

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 sub-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`.

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

Finding Uses (Templates)

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

<code>MyClass<X></code>	needs either defn or declaration of <code>X</code>
<code>vector<X></code>	needs defn of <code>X</code> does <i>not</i> need defn of <code>std::allocator<X></code>
<code>scoped_ptr<X></code>	needs declaration (only) of <code>X</code>
<code>hash_map<X></code>	needs defn of <code>hash<X></code> (in addition to <code>X</code>)
<code>template<></code> <code>struct Foo<int></code>	needs declaration of <code>Foo<T></code>
<code>a.foo()</code>	must evaluate <code>foo()</code> to see if needs defn of <code>A</code>
<code>delete a</code>	must evaluate <code>~MyClass<A>()</code> plus dtor of parents
<code>sizeof(C<A>)</code>	must evaluate fields of <code>C</code>
<code>C<A>()</code>	must evaluate fields of <code>C</code> <i>and</i> ctor <i>and</i> initializers
<code>C<A*>()</code>	must still evaluate (for uses of <code>*A</code>)

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>` $\rightarrow$ `<ios>` $\rightarrow$ `<iostream>`. There can be optional stopping points (`<ios>` above).

Notes on Working with Clang

[Go-to helpdesk](#): IRC channel. (Thanks to dgregor, rjmccall, nlewicky, 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
```