Implementing Include-what-you-use Using Clang

Craig Silverstein
Director of Technology, Google Inc.

4 November 2010

Summary

Google is developing a tool called include what you use. It analyzes symbols and types used in C++ source files, using clang.

Only analyzes source code — uses `RecursiveASTVisitor` heavily. No code generation.

Considered: gcc dehydra, gccxml, eclipse C++ frontend, KDeveloper C++ parser, klockwork, synopsis, EDGcpfe, clang.

Implemented: dehydra and clang.

clang is better suited to this task than gcc-based dehydra, but could be even better.

What is Include What You Use?

IWYU: the principle that if you use a symbol or type from a .h file, you should include that .h file.

foo.cc:

```
fprintf(stderr, "hello");  // uses <stdio.h>
typedef std::set<int8_t> IntSet;  // uses <set>, <stdint.h>
if (FnReturningVector().empty()) ...;  // uses <vector>
#if __WORDSIZE == 64  // uses <bits/wordsize.h>
```

- Always #include necessary .h files *directly*.
- Do not #include unnecessary .h files at all.

Why Include What You Use?

- Refactoring: can remove unneeded #includes from .h files.
- Obsoleting: easily find all clients of a library.
- Dependency breaking: can remove dependency on libraries we don't use anymore.

To maximize dependency breaking, we prefer forward declarations to #includes whenever possible.

Implementation #1: Dehydra

Dehyra gets callbacks from gcc every time a symbol and function is parsed. Available to clients (iwyu) via javascript bindings.

Challenges:

- gcc collapses function declarations and definition.
- Only see instantiated template classes/functions — and they're attributed to the declaration site.
- No way to distinguish template params in templated code.
- No access to preprocessor output (implemented our own preprocessor).
- Debugging javascript.

A local gcc expert could hack on gcc and dehydra to resolve issues. But template problems were a dealbreaker.

Include What You Use is Surprisingly Difficult

foo.h:

```
typedef vector<int>::iterator RegionIterator;
inline RegionIterator RegionBegin() { ... }
```

foo.cc:

```
#include "foo.h"
RegionIterator it = RegionBegin();  // "uses" <vector>?
```

bar.h:

```
template<class A, class B=ClassFromBazH>  MyClass;
```

bar.cc:

```
MyClass<int> a;             // "uses" ClassFromBazH?
hash_set<MyClass<int> > b;  // "uses" hash<MyClass<int> >?
```

Implementation #2: Clang

Needed to wait until C++ support was sufficiently advanced.

Needed to flesh out dgregor's `RecursiveASTVisitor`.

Needed better `TypeLoc` support in clang.

Still need better preprocessor support: no `PPCallbacks` hooks for #if or #ifdef.

iwyu sometimes gets confused due to lack of `TypeLoc` (only big trouble spot left is `NestedNameSpecifier`).

Overall, clang is very clean, and AST structure is a natural fit for iwyu. (Though traversing it requires a lot of casting!)

How IWYU Works

Basic idea:

- Traverse the AST to find all uses of a symbol.
- Use `getDecl()` to find the declaration.
- If they are in different files, mark an IWYU constraint.

Sample complications:

- Also need to capture uses of types. These are often not explicit in the AST.
- There may be many declarations, need to canonicalize.
- The declaration may be in a private header file, so we need to canonicalize that too — a manual process. Or the declaration may be a built-in (`new` vs `placement-new`).

AST Utilities

- ASTNode: a union of all possible AST node types: Decl, Stmt, Type/TypeLoc, TemplateArgument/TemplateArgumentLoc, TemplateName, NestedNameSpace. Critically, it also knows its parent in the AST tree. It has clever location-determining logic.

- ASTNode helpers: logic on an AST node (often involves parents). e.g. IsDefaultTemplateTemplateArg ("you are a TemplateName, parent is a TemplateArgument").

- Decl helpers: e.g. HasImplicitConversionCtor (for CXXRecordDecl).

- Type helpers: TypeToDecl is key (tricky: needs to remove subst-template type params, elaborations, etc). Also: RemoveElaboration, RemovePointerFromType (follows typedefs only if necessary).

Finding Uses (Excepting Templates)

In these examples, a variable named `a` has type `A`.

`stderr`, etc.	needs defn of symbol
`a->b->c`	needs defn of `A` and `B`
`a->b()`	needs defn of `A`, *and* needs defn of `b()` (!)
`delete x`	needs defn of `X` and of some `operator delete`
`new X`	uses some `operator new`
`namespace a=b`	needs defn of `b`
`using ns::a`	needs declaration of all `ns::a`'s (may be overloaded)
`typedef A B`	needs **defn** of `A` ("re-exports" `A`)
`X x`	needs defn of `X`
`X* x`	needs declaration of `X` (`class X* x` needs nothing!)
`#define A B`	needs definition of `B` (TODO if `B` is not a macro)
`#if sizeof(A)`	needs definition of `A` (TODO)

Finding Uses (Templates)

In these examples, variable a has type `TplClass<A>`.

`MyClass<X>`	needs either defn or declaration of `X`
`vector<X>`	needs defn of `X` does *not* need defn of `std::allocator<X>`
`scoped_ptr<X>`	needs declaration (only) of `X`
`hash_map<X>`	needs defn of `hash<X>` (in addition to `X`)

`template<>`
`struct Foo<int>`
	needs declaration of `Foo<T>`

`a.foo()`	must evaluate `foo()` to see if needs defn of `A`
`delete a`	must evaluate `~MyClass<A>`() plus dtor of parents
`sizeof(C<A>)`	must evaluate fields of `C`
`C<A>()`	must evaluate fields of `C` *and* ctor *and* initializers
`C<A*>()`	must still evaluate (for uses of `*A`)

When Forward-Declaring Isn't Enough

Usually just need declarations of pointer/reference types. *But...*

- `MyClass::MyClass(const Foo& foo); // implicit conversion`
- `MyClass::MySubclass* s;  // nested-name-specifier use`

By default just need declarations of template parameters. But if they're used... (And don't forget to check uses like `C<A>::value_type`)

Figuring out if template template parameters can be forward-declared or not, makes my head hurt.

On Beyond Uses

Other situations we keep an #include or forward declare:

- #include of a `.c` file
- #include of an associated, private `.h` file
- Forward-declare with an `__attribute__` or linkage spec
- `// NOLINT(iwyu)`
- In code clang doesn't see (`#if 0 ...`)

Public and Private

If we use a symbol defined in `<bits/stl_vector.h>`, we put the iwyu constraint on `<vector>`.

If we use `NULL`, there are 14 files defining it. We pick to minimize changes.

A hard-coded list:

- 165 mappings for glibc C++
- 152 mappings for glibc C
- 113 mappings for C/C++ symbols
- 17 mappings for third-party code
- 23 for Google code

There can be chains of mappings: `<bits/ios_base.h>` → `<ios>` → `<iostream>`. There can be optional stopping points (`<ios>` above).

Notes on Working with Clang

Go-to helpdesk: IRC channel. (Thanks to dgregor, rjmccall, nlewycky, and others who have patiently helped me out!)

Go-to reference: doxygen documentation on the AST class hierarchy.

Doxygen wishlist: Top-of-class example code snippet:

```
/// foo in: foo<bar, baz>();  // function call
/// foo in: printf(foo);      // variable use
class DeclRefExpr { ...
```

Per-method example code snippet:

```
/// Goes from decl2 to decl1 in this code snippet:
///   template<typename T> class Foo { ... };  // decl1
///   template<> class Foo<int> { ... };       // decl2
```