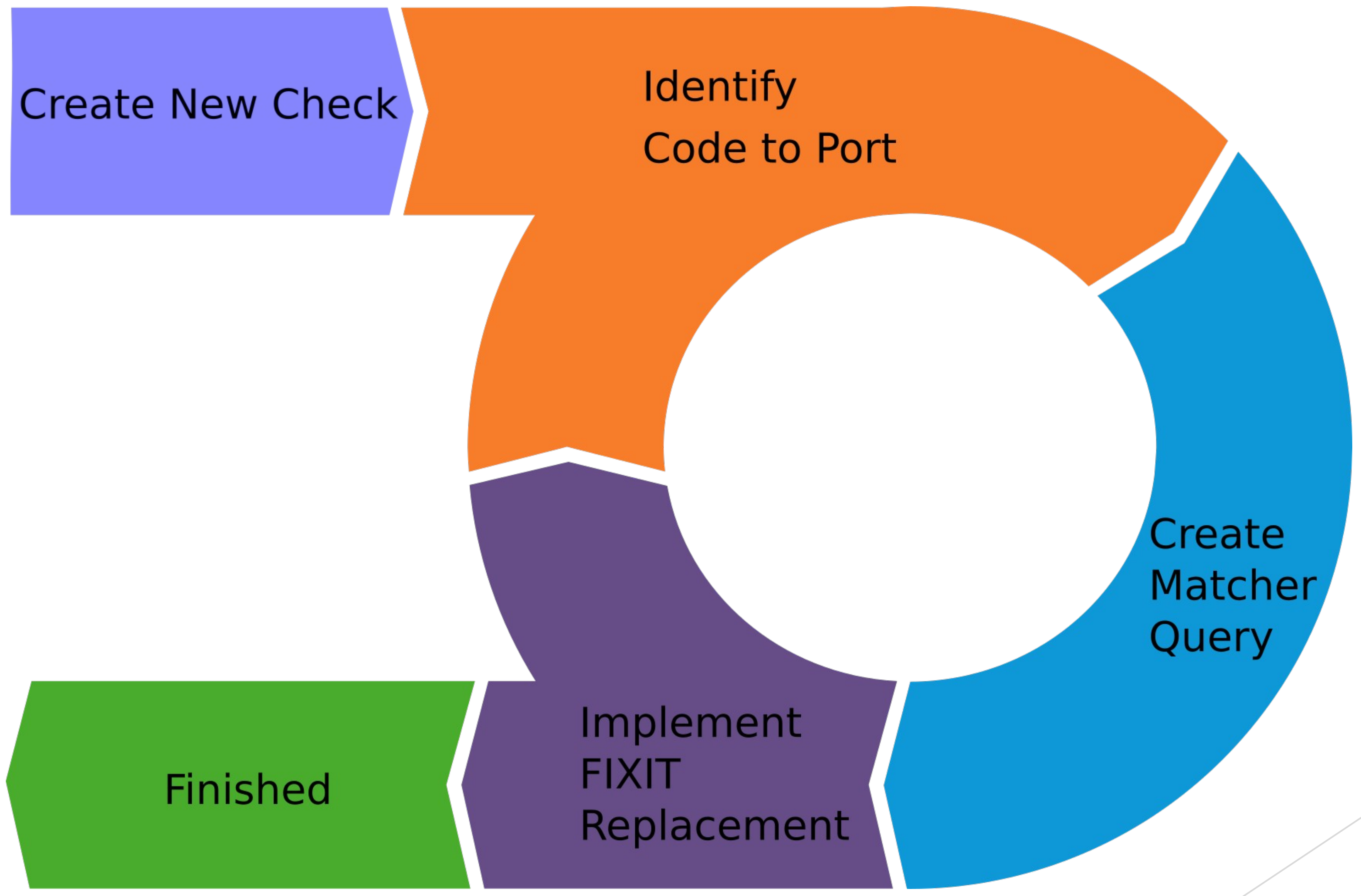
- Rename A::get() to A::makeSomeType()

```
struct A {
  SomeType get() {
    return m_someTypeFactory.get();
  }
  Factory m_someTypeFactory;
};
```

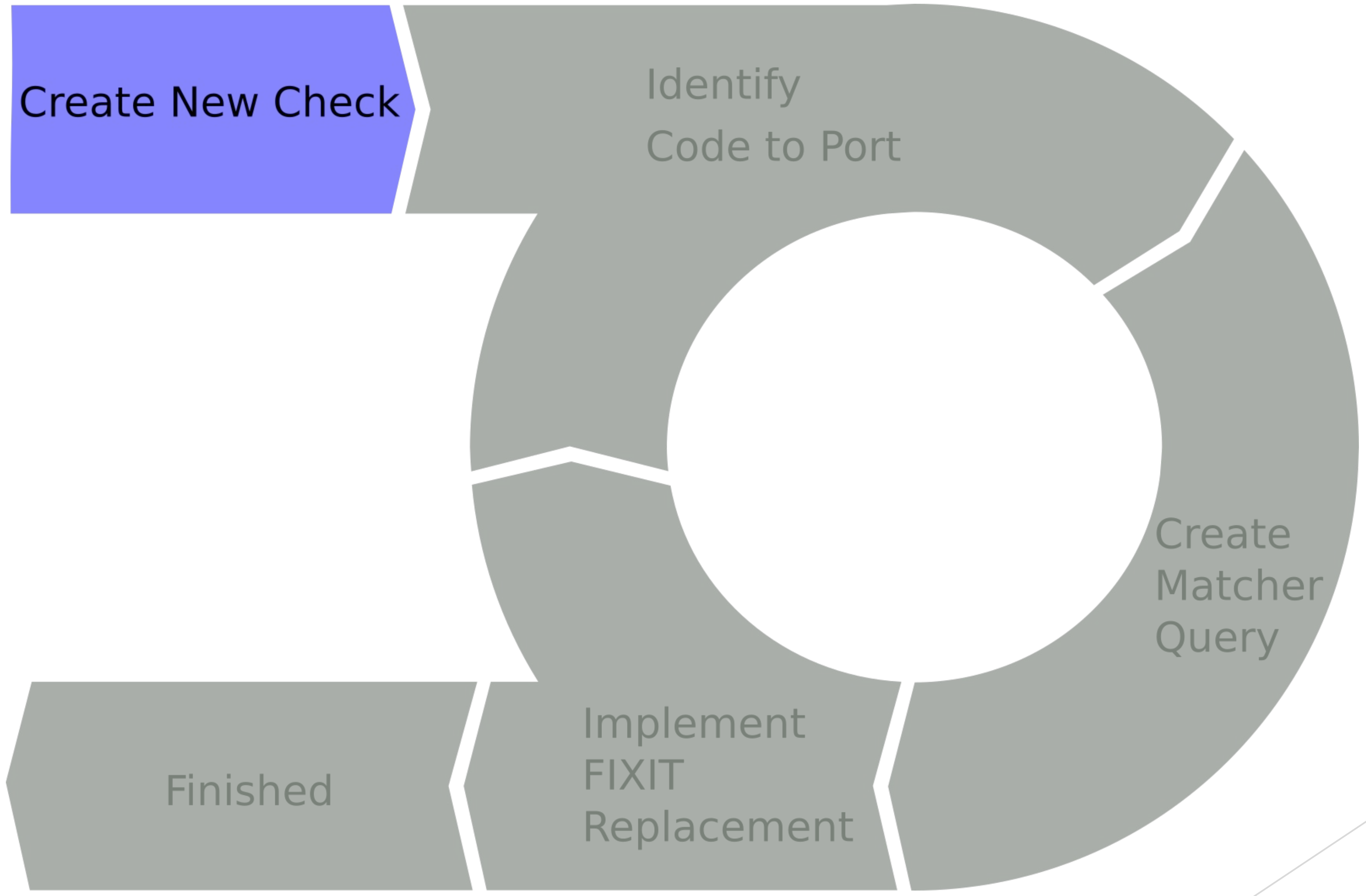
Motivation

- Large scale refactoring
- Not practical to port using sed
- Semantic knowledge of C++ code
- Automation
- Repeatability
- Don’t break code

Extending clang-tidy



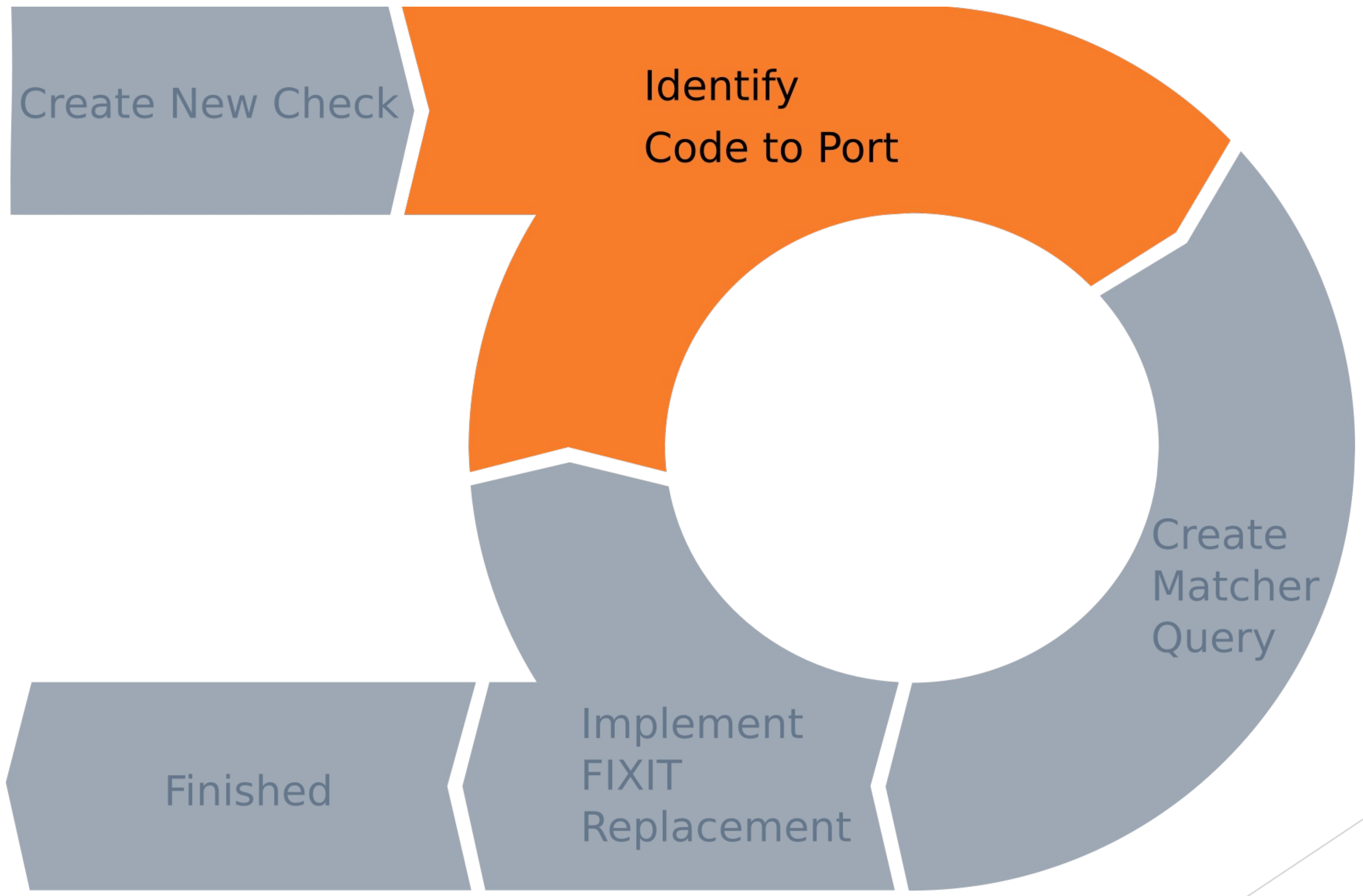
Extending clang-tidy



Demo

- 0002-clang-tidy-new-check

Extending clang-tidy



Extending clang-tidy

- Match
  - Variables
  - Functions
  - Function calls
  - Classes
  - Instances
  - etc

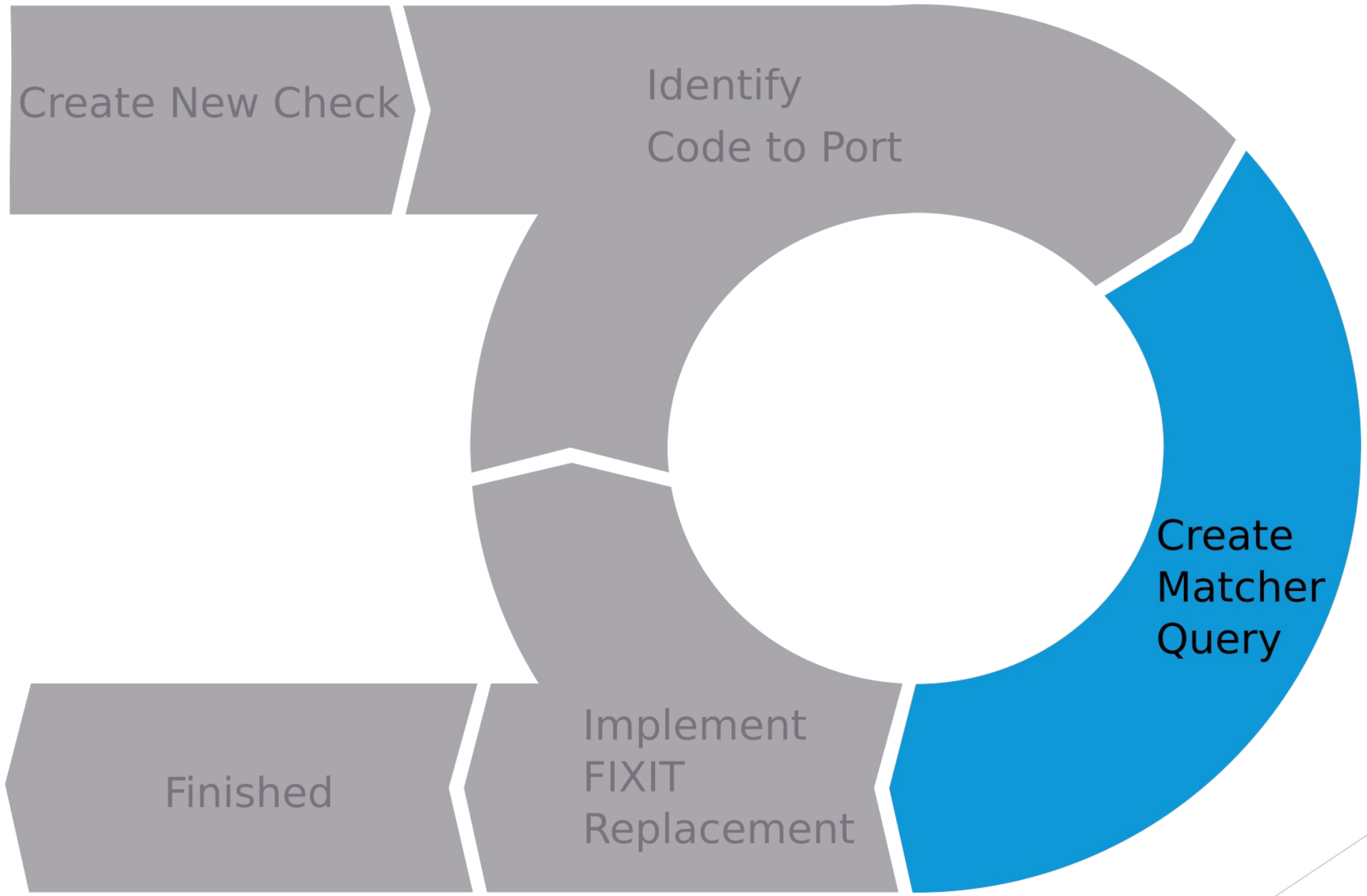
Extending clang-tidy

- Match **particular** entities!
  - Variables
  - Functions
  - Function calls
  - Classes
  - Instances
  - etc

Extending clang-tidy

- Match **particular** entities!
  - By name
  - By (return?) type
  - By parameter type/name
  - By content
  - etc

Extending clang-tidy



AST Matchers

- Predicate language for matching on AST nodes
- Content of a matcher call refines the call

AST Matchers

- Predicate language for matching on AST nodes
- Content of a matcher call refines the call
- `cxxMethodDecl(isOverride())`
- “match method declaration which is an override”

AST Matchers

- Predicate language for matching on AST nodes
- Content of a matcher call refines the call
- `cxxMethodDecl(isOverride())`
- “match method declaration which is an override”
- Dozens of interesting matchers available in Clang
- Match on declarations, expressions, statements, types
- http://clang.llvm.org/docs/LibASTMatchersReference.html

AST Matchers

- Predicate language for matching on AST nodes
- Content of a matcher call refines the call
- `cxxMethodDecl(isOverride())`
- “match method declaration which is an override”
- Dozens of interesting matchers available in Clang
- Match on declarations, expressions, statements, types
- http://clang.llvm.org/docs/LibASTMatchersReference.html
- Extensible with custom matchers

AST Matchers

- `functionDecl()`
- `functionDecl(isInline())`
- `functionDecl(hasName("foo"))`
-
```
functionDecl(
    hasParameter(0, hasName("foo"))
    )
```

Discovery

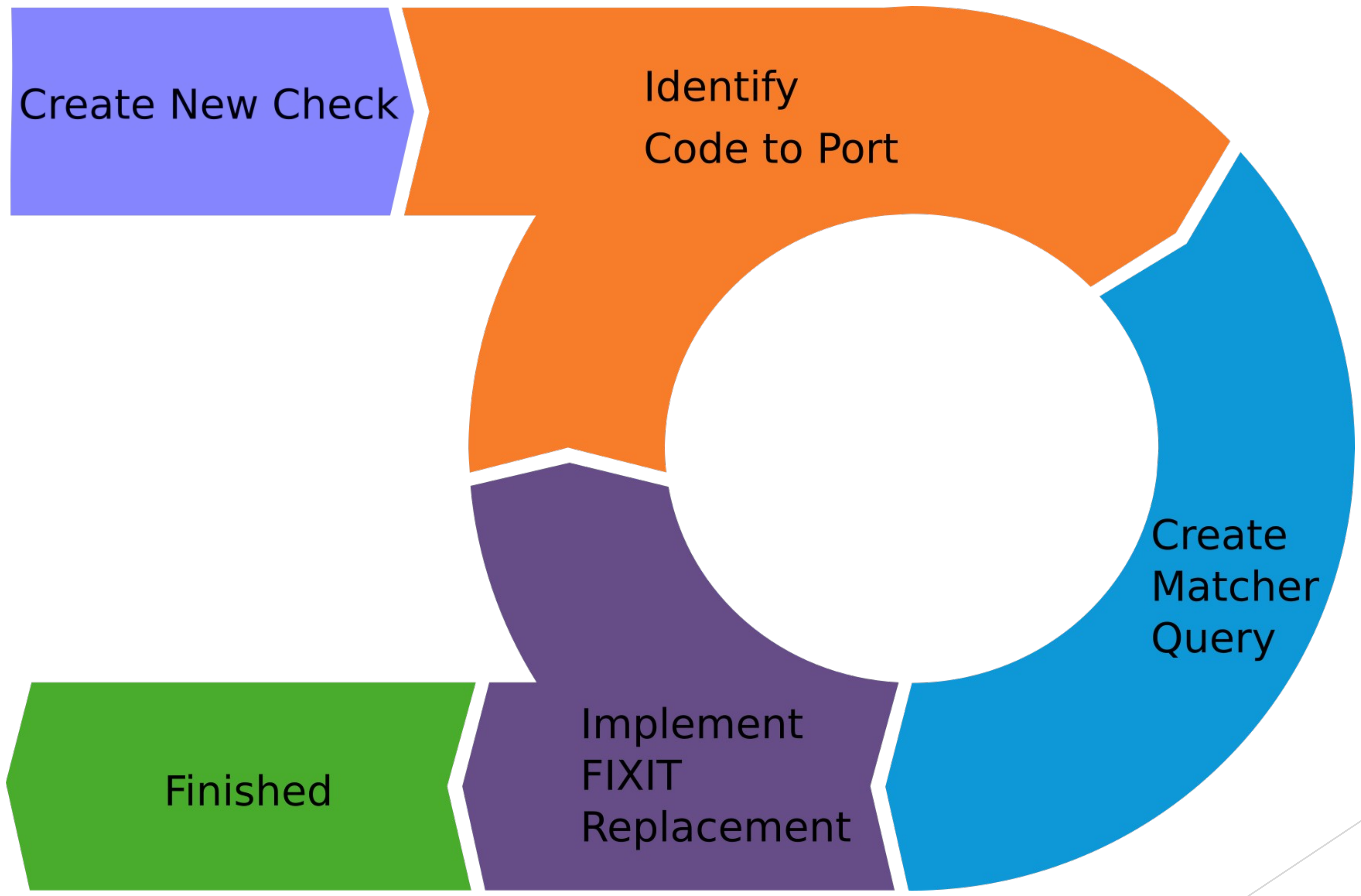
Demo

- 0003-clang-query-demo
- http://ce.steveire.com/z/tsl08L

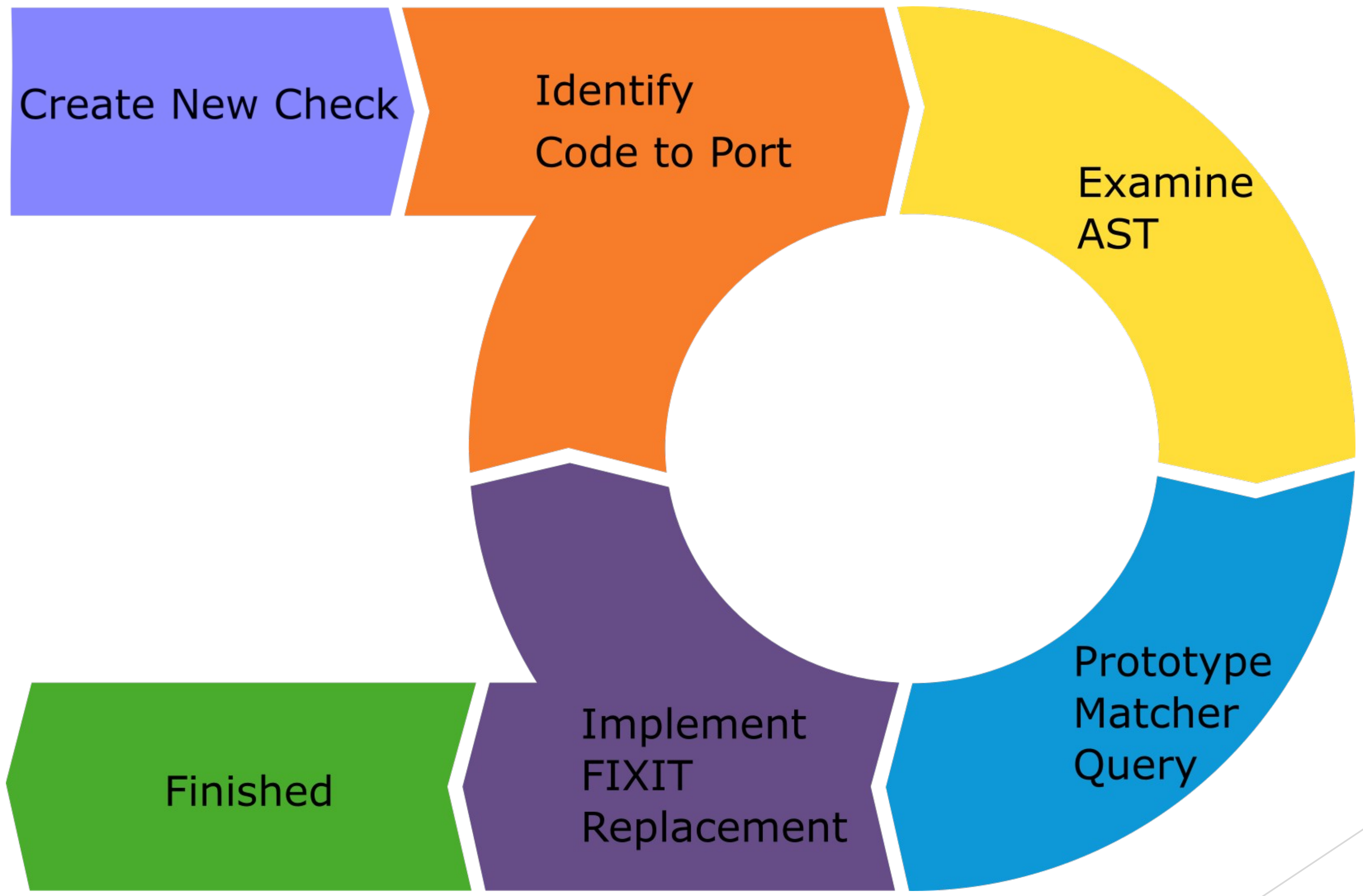
clang-query

- Inspection of code
- Intelligent Code Completion
- Show available matchers
  - And where they match
- Show AST

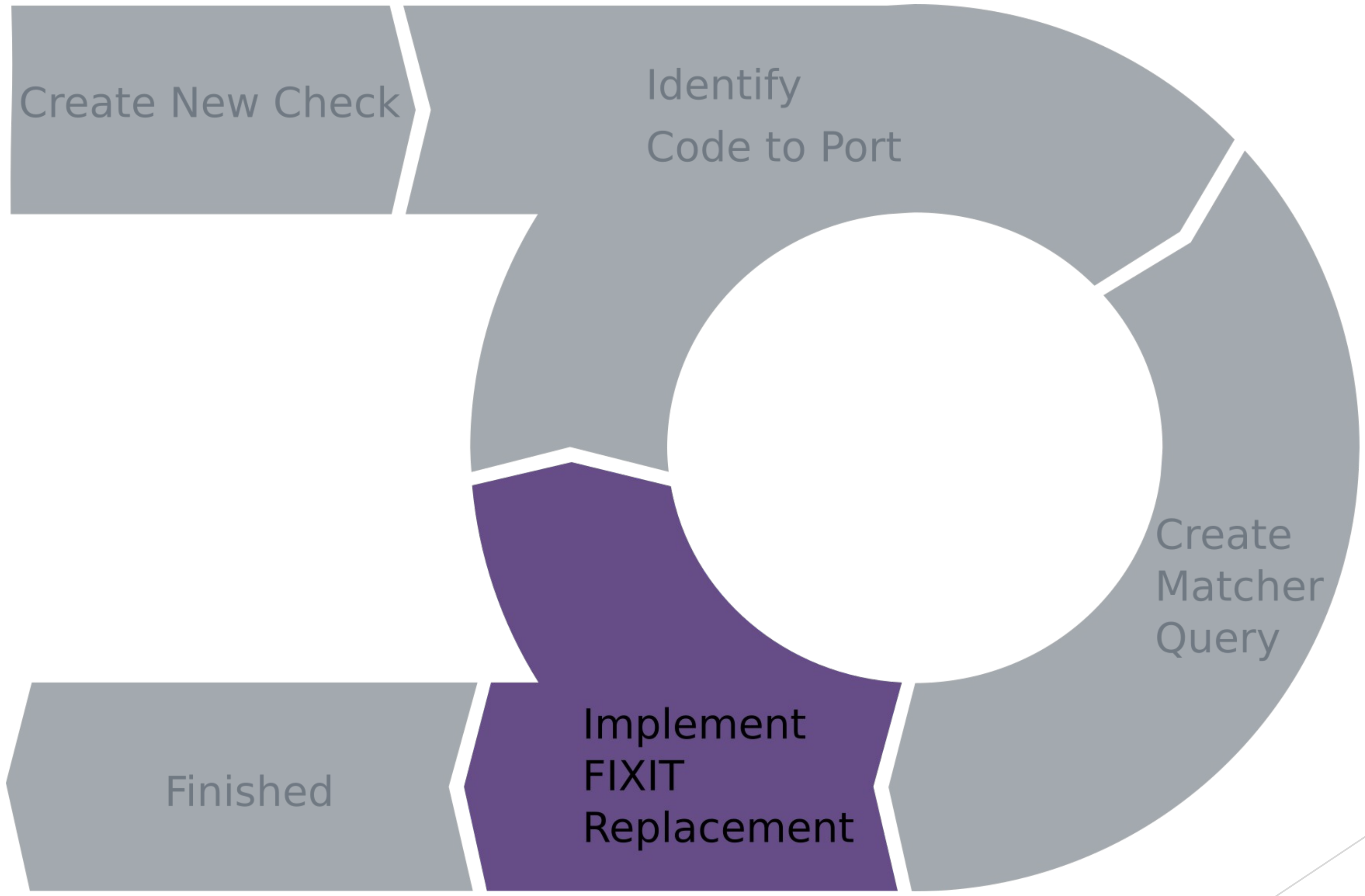
Future clang-tidy workflow



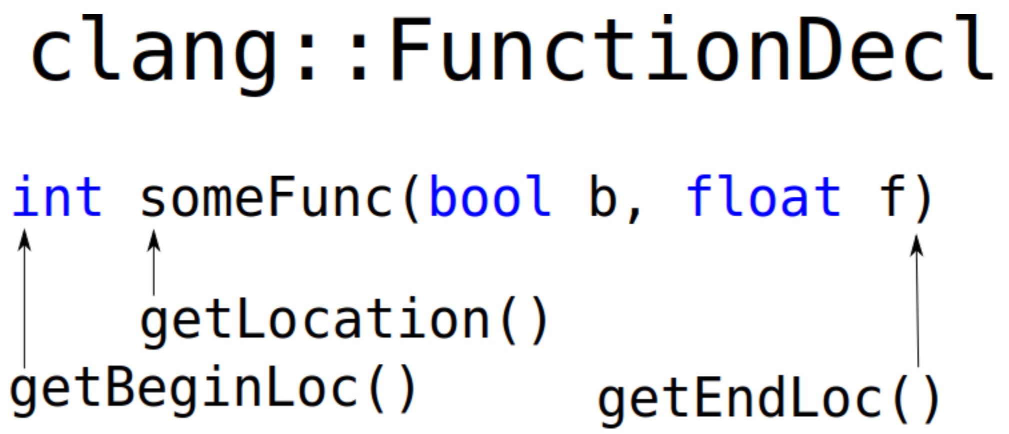
Current clang-tidy workflow



Extending clang-tidy

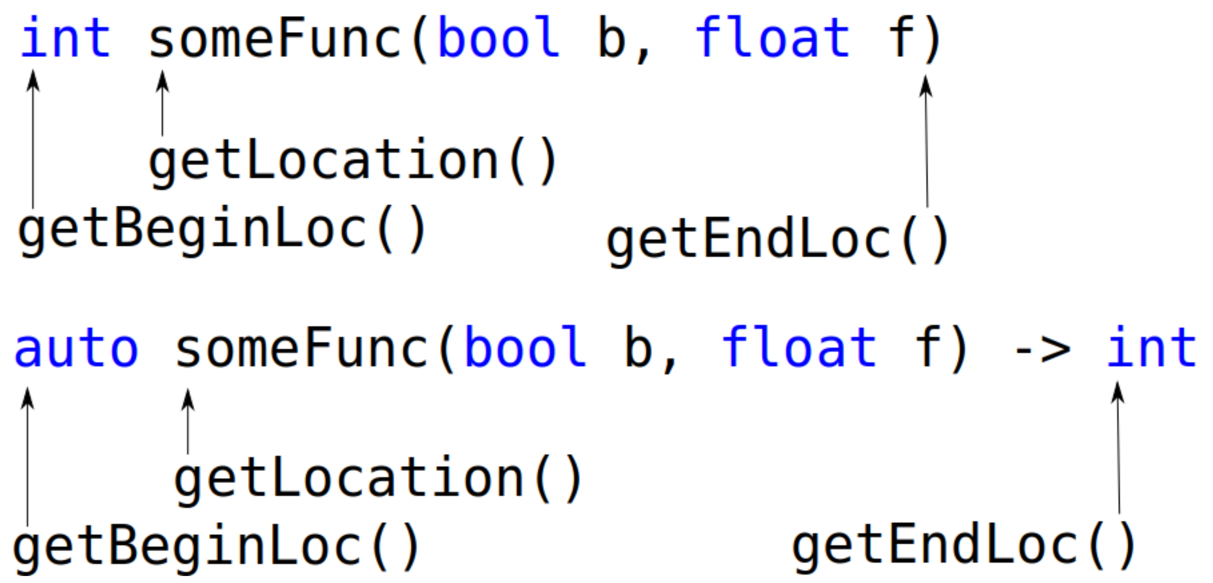


Source Locations



Source Locations

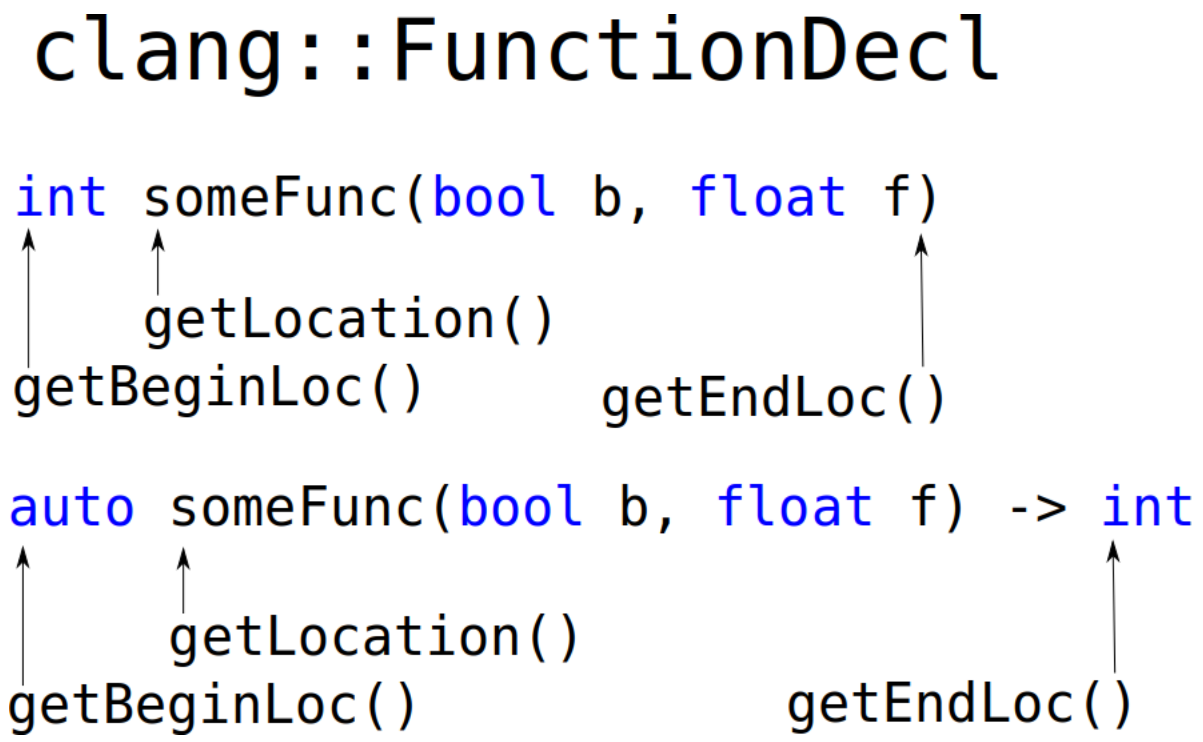
clang::FunctionDecl



```
int someFunc(bool b, float f)
```

```
auto someFunc(bool b, float f) -> int
```

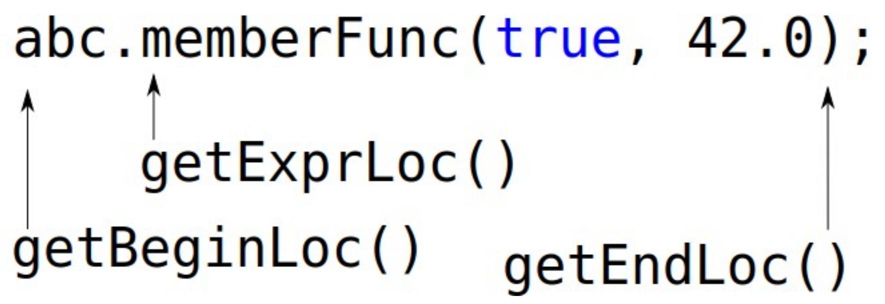
Source Locations (return type)



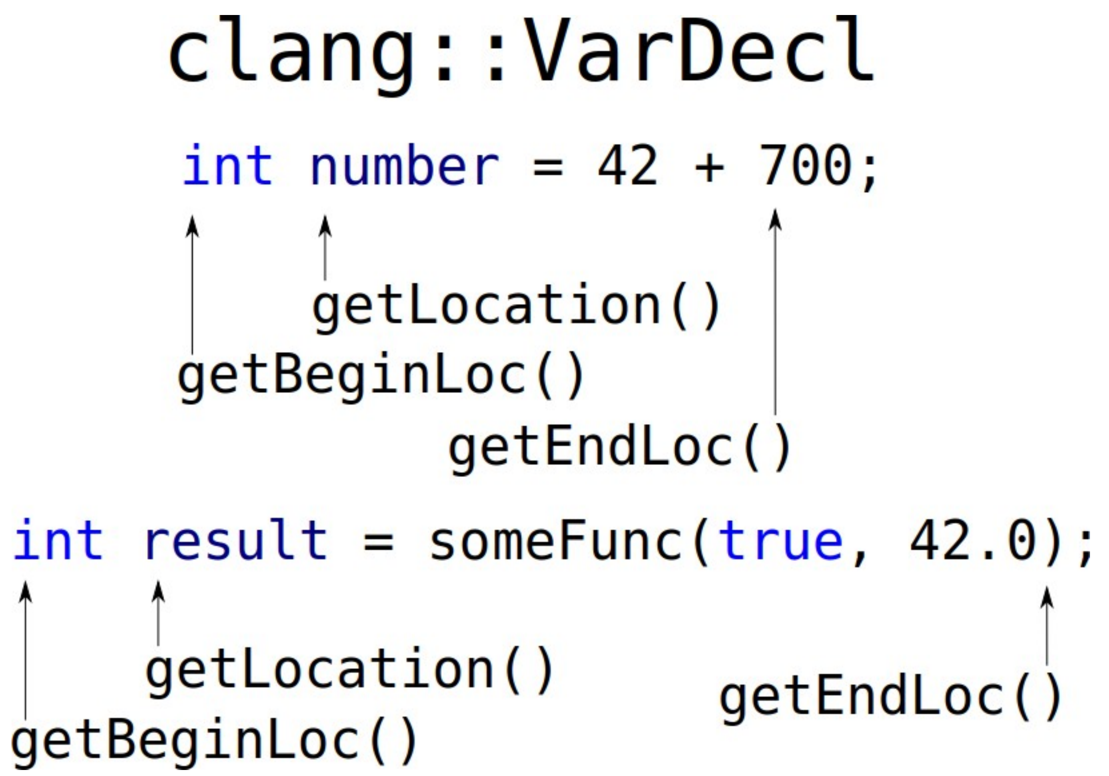
- Return type location:
  - `getTypeSourceInfo()->getTypeLoc().getAs<clang::FunctionTypeLoc>().getReturnLoc()`

Source Locations

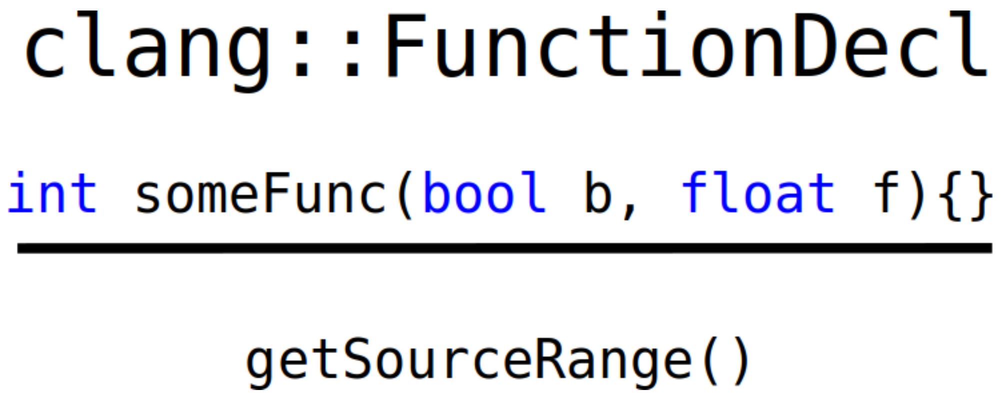
clang::CXXMemberCallExpr



Source Locations



Source Ranges



clang::FunctionDecl

```cpp
int someFunc(bool b, float f){}
```

getSourceRange()

Source Ranges

```
clang::FunctionDecl
```

```
int someFunc(bool b, float f){}
```

```
getTypeSourceInfo()->getTypeLoc().getSourceRange()
```

Source Ranges

`clang::FunctionDecl`

```
int someFunc(bool b, float f)
```

```
getTypeSourceInfo()->getTypeLoc().getAs<clang::FunctionTypeLoc>().getParensRange()
```

Source Locations

Demo

- 0004-clang-query-locations
- http://ce.steveire.com/z/2QnXCB

Demo

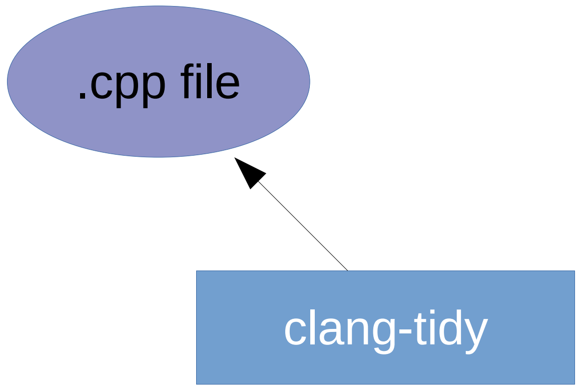
- 0004-clang-query-debugging
- http://ce.steveire.com/z/cAruoF

clang-tidy at scale

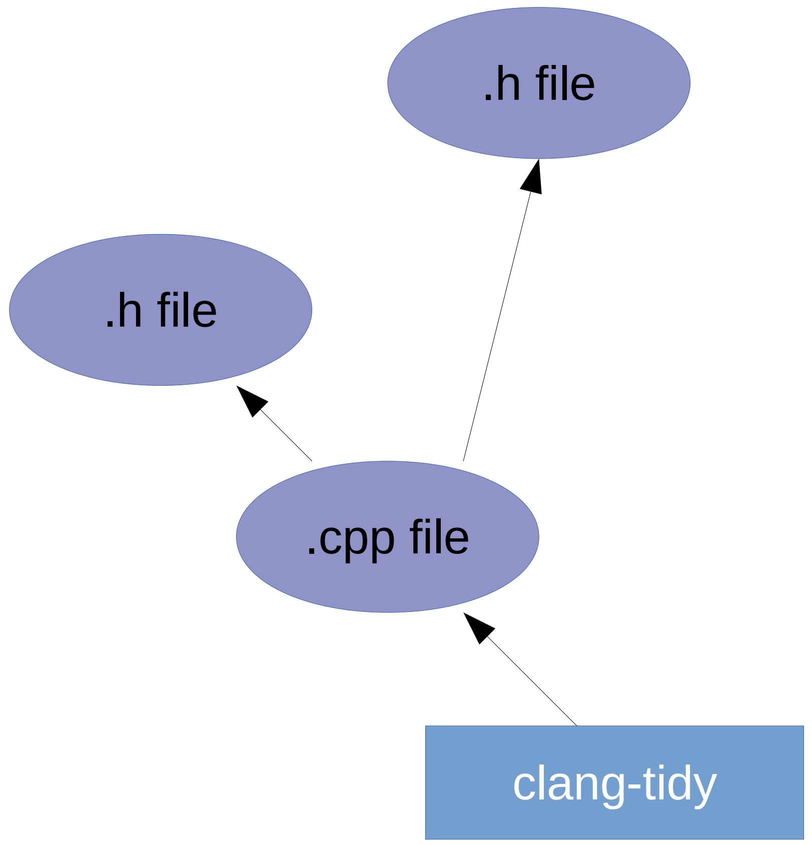
Compile Options

- Prototype with `--`
  - `clang-query m.cpp -- -I /usr/include/qt5/ -fPIC`
- Generate Compilation Database
  - CMake
  - Ninja
  - Custom

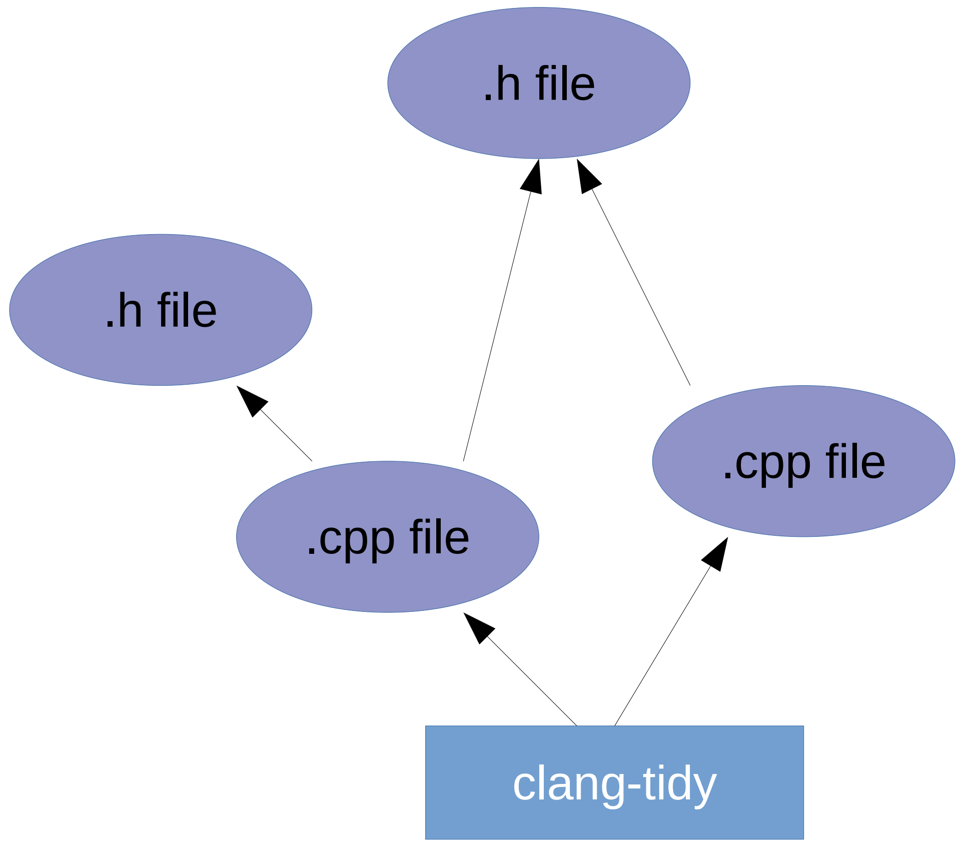
Replacements



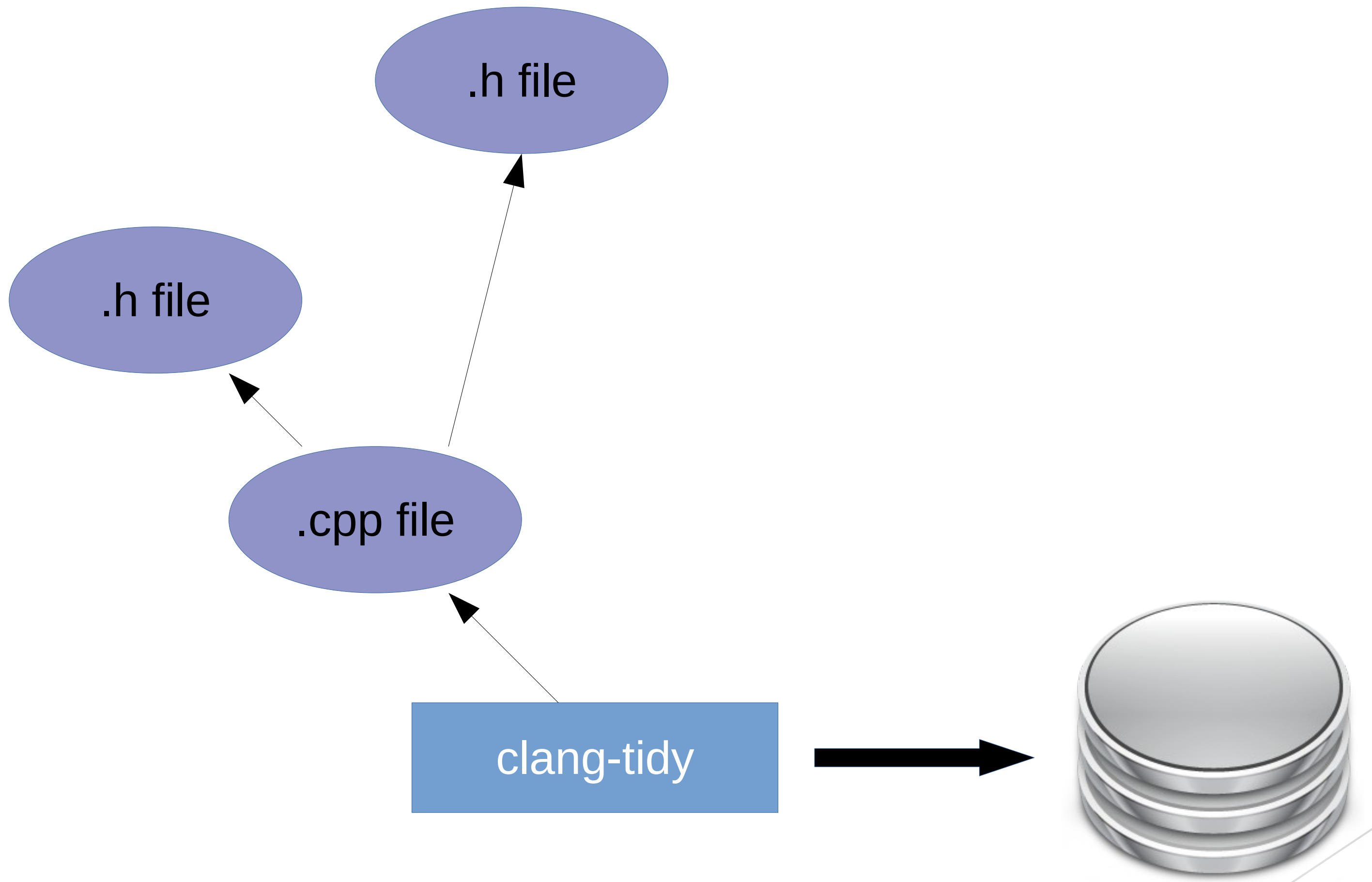
Replacements



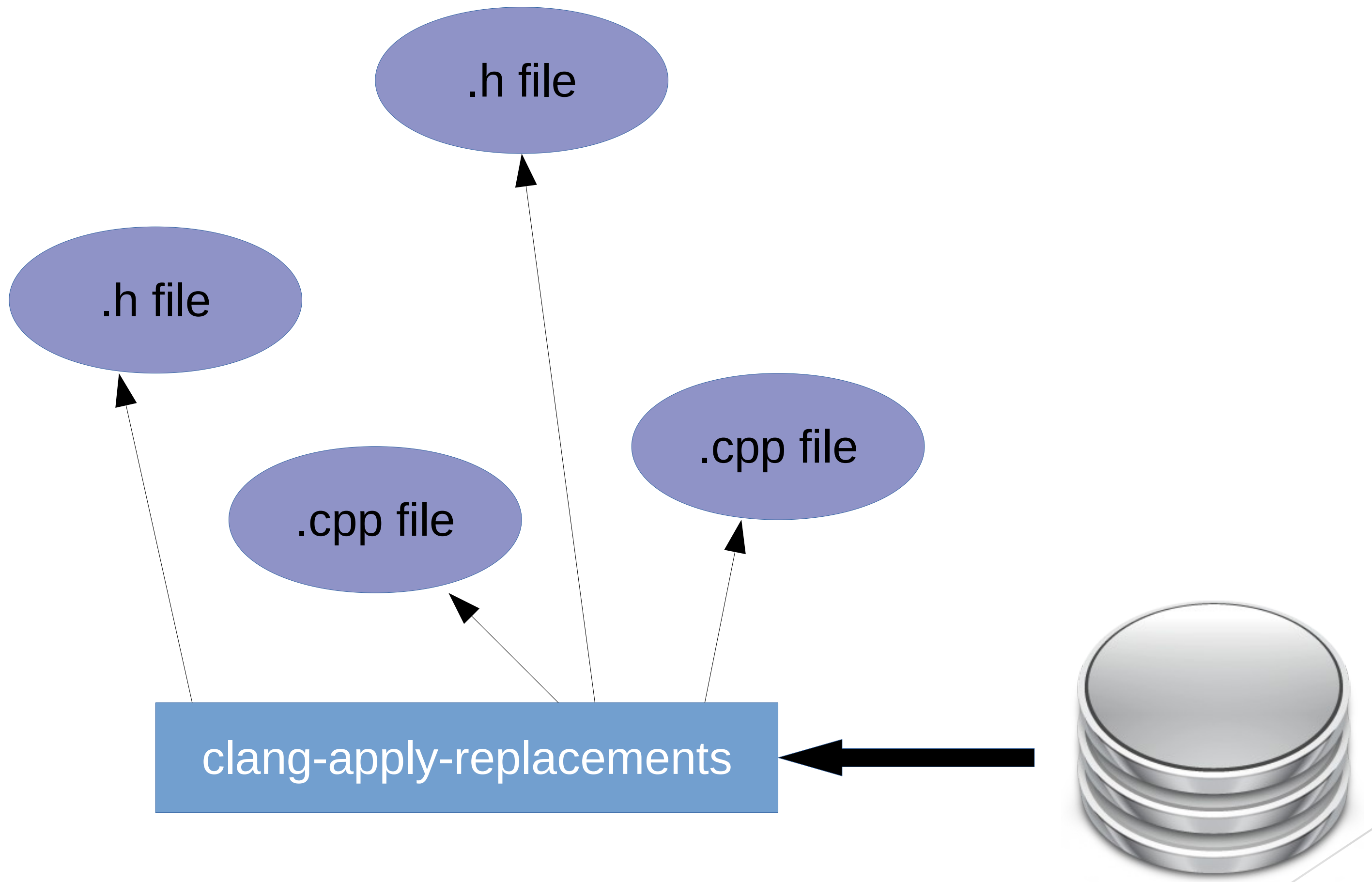
Replacements



Replacements



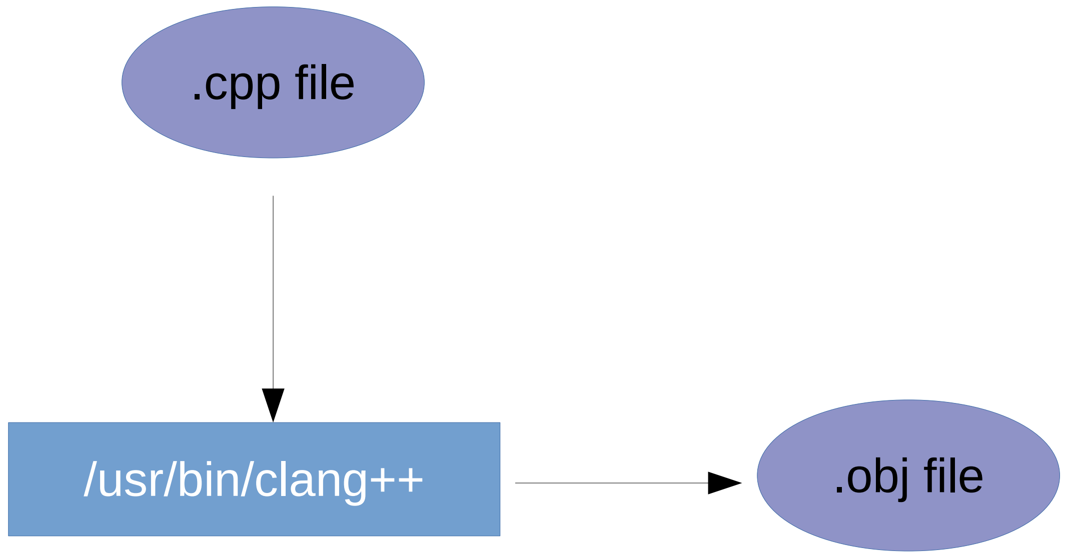
Replacements



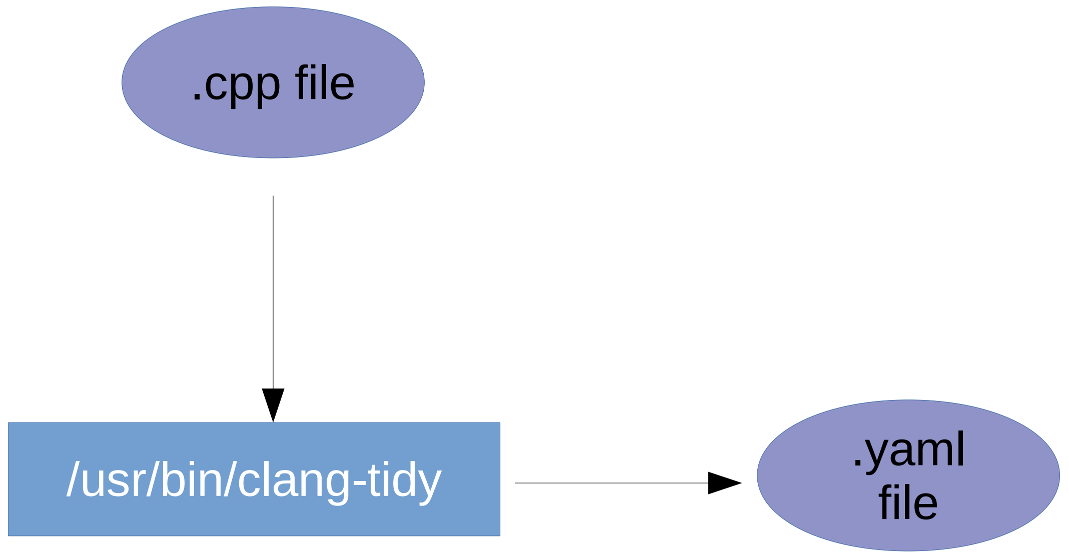
Scaling Tools

- `clang-tidy/run-clang-tidy.py`
  - Parallel `clang-tidy` runner
  - Operates on files matching pattern
  - Handles deferred replacement

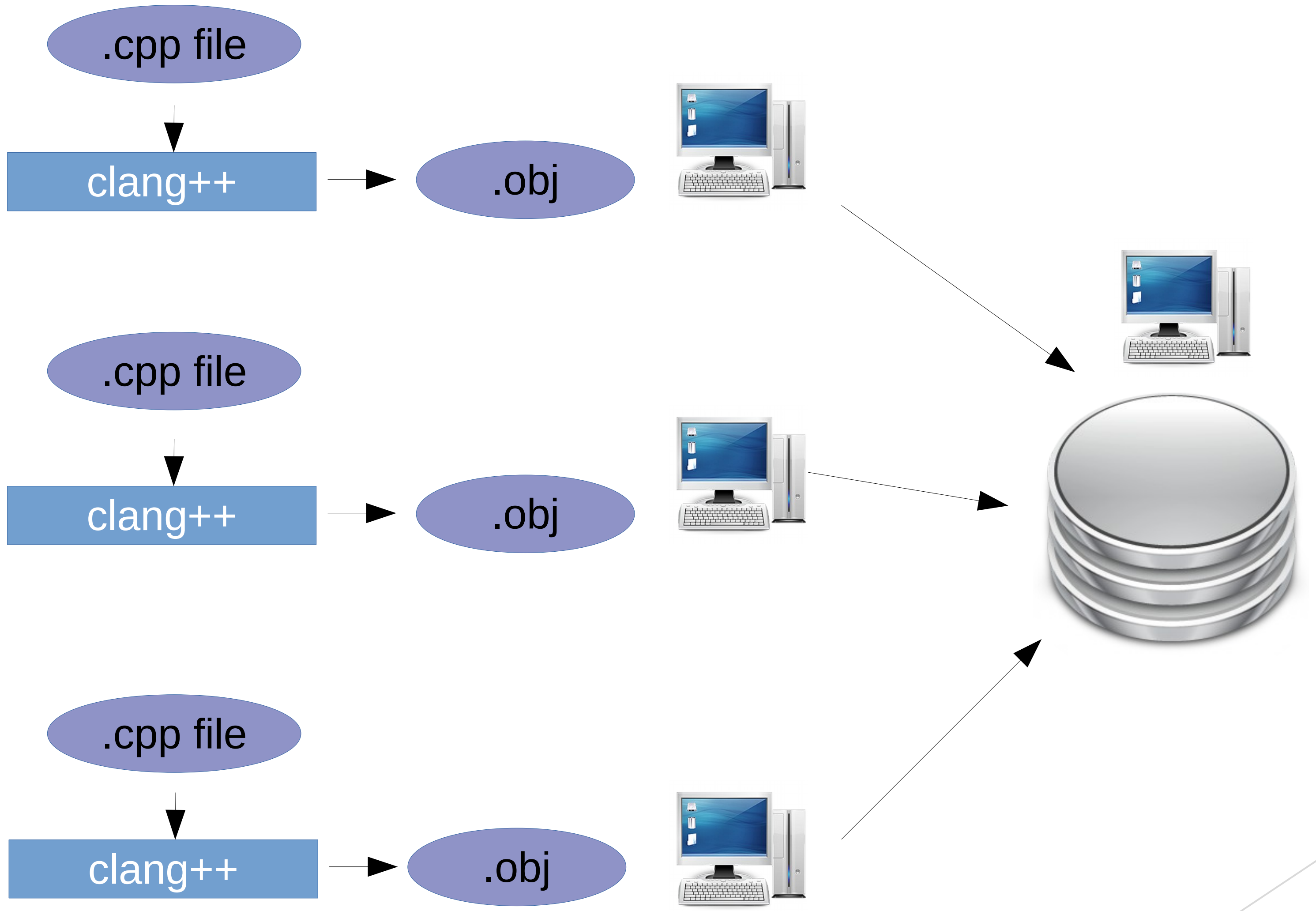
Scaling Tools



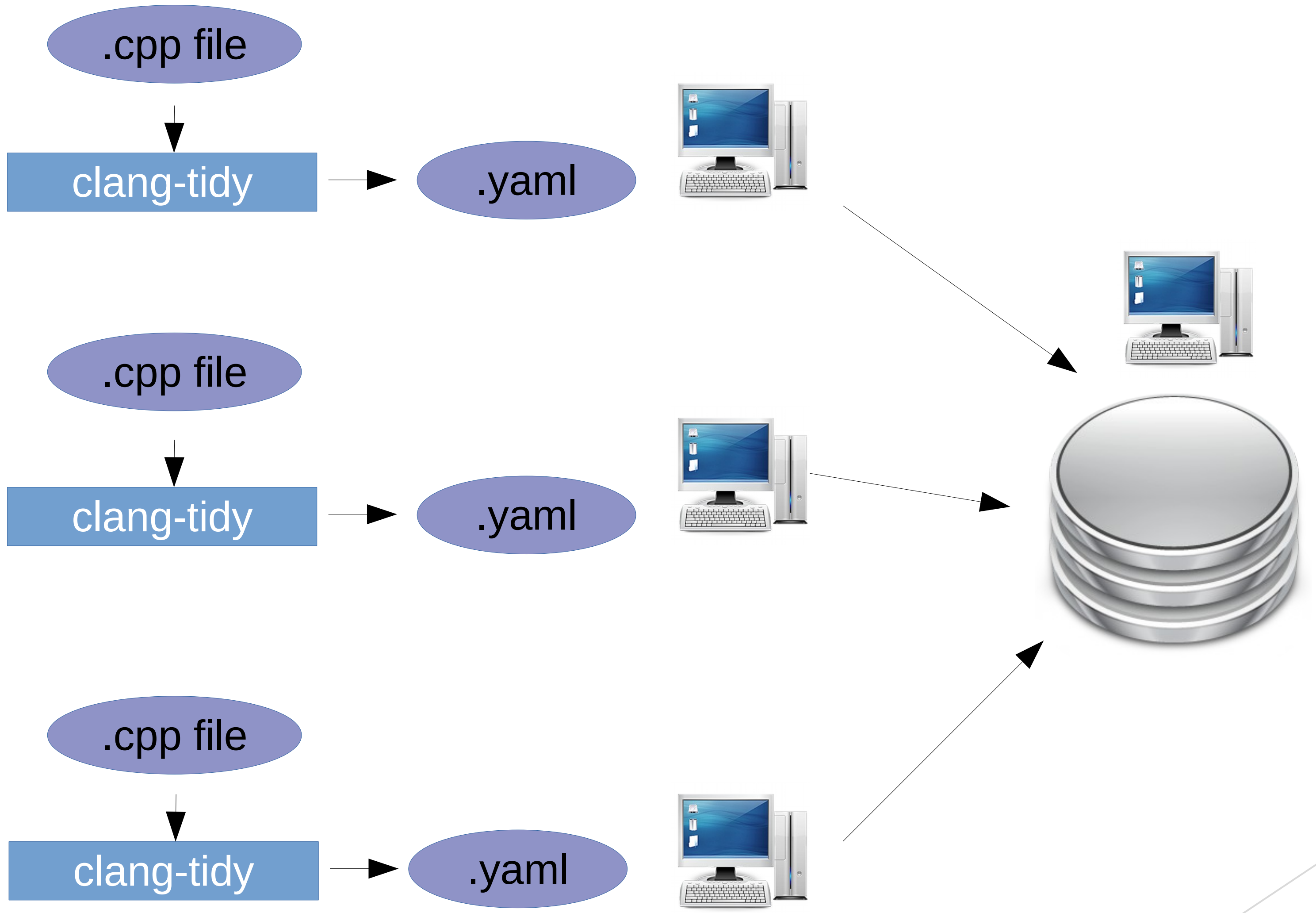
Scaling Tools



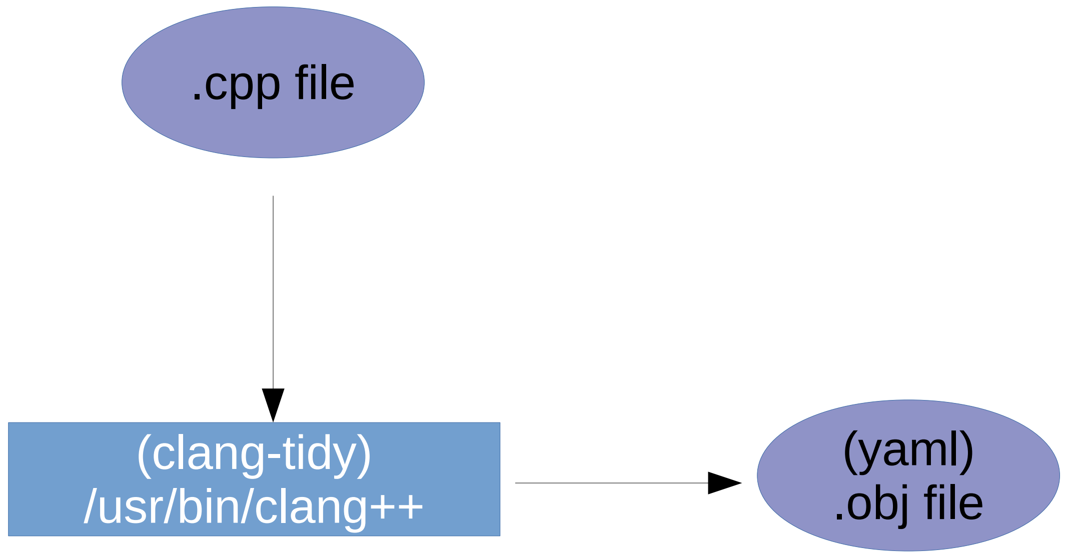
Build Distribution



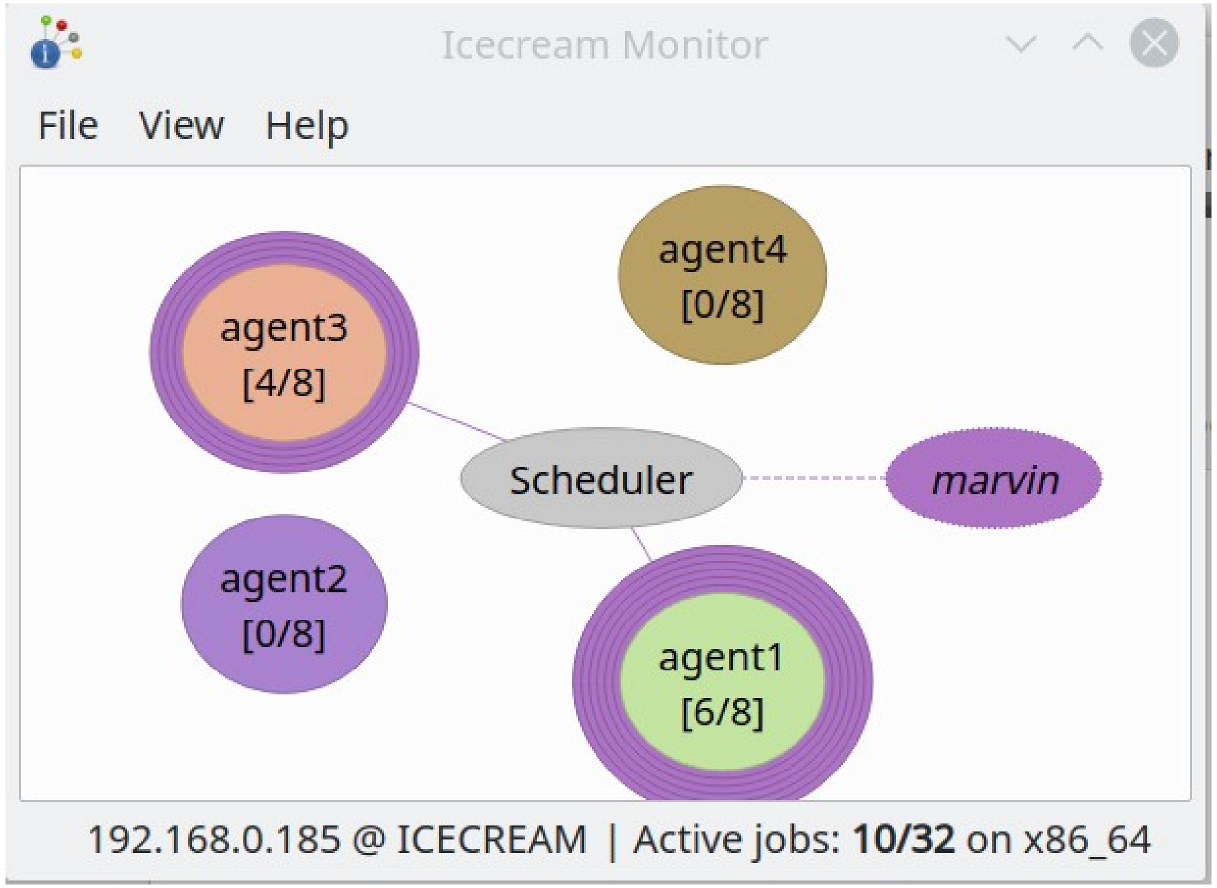
Build Distribution



Scaling Tools



Scaling Tools



192.168.0.185 @ ICECREAM | Active jobs: **10/32** on x86_64

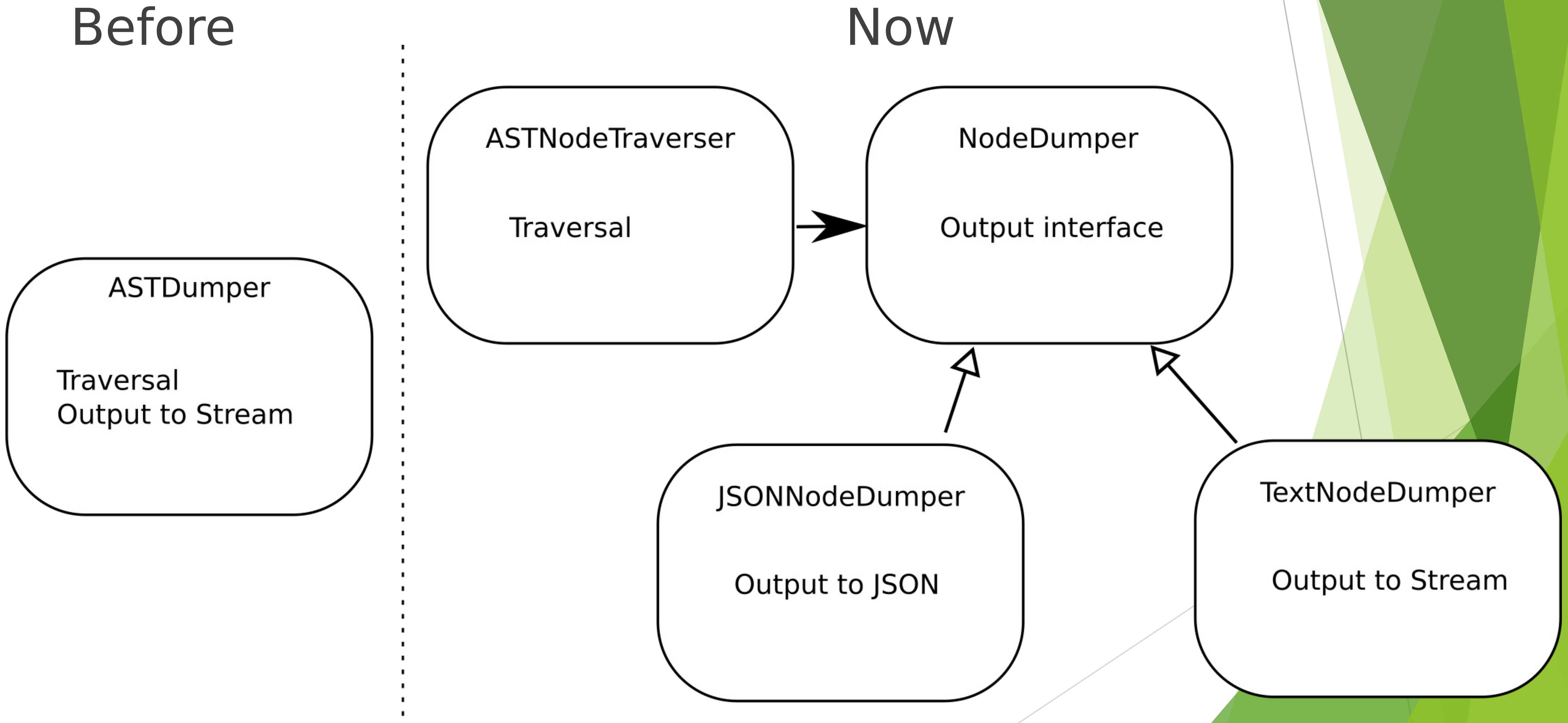
Build Issues

- Generated files (do a normal build first)
- Unity builds
- Precompiled headers
- Build distribution

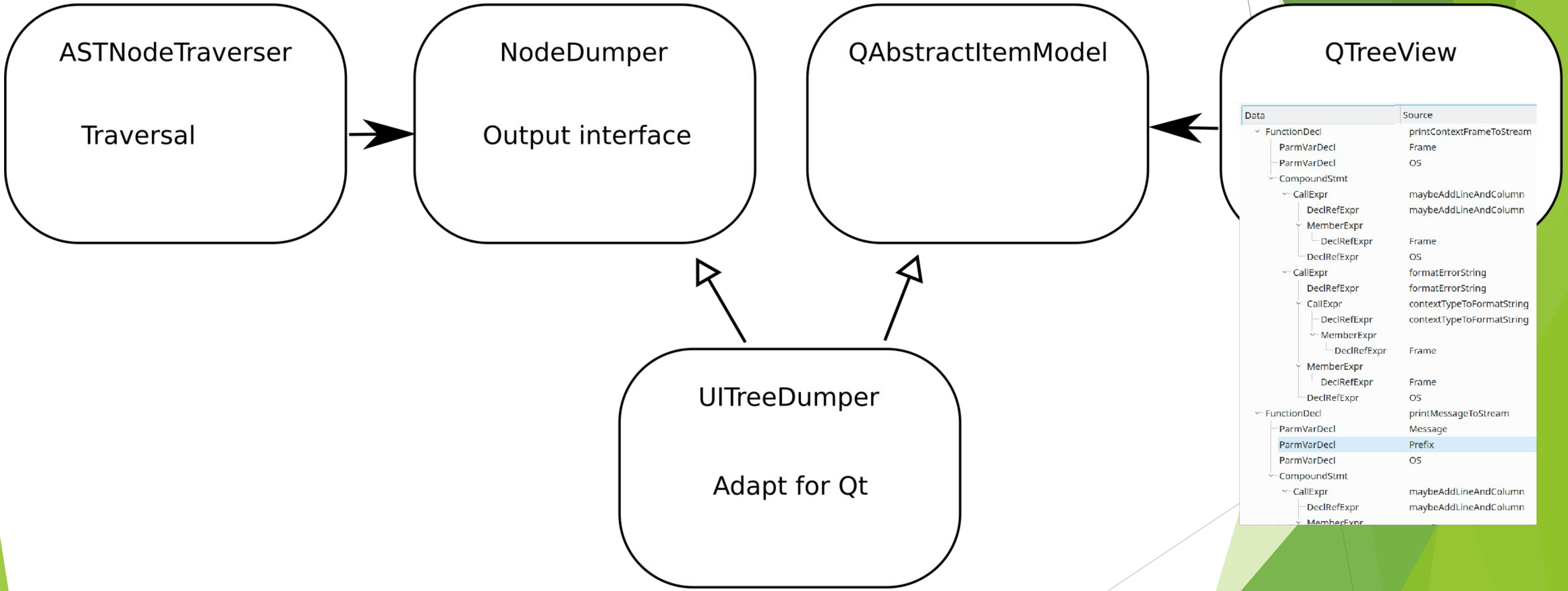
Upstreaming

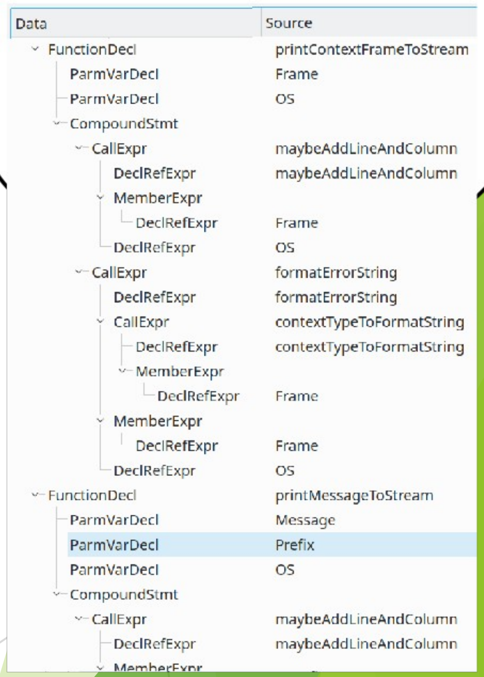
- New Features
  - Discovery in clang-query
  - Simpler output
- New APIs
  - DebuggingInterface
  - Output independent data
  - Enable New tools

Output independent APIs



Output independent APIs



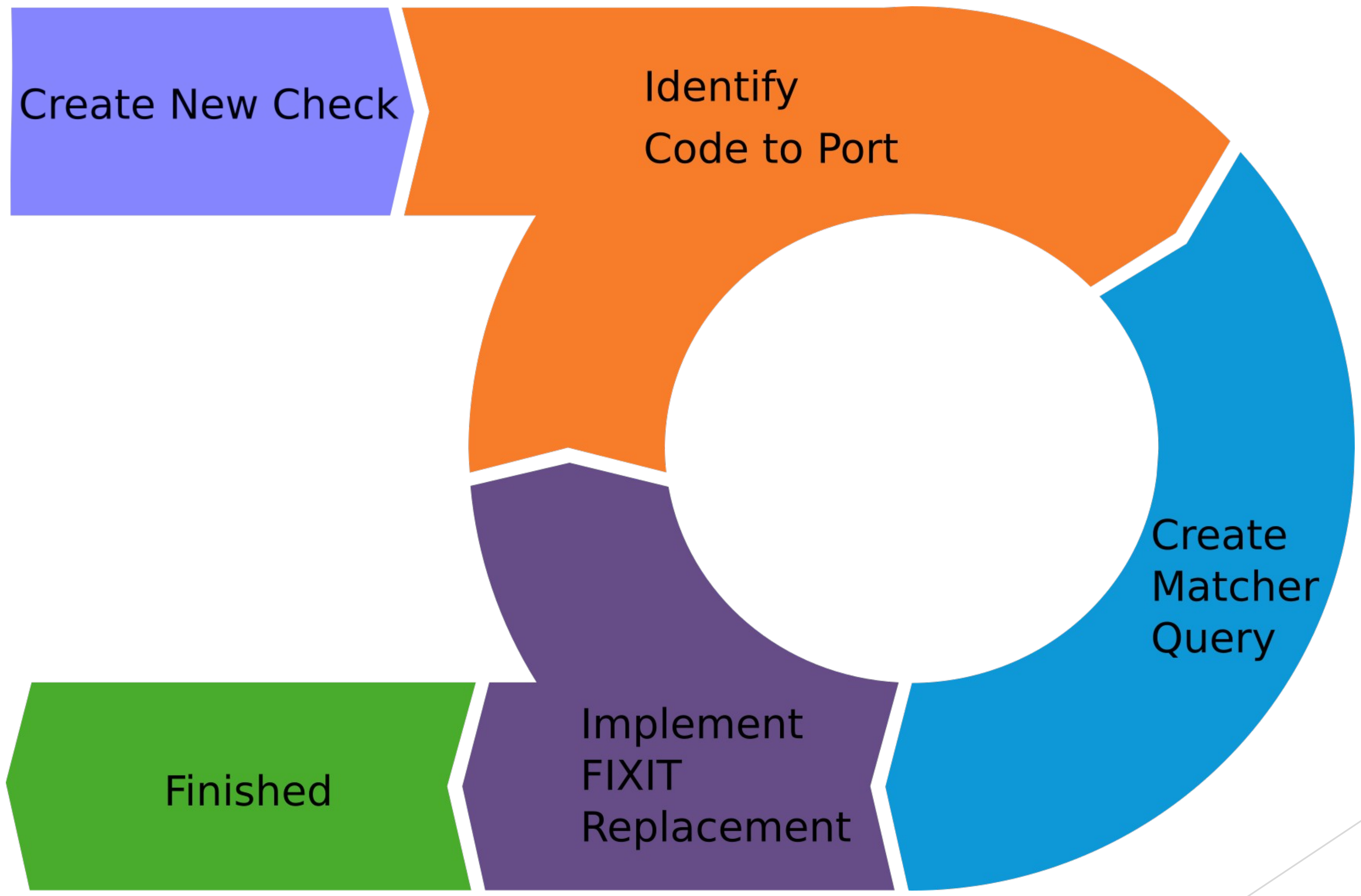


```
Data                              Source
⌄ FunctionDecl                    printContextFrameToStream
    ParmVarDecl                   Frame
    ParmVarDecl                   OS
  ⌄ CompoundStmt
      ⌄ CallExpr                  maybeAddLineAndColumn
          DeclRefExpr             maybeAddLineAndColumn
        ⌄ MemberExpr
            DeclRefExpr           Frame
          DeclRefExpr             OS
      ⌄ CallExpr                  formatErrorString
          DeclRefExpr             formatErrorString
        ⌄ CallExpr                contextTypeToFormatString
            DeclRefExpr           contextTypeToFormatString
          ⌄ MemberExpr
              DeclRefExpr         Frame
        ⌄ MemberExpr
            DeclRefExpr           Frame
          DeclRefExpr             OS
⌄ FunctionDecl                    printMessageToStream
    ParmVarDecl                   Message
    ParmVarDecl                   Prefix
    ParmVarDecl                   OS
  ⌄ CompoundStmt
      ⌄ CallExpr                  maybeAddLineAndColumn
          DeclRefExpr             maybeAddLineAndColumn
        ⌄ MemberExpr
```

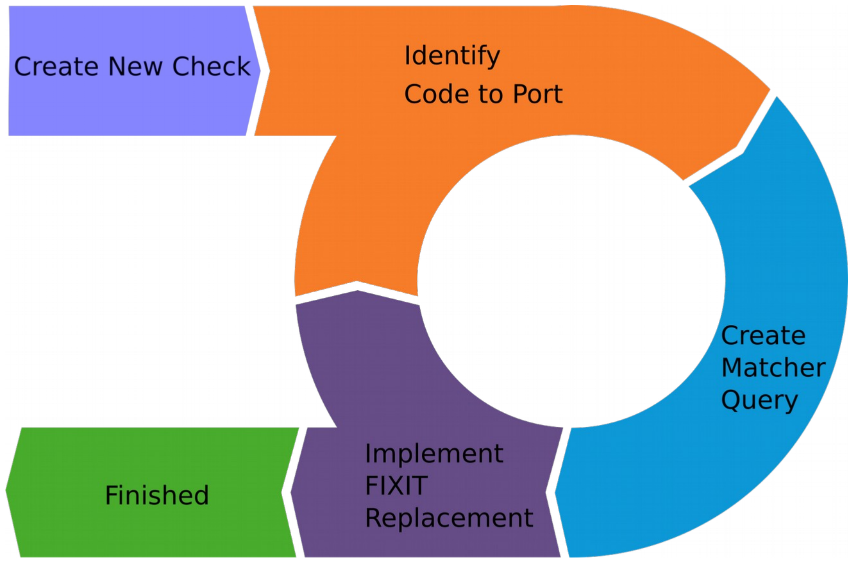
Demo

Demo (Quaplah)

Workflow (future)



Workflow (more-future)



/Tools

Tips, Tricks and Traps

AST classes

- Focus on porting Exprs
  - You have more of them
  - Decls tend to be easy
- Expressions reference declarations
  - Get familiar with all AST classes
  - Port Exprs based on type of Decls used
- Types are generally easy
  - Use `asString("class Foo")` from `clang-query`

CallExpr

```
void foo();
void bar();

void foo()
{
    bar();
}
```

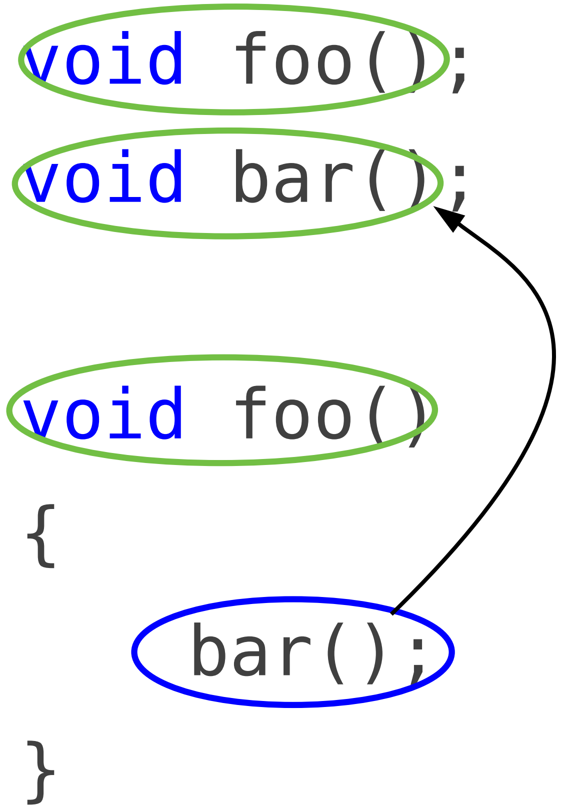
CallExpr

```
void foo();
void bar();

void foo()
{
    bar();
}
```

FunctionDecl

```
void foo();
void bar();

void foo()
{
    bar();
}
```

callExpr(callee(functionDecl()))



```
void foo();
void bar();

void foo()
{
    bar();
}
```

DeclRefExpr

```
int foo(int input)
{
    int i = input;
    for (int j = 0; j < 100; ++j)
        i += j;
    return i;
}
```

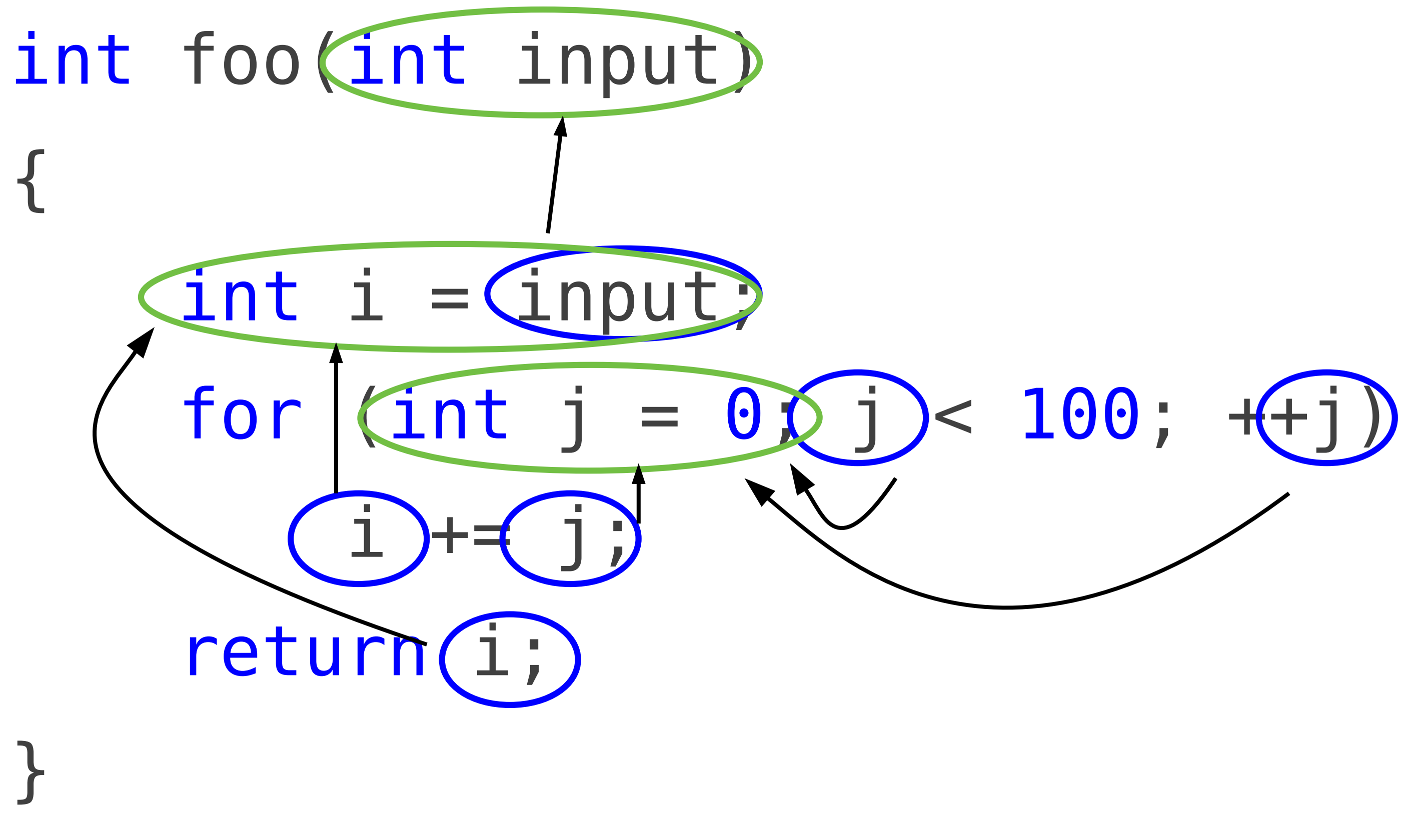
DeclRefExpr

```
int foo(int input)
{
    int i = input;
    for (int j = 0; j < 100; ++j)
        i += j;
    return i;
}
```

VarDecl (and ParmVarDecl)

```
int foo(int input)
{
    int i = input;
    for (int j = 0; j < 100; ++j)
        i += j;
    return i;
}
```

declRefExpr(to(varDecl()))



```
int foo(int input)
{
    int i = input;
    for (int j = 0; j < 100; ++j)
        i += j;
    return i;
}
```

MemberExpr

```
struct A
{
    int member = 0;
    int foo();
};
int A::foo()
{
    A a;
    a.member = 42;
    return member;
}
```

MemberExpr

```
struct A
{
    int member = 0;
    int foo();
};
int A::foo()
{
    A a;
    a.member = 42;
    return member;
}
```

FieldDecl

```
struct A
{
    int member = 0;
    int foo();
};
int A::foo()
{
    A a;
    a.member = 42;
    return member;
}
```

memberExpr(member(fieldDecl()))

```
struct A
{
    int member = 0;
    int foo();
};
int A::foo()
{
    A a;
    a.member = 42;
    return member;
}
```

Expr to Decl Traversal

Expr	Matcher	Decl
CallExpr	callee()	FunctionDecl
DeclRefExpr	to()	VarDecl
MemberExpr	member()	FieldDecl

- See Traversal Matchers documentation

Optional matches

```
cxxRecordDecl(
  hasMethod(
    cxxMethodDecl(hasName("foo")).bind("method")
    )
  ).bind("classDecl")
```

Optional matches

```
cxxRecordDecl(
  anyOf(
    hasMethod(
      cxxMethodDecl(hasName("foo")).bind("method"))
      ),
    anything()
    )
  ).bind("classDecl")
```

Optional matches

```
void MyFirstCheckCheck::check(...) {
  auto c = getNodeAs<Decl>("classDecl");
  if (auto m = getNodeAs<Decl>("method"))
  {
    // ...
  }
  // ...
}
```

Extending and Reuse

- Use local variables for block re-use

```
auto optionalFooMethod = anyOf(
    hasMethod(
      cxxMethodDecl(hasName("foo")).bind("method"))
      ),
    anything()
    );
```

Extending and Reuse

- Use local variables for block re-use

```
cxxRecordDecl(
  optionalFooMethod
  ).bind("classDecl")
```

Extending and Reuse

- Use functions for composition/decoration

```
auto optional = [](auto matcher) {
  return anyOf(
    matcher,
    anything()
    );
};
```

Extending and Reuse

- Use functions for composition/decoration

```
cxxRecordDecl(
  optional(hasMethod(
    cxxMethodDecl(hasName("foo")).bind("method"))
    ))
  ).bind("classDecl")
```

Extending and Reuse

- Use macros to extend predicate API

```
AST_MATCHER(VarDecl, isStaticDataMember)
{
  return Node.isStaticDataMember();
}
```

Extending and Reuse

- Use macros to extend predicate API

```
varDecl(isStaticDataMember()).bind("varDecl")
```

Evolution / Non-Atomic Refactoring

Evolution / Non-Atomic Refactoring

- You might not want one huge commit
  - Libraries with differing stability/customers
  - Hard to track down problems if CI fails
  - Hard to revert if needed
  - Mechanical patches need review too!

Evolution / Non-Atomic Refactoring

- Migrate files in particular directories
  - `clang-tidy -header-filter=PATTERN`
- Migrate particular entities
  - Local variables
  - Fields
  - Parameters
  - Return values

Reference Traps

```
void foo(MyString)
void bar(MyString&)

MyString s; // Converts to YourString
int i = foo(s);
int j = bar(s);
```

Reference Traps

```
struct A
{
    MyString const& getString() const;
private:
    MyString m_s;
};
```

Virtual Traps

```
struct ExternalBase {
    virtual void foo(MyString);
};

struct Derived : Base {
    void foo(MyString) override;
};
```

- cxxMethodDecl(unless(isOverride()))

Performance Trap

```
MyString s = getString(); // Returns YourString
processString(s); // Takes YourString
```

- Compiles, but might be slow!

Special Functions Trap

- How do we match

```
    if ( a == b ) {}
```

?

Special Functions Trap

-
  ```
  int a, b;
  return a == b;
  ```

-
  ```
  struct A {
    bool operator==(const A& other) { return true; }
  };
  ```

-
  ```
  struct A {};
  bool operator==(const A& lhs, const A& rhs) { return true; }
  ```

Special Functions Trap

```
bool foo(int a, int b)
{
    return a == b;
}
```

- `binaryOperator(hasOperatorName("=="))`

Special Functions Trap

```
struct A {
  bool operator==(A const& other) { return true; }
};

bool foo(A const& a1, A const& a2)
{
    return a1 == a2;
}
```

- `cxxOperatorCallExpr(hasOverloadedOperatorName("=="))`

Special Functions Trap

```
struct A {};
bool operator==(const A& l, const A& r) { return true; }



bool foo(A const& a1, A const& a2)
{
    return a1 == a2;
}
```

- `cxxOperatorCallExpr(hasOverloadedOperatorName("=="))`

Take-aways

- Large refactorings possible
  - Bespoke needs
  - In your code
- Improving in near future
  - Better tooling
  - Better collaboration

Summary

- Use clang-tidy for bespoke code transformations
- Use clang-query to discover AST Matchers
- Use the reference
  - http://clang.llvm.org/docs/LibASTMatchersReference.html
- Distribute workload with build-distribution system

Resources / Questions

- @steveire / steveire.wordpress.com
- ce.steveire.com
- https://blogs.msdn.microsoft.com/vcblog
- StackOverflow `[clang-ast-matchers]`
- Learn, Share and Blog!

```
match questionDecl(
    hasAnswer(clearExpr().bind("Answer"))
    )

void check(auto const& Result)
{
    auto Answer =
        Result.Nodes->getAs<ClearExpr>("Answer");
    Answer->dump();
}
```